

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ
РОССИЙСКОЙ ФЕДЕРАЦИИ

ФГАОУ ВО «Крымский федеральный университет
имени В. И. Вернадского»

С. П. ГОРЛЯНСКИЙ

**РЕШЕНИЕ ЗАДАЧ
НА ЯЗЫКЕ ПРОГРАММИРОВАНИЯ ЛИСП**

Учебное пособие

Симферополь
2016

УДК 004.432.42
ББК 32.973-018.1
Г69

Рекомендовано к печати учебно-методическим советом ФГАОУ ВО «Крымский федеральный университет имени В. И. Вернадского» от 28 апреля 2016 г., протокол № 11. Рекомендовано к печати учебно-методическим советом Таврической Академии (СП) ФГАОУ ВО «Крымский федеральный университет имени В. И. Вернадского» от 13 апреля 2016 г., протокол № 2.

Рецензенты:

В. И. Донской, доктор физико-математических наук, профессор, заслуженный деятель науки и техники Украины;

А. И. Козлов, кандидат технических наук, доцент.

Горлянский С. П.

Решение задач на языке программирования Лисп: Учебное пособие / С. П. Горлянский. ФГАОУ ВО «Крымский федеральный университет имени В. И. Вернадского». – Симферополь, 2016. – 1248 с.

Данное пособие является частью мультимедийной системы подготовки обучающихся по направлению подготовки «история», составлено в соответствии с программой курса «Информационные технологии в исторических исследованиях и образовании», содержит варианты решений учебных задач на языке программирования Лисп (ANSI Common Lisp) и предназначено для промежуточного контроля знаний и навыков, а также самопроверки и самоподготовки студентов-историков.

УДК 004.432.42
ББК 32.973-018.1

© С. П. Горлянский, 2016

Оглавление

Предисловие.....	45
Установка программного обеспечения.....	47
Команды редактора Emacs SLIME.....	55
Файлы.....	55
Буфер.....	55
Курсор.....	55
Редактирование.....	56
Поиск.....	56
Замена.....	56
Завершение.....	56
Заккрытие.....	56
Отступы.....	57
Компиляция.....	57
Отладчик.....	57
Навигация.....	57
Команды.....	57
Макрос.....	57
Пакеты.....	58
Синтаксис.....	59
ТИПЫ ДАННЫХ.....	61
Символы.....	61
Перевод строки в символ.....	63
Логические значения t и nil.....	63
Числа.....	63
Перевод строки в число.....	64
Перевод знака в число.....	64

Буквы.....	64
Перевод строки в список букв.....	64
Список кодов букв строки	65
Строки	65
Сравнение строк	65
Перевод символа в строку	65
Перевод числа в строку.....	65
Перевод списка букв в строку	66
Поиск подстроки в строке	66
Присоединение буквы к строке.....	66
Соединение строк.....	66
Сортировка знаков строки	66
Список слов строки	66
Список однобуквенных подстрок строки.....	66
Возврат части строки	67
Последовательности.....	67
ФУНКЦИИ.....	69
Базовые функции	69
Основные функции обработки списков.....	69
Функции проверки	70
Стандартные функции	71
Вложение функций.....	73
Стандартные функции (продолжение)	74
Логические функции	77
Ветвление вычислений	77
Создание (определение) функций.....	78
Рекурсивные функции.....	79
Простая рекурсия	79
Хвостовая рекурсия.....	80
Параллельная рекурсия.....	80
Взаимная рекурсия	81
Функция MAPCAR.....	81
Макрос LAMBDA	82

Макрос LOOP	82
Функция FORMAT	83
Примеры решений	85
Структура 1 (функция ATOM) > ATOM.....	85
Задача 1.1 inc.lisp.....	85
Задача 1.2 dec.lisp	85
Задача 1.3 1000+.lisp	86
Задача 1.4 triple-number.lisp.....	86
Задача 1.5 years-seconds.lisp	86
Задача 1.6 julian-years-seconds.lisp	87
Задача 1.7 gregorian-years-seconds.liisp.....	87
Задача 1.8 digits-expt-125-100.lisp	88
Задача 1.9 romanp.lisp	90
Задача 1.10 special-pythagorean-triplet-product.lisp.....	92
Задача 1.11 trim-words.lisp	94
Задача 1.12 remove-extra-spaces.lisp	95
Задача 1.13 max-lets.lisp	96
Задача 1.14 dump-duplicates.lisp	99
Задача 1.15 chess-prize.lisp.....	101
Задача 1.16 sort-vowels<.lisp	101
Задача 1.17 select-words.lisp.....	102
Задача 1.18 string-upcase.lisp.....	103
Задача 1.19 2exptp.lisp.....	103
Задача 1.20 factorial.lisp	104
Задача 1.21 factorial+.lisp	107
Задача 1.22 char-sum-code+.lisp	108
Задача 1.23 $2/1+3/2\ldots+[n+1]/n$.lisp	109
Задача 1.24 num-magic-ticket-9999.lisp	109
Задача 1.25 sum-odd-plus.lisp.....	110
Задача 1.26 longest-palindrome.lisp.....	111
Задача 1.27 remove-repetition.lisp	112
Задача 1.28 set-length.lisp.....	114
Задача 1.29 display.lisp	114

Задача 1.30	num-substr.lisp	115
Задача 1.31	sum-divisors.lisp	115
Задача 1.32	min-shuffle.lisp	116
Задача 1.33	capitalize-line.lisp	117
Задача 1.34	deci-to-hex.lisp	118
Задача 1.35	hex-to-bin.lisp	118
Задача 1.36	ration.lisp	119
Задача 1.37	extract-nums.lisp	120
Задача 1.38	count-divisors.lisp	120
Задача 1.39	adjust-subtitles.lisp	121
Задача 1.40	decode-phone-n.lisp	122
Задача 1.41	reverse-digits.lisp	124
Задача 1.42	remove-extra-spaces.lisp	125
Задача 1.43	swap-lets.lisp	126
Задача 1.44	factorial-of.lisp	127
Задача 1.45	min-network-connections-length.lisp	130
Задача 1.46	fibonacci.lisp	131
Задача 1.47	ctg.lisp	133
Задача 1.48	circle-area.lisp	133
Задача 1.49	dot-lowercase.lisp	134
Задача 1.50	sum-natural.lisp	136
Задача 1.51	sum-natural-even.lisp	136
Задача 1.52	single-elimination-wins.lisp	137
Задача 1.53	largest-prime-factor.lisp	138
Задача 1.54	sum-square-difference.lisp	139
Задача 1.55	count-word.lisp	140
Задача 1.56	archive-string.lisp	140
Задача 1.57	reap-reverse.lisp	141
Задача 1.58	sum-square-reaper.lisp	141
Задача 1.59	deci.lisp	142
Задача 1.60	count-reversible-numbers.lisp	143
Задача 1.61	reverse-sentence.lisp	144
Задача 1.62	sum-evenn.lisp	145

Задача 1.63	replace-ing-ed.lisp.....	146
Задача 1.64	product-even.lisp.....	146
Задача 1.65	n!+n^20.lisp.....	147
Задача 1.66	last-number.lisp.....	147
Задача 1.67	eeeq.lisp.....	148
Задача 1.68	min-lets.lisp.....	152
Задача 1.69	bis-sexstus.lisp.....	155
Задача 1.70	newton-2.lisp.....	156
Задача 1.71	newton-3.lisp.....	156
Задача 1.72	+digit.lisp.....	157
Задача 1.73	++digit.lisp.....	157
Задача 1.74	punctuation-marks.lisp.....	158
Задача 1.75	reverse-parts.lisp.....	159
Задача 1.76	fahrenheit-celsius.lisp.....	160
Задача 1.77	celsius-fahrenheit.lisp.....	160
Задача 1.78	calculate.lisp.....	161
Задача 1.79	sum.lisp.....	162
Задача 1.80	num-digit.lisp.....	162
Задача 1.81	cnt.lisp.....	163
Задача 1.82	srt.lisp.....	163
Задача 1.83	reap-expt.lisp.....	164
Задача 1.84	maxxx.lisp.....	165
Задача 1.85	max-divisor.lisp.....	166
Задача 1.86	card-deck-factorial.lisp.....	166
Задача 1.87	eai-most-frequentp.lisp.....	167
Задача 1.88	odd-seqp.lisp.....	168
Задача 1.89	product-digit.lisp.....	170
Задача 1.90	lucky-tktp.lisp.....	170
Задача 1.91	is-power-of-2.lisp.....	171
Задача 1.92	mini-max.lisp.....	172
Задача 1.93	convert.lisp.....	173
Задача 1.94	same-length-word-textp.lisp.....	173
Задача 1.95	fish.lisp.....	174

Задача 1.96 0102-string.lisp	175
Задача 1.97 mirror.lisp.....	176
Структура 2 (функция АТОМ) > (СПИСОК)	177
Задача 2.1 numbers.lisp	177
Задача 2.2 numbers-in-reverse.lisp.....	178
Задача 2.3 balanced.lisp.....	180
Задача 2.4 int>digits.lisp	181
Задача 2.5 squares.lisp.....	182
Задача 2.6 prin&list.lisp	182
Задача 2.7 deci-bin.lisp.....	183
Задача 2.8 bin-deci.lisp.....	184
Задача 2.9 drop-word-duplicates.lisp.....	184
Задача 2.10 prime2n-1.lisp	185
Задача 2.11 count-letters.lisp.....	185
Задача 2.12 print-linen.lisp.....	186
Задача 2.13 odd50.lisp.....	186
Задача 2.14 f.lisp	187
Задача 2.15 magic-ticket-9999.lisp.....	188
Задача 2.16 magic-ticket-222222.lisp.....	189
Задача 2.17 sum-20.lisp.....	190
Задача 2.18 sum-n.lisp.....	190
Задача 2.19 onion-n.lisp	191
Задача 2.20 factorials.lisp.....	191
Задача 2.21 dice-game.lisp	192
Задача 2.22 rand.lisp.....	193
Задача 2.23 cumulate.lisp	194
Задача 2.24 divisors.lisp.....	196
Задача 2.25 suffixes.lisp	196
Задача 2.26 bin-deci.lisp.....	197
Задача 2.27 fn>*123.lisp	197
Задача 2.28 res.lisp	197
Задача 2.29 amicable-nums.lisp.....	198
Задача 2.30 n-234.lisp	200

Задача 2.31	lets-digs.lisp	201
Задача 2.32	found-more-than-once.lisp	201
Задача 2.33	same-let-words.lisp	203
Задача 2.34	word-length.lisp	204
Задача 2.35	reaper.lisp.....	204
Задача 2.36	prime-divisors.lisp.....	209
Задача 2.37	reversible-numbers.lisp	209
Задача 2.38	symb-freq-percent.lisp	210
Задача 2.39	explode.lisp	213
Задача 2.40	fibonacci.lisp.....	213
Задача 2.41	count-letters.lisp.....	213
Задача 2.42	100m.lisp.....	214
Задача 2.43	200m.lisp.....	214
Задача 2.44	400m.lisp.....	214
Задача 2.45	square-digits.lisp	215
Задача 2.46	digi-div-n.lisp.....	215
Задача 2.47	digi-div.lisp	216
Задача 2.48	fibonacci-sequence.lisp	219
Задача 2.49	doors.lisp.....	220
Задача 2.50	count-word.lisp	221
Задача 2.51	run-test.lisp.....	224
Задача 2.52	1357...2*n-1.lisp.....	225
Задача 2.53	product-odd-number-odd.lisp.....	226
Задача 2.54	cubed.lisp	226
Задача 2.55	collect-lowercase.lisp.....	227
Задача 2.56	roman-alpha.lisp.....	230
Задача 2.57	sum-number-digit.lisp	232
Задача 2.58	consonant.lisp.....	232
Задача 2.59	armstrong-decimal-number.lisp	233
Задача 2.60	longest-roman-prime.lisp	239
Задача 2.61	split.lisp.....	241
Задача 2.62	max-length-number.lisp	242
Задача 2.63	perfect.lisp.....	243

Задача 2.64 invert-deci-bin.lisp	244
Задача 2.65 abundant-word.lisp	245
Задача 2.66 s-t-r-i-n-g.lisp	246
Задача 2.67 sierpinski-carpet.lisp	246
Задача 2.68 sd.lisp	247
Задача 2.69 queens.lisp	248
Задача 2.70 cleave.lisp	249
Задача 2.71 snail.lisp	250
Задача 2.72 [[a][ba][aba]].lisp	259
Задача 2.73 0102-list.lisp	260
Структура 3 (функция (СПИСОК)) > ATOM	261
Задача 3.1 cantinier.lisp	261
Задача 3.2 count-elm.lisp	262
Задача 3.3 sum137.lisp	263
Задача 3.4 sum-odd-plus.lisp	264
Задача 3.5 rare-num.lisp	265
Задача 3.6 subp.lisp	266
Задача 3.7 uno-atom.lisp	267
Задача 3.8 leader-atom.lisp	269
Задача 3.9 num-sym-3rd.lisp	269
Задача 3.10 some-odd.lisp	270
Задача 3.11 last-odd.lisp	270
Задача 3.12 best-marks.lisp	270
Задача 3.13 average-height.lisp	271
Задача 3.14 max-height.lisp	271
Задача 3.15 counter.lisp	271
Задача 3.16 last-atom.lisp	272
Задача 3.17 number-max.lisp	273
Задача 3.18 almost-last.lisp	273
Задача 3.19 palindrome.lisp	274
Задача 3.20 not-equal-last.lisp	274
Задача 3.21 sum-number.lisp	274
Задача 3.22 equal-set.lisp	276

Задача 3.23	sum-plusp.lisp	277
Задача 3.24	tops-number.lisp	277
Задача 3.25	sum-unique.lisp	277
Задача 3.26	our-car-cdaddr.lisp	278
Задача 3.27	descend.lisp	278
Задача 3.28	our-caddr.lisp	279
Задача 3.29	count-low-atoms.lisp	279
Задача 3.30	even.lisp	281
Задача 3.31	sum-odd.lisp	282
Задача 3.32	sum-even.lisp	282
Задача 3.33	count-zero.lisp	283
Задача 3.34	product-even.lisp	285
Задача 3.35	average-cadr.lisp	285
Задача 3.36	binary-arithmetic.lisp	286
Задача 3.37	binary+.lisp	287
Задача 3.38	count-atoms.lisp	288
Задача 3.39	sum-137-last.lisp	289
Задача 3.40	double-plusp.lisp	289
Задача 3.41	avarage-num.lisp	289
Задача 3.42	more-nums.lisp	290
Задача 3.43	list2-3.lisp	290
Задача 3.44	matrix-main.lisp	290
Задача 3.45	max-depth-0.lisp	292
Задача 3.46	evenp-sum.lisp	292
Задача 3.47	parens left .lisp	295
Задача 3.48	max-occur.lisp	297
Задача 3.49	caar-cdaaar.lisp	298
Задача 3.50	caddd.lisp	298
Задача 3.51	count-till*.lisp	298
Задача 3.52	winner.lisp	299
Задача 3.53	advantage.lisp	299
Задача 3.54	clear-average.lisp	300
Задача 3.55	lead-atom.lisp	300

Задача 3.56	product-even.lisp.....	301
Задача 3.57	our-length.lisp	301
Задача 3.58	sumx.lisp	302
Задача 3.59	mult.lisp	302
Задача 3.60	longest.lisp	303
Задача 3.61	prod-digits.lisp	303
Задача 3.62	sum-digits.lisp	304
Задача 3.63	magic-square.lisp	305
Задача 3.64	count-not-numbers.lisp.....	305
Задача 3.65	double-zero.lisp	306
Задача 3.66	max-min.lisp	307
Задача 3.67	expt-7-8.lisp	308
Задача 3.68	max-depth-1.lisp.....	308
Задача 3.69	average-wage.lisp.....	308
Задача 3.70	min-plusp.lisp.....	309
Задача 3.71	max-line.lisp.....	310
Задача 3.72	sum-eventh.lisp	310
Задача 3.73	compare-first-last.lisp.....	310
Задача 3.74	non-number.lisp	311
Задача 3.75	expp.lisp.....	311
Задача 3.76	min-number.lisp	312
Задача 3.77	sum-elms.lisp	312
Задача 3.78	deep-product.lisp.....	313
Задача 3.79	sum-10+.lisp.....	315
Задача 3.80	sum-twins.lisp	315
Задача 3.81	count-atoms.lisp	316
Задача 3.82	glue-numbers.lisp	317
Задача 3.83	prns.lisp.....	318
Задача 3.84	sum-ns.lisp	319
Задача 3.85	escalade.lisp	320
Задача 3.86	sum-int-10+.lisp	321
Задача 3.87	list-inside.lisp.....	321
Задача 3.88	$1+z+z^2/2+z^3/3$.lisp.....	322

Задача 3.89	over-n.lisp	322
Задача 3.90	8th.lisp.....	323
Задача 3.91	count-aas.lisp	324
Задача 3.92	lastt.lisp.....	325
Задача 3.93	unique.lisp.....	325
Задача 3.94	count-min.lisp	328
Задача 3.95	count-odd.lisp.....	328
Задача 3.96	(lambda (w) (if...)).lisp	329
Задача 3.97	sum-oddp.lisp.....	330
Задача 3.98	monotonic.lisp.....	331
Задача 3.99	count-minus.lisp.....	332
Задача 3.100	check-<.lisp.....	333
Задача 3.101	atom-n.lisp	333
Задача 3.102	count-all.lisp.....	334
Задача 3.103	anarchy-three.lisp	336
Задача 3.104	palindrome-number.lisp	336
Задача 3.105	verify-first-second-identity.lisp	338
Задача 3.106	zerop-car.lisp.....	338
Задача 3.107	deep-sum.lisp	338
Задача 3.108	chosen-sum.lisp.....	339
Задача 3.109	smallest-multiple.lisp	340
Задача 3.110	min-even.lisp.....	340
Задача 3.111	every-numberp.lisp	342
Задача 3.112	most-expensive.lisp.....	344
Задача 3.113	cheapest.lisp	344
Задача 3.114	stock-amount.lisp	345
Задача 3.115	stock-cost.lisp.....	346
Задача 3.116	sum-first-last.lisp.....	347
Задача 3.117	largest-number.lisp.....	347
Задача 3.118	maximum-gap.lisp.....	350
Задача 3.119	count-symbol.lisp.....	351
Задача 3.120	dominance.lisp	352
Задача 3.121	last-elm.lisp.....	352

Задача 3.122	sum-numbers.lisp	352
Задача 3.123	sum-evenp.lisp	353
Задача 3.124	reduce-gcd.lisp	353
Задача 3.125	min-number.lisp	354
Задача 3.126	square-matrix-main-diagonal-below-sum.lisp	355
Задача 3.127	square-matrix-main-diagonal-above-sum.lisp	355
Задача 3.128	product.lisp	356
Задача 3.129	min-char.lisp	357
Задача 3.130	some-duhuplicates.lisp	358
Задача 3.131	average-w.lisp	359
Задача 3.132	sum-even.lisp	360
Задача 3.133	count-n.lisp	361
Задача 3.134	double-empty.lisp	362
Задача 3.135	flatp.lisp	362
Задача 3.136	sum.lisp	362
Задача 3.137	count-parities.lisp	363
Задача 3.138	max-zeros.lisp	365
Задача 3.139	max-zero-seq.lisp	365
Задача 3.140	count-zero.lisp	367
Задача 3.141	last-element.lisp	367
Задача 3.142	count-exclusive.lisp	368
Задача 3.143	sum-max-even-min-odd.lisp	368
Задача 3.144	sum-max-even-min-odd-order.lisp	369
Задача 3.145	consistent-content.lisp	369
Задача 3.146	product-flat.lisp	371
Задача 3.147	product-flat-sine-zero.lisp	371
Задача 3.148	product-number-of-positive.lisp	372
Задача 3.149	average-of-positive.lisp	373
Задача 3.150	numbers.lisp	374
Задача 3.151	sum-min-max.lisp	374
Задача 3.152	alternating-sum.lisp	375
Задача 3.153	max-product.lisp	375
Задача 3.154	first-most-boring-element.lisp	377

Задача 3.155	deepest.lisp.....	377
Задача 3.156	len.lisp.....	378
Задача 3.157	self-dual.lisp.....	378
Задача 3.158	and -four.lisp.....	378
Задача 3.159	evennn.lisp.....	379
Задача 3.160	decreasing.lisp.....	381
Задача 3.161	knock-into-one.lisp.....	381
Задача 3.162	digits-number.lisp.....	384
Задача 3.163	twins.lisp.....	385
Задача 3.164	sum-equals.lisp.....	385
Задача 3.165	decreasing.lisp.....	386
Задача 3.166	eqq.lisp.....	387
Задача 3.167	duplicatep.lisp.....	388
Задача 3.168	deep-sum.lisp.....	388
Задача 3.169	nearp.lisp.....	389
Задача 3.170	list-insidep.lisp.....	390
Задача 3.171	string-sine-length.lisp.....	391
Задача 3.172	max-cnt.lisp.....	391
Задача 3.173	product-deep.lisp.....	393
Задача 3.174	sorted.lisp.....	394
Задача 3.175	count-triples.lisp.....	395
Задача 3.176	our-if.lisp.....	396
Задача 3.177	sum-numbers.lisp.....	397
Задача 3.178	cnt-smb.lisp.....	398
Задача 3.179	cnt-char.lisp.....	399
Задача 3.180	count-string-with-zero.lisp.....	401
Задача 3.181	very-big-sum.lisp.....	403
Задача 3.182	simple-array-sum.lisp.....	404
Задача 3.183	staircase.lisp.....	405
Задача 3.184	deep-twins.lisp.....	406
Задача 3.185	join-with-delimiter.lisp.....	407
Задача 3.186	ascendingp.lisp.....	410
Задача 3.187	length4p.lisp.....	411

Задача 3.188 length4-atomp.lisp.....	412
Структура 4 (функция (СПИСОК)) > (СПИСОК).....	413
Задача 4.1 our-last.lisp.....	413
Задача 4.2 odd-even.lisp.....	413
Задача 4.3 age.lisp	414
Задача 4.4 ages.lisp.....	414
Задача 4.5 w021.lisp.....	415
Задача 4.6 list-min.lisp	415
Задача 4.7 del-third.lisp	416
Задача 4.8 order-elms.lisp.....	419
Задача 4.9 del-last3.lisp	419
Задача 4.10 reverse-els.lisp.....	420
Задача 4.11 maxub.lisp.....	421
Задача 4.12 group3.lisp.....	422
Задача 4.13 list-company.lisp	424
Задача 4.14 twice-atom.lisp.....	425
Задача 4.15 vowels-del.lisp	425
Задача 4.16 lets&nums.lisp.....	426
Задача 4.17 nums-to-words.lisp.....	427
Задача 4.18 sec-blast-num.lisp.....	428
Задача 4.19 car-encl-list.lisp.....	428
Задача 4.20 arithmetic-to-text.lisp	429
Задача 4.21 parse-pascal.lisp	430
Задача 4.22 arabic-to-roman.lisp	431
Задача 4.23 reverse-list.lisp	431
Задача 4.24 liat.lisp.....	432
Задача 4.25 reverse-subs.lisp	433
Задача 4.26 num-list-atom.lisp	434
Задача 4.27 del-multy.lisp	434
Задача 4.28 even-2-3.lisp.....	435
Задача 4.29 atbash.lisp.....	435
Задача 4.30 sine-twins.lisp	436
Задача 4.31 conc-asterisc.lisp	437

Задача 4.32	wordy.lisp	438
Задача 4.33	sum-matrixx.lisp	438
Задача 4.34	area.lisp	439
Задача 4.35	clear-twins.lisp	440
Задача 4.36	fold-n-m.lisp	440
Задача 4.37	swing.lisp	440
Задача 4.38	map-sum.lisp	441
Задача 4.39	best-grades.lisp	443
Задача 4.40	unique-subs.lisp	443
Задача 4.41	permutate.lisp	444
Задача 4.42	sine-penultimate.lisp	445
Задача 4.43	check-prime-numbers.lisp	445
Задача 4.44	same-celsius.lisp	446
Задача 4.45	sum-lists.lisp	446
Задача 4.46	minus>0.lisp	447
Задача 4.47	unique-atoms.lisp	447
Задача 4.48	intercom-section-marks.lisp	449
Задача 4.49	append-reversed.lisp	449
Задача 4.50	cadr.lisp	451
Задача 4.51	rotate-last-two.lisp	451
Задача 4.52	atom-level.lisp	452
Задача 4.53	ascending-row.lisp	453
Задача 4.54	cadr-car.lisp	453
Задача 4.55	transpose-matrix.lisp	453
Задача 4.56	delete-twins.lisp	454
Задача 4.57	allocate-cards.lisp	455
Задача 4.58	average-sine.lisp	457
Задача 4.59	not-zero.lisp	458
Задача 4.60	1..4>zero.lisp	459
Задача 4.61	odd/even.lisp	460
Задача 4.62	remove-twins.lisp	461
Задача 4.63	pair-elms.lisp	462
Задача 4.64	reverse-mxx.lisp	463

Задача 4.65	annex.lisp	465
Задача 4.66	remove-lists.lisp	467
Задача 4.67	read&change.lisp.....	467
Задача 4.68	chess-board.lisp.....	468
Задача 4.69	high-landers.lisp.....	469
Задача 4.70	matrix-column-average.lisp.....	469
Задача 4.71	minus>a.lisp	469
Задача 4.72	xelms.lisp	470
Задача 4.73	matrix.lisp	471
Задача 4.74	x-doubles.lisp	472
Задача 4.75	zeros.lisp	473
Задача 4.76	grow-deep.lisp.....	474
Задача 4.77	up-words.lisp.....	476
Задача 4.78	drop-dups.lisp.....	476
Задача 4.79	exponent.lisp	478
Задача 4.80	cap-words.lisp	478
Задача 4.81	sum-pair.lisp.....	479
Задача 4.82	drop-pluspp.lisp.....	481
Задача 4.83	sum-pairs.lisp	481
Задача 4.84	remove-eventh.lisp	482
Задача 4.85	elm-freq.lisp	482
Задача 4.86	arrange-num-uscore-char.lisp.....	484
Задача 4.87	double.lisp.....	484
Задача 4.88	rearrange.lisp.....	485
Задача 4.89	drop-plusp.lisp.....	485
Задача 4.90	rotate.lisp.....	486
Задача 4.91	wvd.lisp.....	487
Задача 4.92	cut-1.lisp.....	487
Задача 4.93	plusp10.lisp	488
Задача 4.94	list-t-nil.lisp.....	488
Задача 4.95	even-plusp.lisp	489
Задача 4.96	compress-word.lisp	489
Задача 4.97	arrange.lisp.....	492

Задача 4.98	arrange-vcnm.lisp.....	493
Задача 4.99	decompose.lisp.....	495
Задача 4.100	translate.lisp	497
Задача 4.101	delete-first-atom.lisp	499
Задача 4.102	coin.lisp.....	500
Задача 4.103	intersection-nums.lisp	501
Задача 4.104	remove-names.lisp	501
Задача 4.105	rooms&price.lisp.....	502
Задача 4.106	edges-up.lisp	503
Задача 4.107	sort-numbers.lisp.....	504
Задача 4.108	compress-words.lisp.....	505
Задача 4.109	odd>0-even>1.lisp.....	506
Задача 4.110	del-blast.lisp.....	507
Задача 4.111	sum-digits.lisp.....	507
Задача 4.112	sort-words.lisp.....	508
Задача 4.113	list-caars.lisp	509
Задача 4.114	remove-odds.lisp.....	509
Задача 4.115	smbs-only.lisp	510
Задача 4.116	list-products.lisp.....	510
Задача 4.117	select-sort.lisp	511
Задача 4.118	selection-sort.lisp	511
Задача 4.119	sign-to-word.lisp	512
Задача 4.120	del-dkrange.lisp.....	512
Задача 4.121	rev&del-pairs.lisp.....	513
Задача 4.122	even-odd.lisp.....	513
Задача 4.123	minus-plus.lisp.....	513
Задача 4.124	xlambda.lisp	514
Задача 4.125	append-reverse.lisp	514
Задача 4.126	grow-numbers.lisp.....	515
Задача 4.127	quicksort.lisp.....	515
Задача 4.128	swap-elms.lisp.....	517
Задача 4.129	sin>cos-sqrt>abs.lisp.....	517
Задача 4.130	popular-names.lisp	517

Задача 4.131	list-data.lisp	519
Задача 4.132	read-numbers.lisp	519
Задача 4.133	cons-n.lisp	519
Задача 4.134	drop-deep-duplicates.lisp	520
Задача 4.135	meet.lisp	521
Задача 4.136	show-ironian-names.lisp	521
Задача 4.137	massp.lisp	524
Задача 4.138	word-right-p.lisp	525
Задача 4.139	ordered-chaos.lisp	525
Задача 4.140	un-even.lisp	526
Задача 4.141	minus-odd.lisp	526
Задача 4.142	snow-flake.lisp	527
Задача 4.143	max>min-substitute.lisp	528
Задача 4.144	insert-asterisc.lisp	528
Задача 4.145	pair-unpaired.lisp	529
Задача 4.146	map-repeat.lisp	529
Задача 4.147	surnames-common-ages.lisp	530
Задача 4.148	xine-twins.lisp	531
Задача 4.149	count-atoms.lisp	532
Задача 4.150	list-cadr.lisp	532
Задача 4.151	delete-on-list.lisp	533
Задача 4.152	insection.lisp	534
Задача 4.153	show-names.lisp	536
Задача 4.154	product-tail.lisp	537
Задача 4.155	min-trio.lisp	538
Задача 4.156	delete-on.lisp	538
Задача 4.157	list>a.lisp	539
Задача 4.158	neighbors.lisp	539
Задача 4.159	sbst.lisp	540
Задача 4.160	za-sentence.lisp	540
Задача 4.161	pos/neg.lisp	540
Задача 4.162	ini-lets.lisp	541
Задача 4.163	check.lisp	541

Задача 4.164	num-letters.lisp.....	541
Задача 4.165	polish-notation.lisp.....	543
Задача 4.166	odd-even.lisp.....	543
Задача 4.167	reverse-annex.lisp	544
Задача 4.168	unique-symb.lisp.....	544
Задача 4.169	sum-shift.lisp.....	544
Задача 4.170	del-list.lisp.....	545
Задача 4.171	add.lisp.....	545
Задача 4.172	increase.lisp.....	546
Задача 4.173	let-number.lisp	546
Задача 4.174	list-unlist.lisp.....	547
Задача 4.175	last-first.lisp	547
Задача 4.176	min-max.lisp	548
Задача 4.177	letter-likeness.lisp	549
Задача 4.178	matrix-rotate-max-min.lisp	549
Задача 4.179	del-second.lisp.....	550
Задача 4.180	sublist-cdddr.lisp.....	550
Задача 4.181	pos-max-min.lisp.....	551
Задача 4.182	name-years.lisp	552
Задача 4.183	flat-zero.lisp	554
Задача 4.184	n-atoms-n-lists.lisp.....	554
Задача 4.185	multiply.lisp	556
Задача 4.186	pair-chainge.lisp.....	556
Задача 4.187	del-over-average.lisp.....	557
Задача 4.188	eventh-max-minusp.lisp	558
Задача 4.189	_reverse.lisp	558
Задача 4.190	drop-eventh.lisp.....	560
Задача 4.191	delm-thirds.lisp	561
Задача 4.192	drop-sym-with-digits.lisp.....	561
Задача 4.193	drop-core.lisp	562
Задача 4.194	sum-underscore-code.lisp.....	563
Задача 4.195	_maplist.lisp.....	565
Задача 4.196	sum-list-atom.lisp.....	566

Задача 4.197	drop-elm.lisp	566
Задача 4.198	flat-unique.lisp	567
Задача 4.199	compress.lisp	568
Задача 4.200	del-deviant.lisp	570
Задача 4.201	maxeq.lisp	571
Задача 4.202	max-chain.lisp	573
Задача 4.203	our-mapcon.lisp	574
Задача 4.204	drop-last.lisp	574
Задача 4.205	deep-reverse.lisp	575
Задача 4.206	not-numbers-and-sum.lisp	576
Задача 4.207	elm-share.lisp	576
Задача 4.208	smash-mx.lisp	577
Задача 4.209	primes.lisp	579
Задача 4.210	all-chains.lisp	579
Задача 4.211	reverse-letters.lisp	580
Задача 4.212	reverse-words.lisp	581
Задача 4.213	reverse-input.lisp	582
Задача 4.214	eventh-min.lisp	583
Задача 4.215	second-half.lisp	583
Задача 4.216	compress&ratio.lisp	584
Задача 4.217	plus-num-sum.lisp	585
Задача 4.218	div3.lisp	586
Задача 4.219	div.lisp	587
Задача 4.220	column-plus-sum-num.lisp	587
Задача 4.221	nx-insert.lisp	588
Задача 4.222	max-min-mx.lisp	589
Задача 4.223	sets.lisp	590
Задача 4.224	integers.lisp	590
Задача 4.225	rotate.lisp	591
Задача 4.226	nrotate.lisp	593
Задача 4.227	drop-center.lisp	594
Задача 4.228	main-dmax.lisp	595
Задача 4.229	simplify-n.lisp	595

Задача 4.230	simplify-a.lisp	596
Задача 4.231	cohere.lisp	596
Задача 4.232	opposite-word.lisp	598
Задача 4.233	prefix.lisp	598
Задача 4.234	rotate-core.lisp	600
Задача 4.235	filter.lisp	601
Задача 4.236	inner-reverse.lisp	601
Задача 4.237	grades.lisp	602
Задача 4.238	minusp-to-zero.lisp	602
Задача 4.239	len&sum.lisp	603
Задача 4.240	co-insiders.lisp	604
Задача 4.241	neg.lisp	605
Задача 4.242	overs-n.lisp	605
Задача 4.243	backward.lisp	606
Задача 4.244	evens.lisp	606
Задача 4.245	del-thirds.lisp	607
Задача 4.246	make-assoc.lisp	608
Задача 4.247	num-eventh.lisp	609
Задача 4.248	sum-pairs.lisp	609
Задача 4.249	sub-lengths.lisp	610
Задача 4.250	make-pairs.lisp	611
Задача 4.251	abstract-sum.lisp	611
Задача 4.252	insert-nums.lisp	612
Задача 4.253	lets-marks.lisp	613
Задача 4.254	1-2-2-1.lisp	614
Задача 4.255	if-intp-abs-min-max-list-average.lisp	615
Задача 4.256	class.lisp	616
Задача 4.257	rev-ev-lev.lisp	616
Задача 4.258	list-elm-num.lisp	617
Задача 4.259	lessen.lisp	618
Задача 4.260	drop-second.lisp	618
Задача 4.261	flat-pair.lisp	619
Задача 4.262	minus-plus.lisp	620

Задача 4.263	odd*2-even*3.lisp	621
Задача 4.264	odddth/3.lisp	622
Задача 4.265	twin-elm.lisp	622
Задача 4.266	plus-exp.lisp	623
Задача 4.267	snap.lisp	624
Задача 4.268	plus-minus.lisp	625
Задача 4.269	all-zero.lisp	626
Задача 4.270	let>1.lisp	626
Задача 4.271	dig-over.lisp	627
Задача 4.272	sieve-of-eratosthenes.lisp	629
Задача 4.273	elm-occur.lisp	630
Задача 4.274	prefix.lisp	631
Задача 4.275	atom-level.lisp	631
Задача 4.276	matrix-max-min.lisp	632
Задача 4.277	max-ab.lisp	634
Задача 4.278	max-min-elms.lisp	634
Задача 4.279	atom-sublist.lisp	635
Задача 4.280	missed-nums.lisp	635
Задача 4.281	-list.lisp	636
Задача 4.282	matrix-sorted.lisp	637
Задача 4.283	str-frequent.lisp	638
Задача 4.284	sublists.lisp	639
Задача 4.285	max-sum-sublists.lisp	639
Задача 4.286	max-sum-all-sublists.lisp	640
Задача 4.287	max-t.lisp	640
Задача 4.288	min-t.lisp	641
Задача 4.289	listt.lisp	641
Задача 4.290	atom-counter.lisp	643
Задача 4.291	flatten-and-count.lisp	643
Задача 4.292	get-plusp.lisp	644
Задача 4.293	sort-atoms-by-freq.lisp	645
Задача 4.294	symbols-numbers.lisp	645
Задача 4.295	level-number-of-atoms.lisp	647

Задача 4.296	deploy-atoms.lisp	647
Задача 4.297	sum-len.lisp	648
Задача 4.298	drop-num.lisp	649
Задача 4.299	reconquista.lisp	650
Задача 4.300	separate.lisp	650
Задача 4.301	skim.lisp	651
Задача 4.302	deep-sort.lisp	652
Задача 4.303	sum-elt.lisp	654
Задача 4.304	even/n-odd.lisp	655
Задача 4.305	sum-list.lisp	655
Задача 4.306	>postfix.lisp	656
Задача 4.307	sine-adjacent.lisp	656
Задача 4.308	sort-flatten.lisp	658
Задача 4.309	most-frequent.lisp	658
Задача 4.310	cleave.lisp	659
Задача 4.311	compress.lisp	659
Задача 4.312	deep-atoms.lisp	661
Задача 4.313	add-matrices.lisp	661
Задача 4.314	in-depth.lisp	662
Задача 4.315	peak-numbers.lisp	663
Задача 4.316	maximum-product-sequence.lisp	664
Задача 4.317	annex-<.lisp	665
Задача 4.318	dump-top-level-duplicates.lisp	665
Задача 4.319	listt.lisp	666
Задача 4.320	max-number.lisp	667
Задача 4.321	dive.lisp	668
Задача 4.322	rearrange.lisp	668
Задача 4.323	random-product.lisp	669
Задача 4.324	infixlisp	669
Задача 4.325	plusp>min.lisp	669
Задача 4.326	plusp>min-deep.lisp	670
Задача 4.327	abs-average.lisp	671
Задача 4.328	reverse-top.lisp	673

Задача 4.329	row-sums.lisp	674
Задача 4.330	clear-duplicates.lisp	675
Задача 4.331	bubble-sort.lisp	677
Задача 4.332	drop-core.lisp	677
Задача 4.333	front-end-pairs.lisp	678
Задача 4.334	frequent-column.lisp	679
Задача 4.335	jam.lisp	679
Задача 4.336	onion.lisp	679
Задача 4.337	aver-sine.lisp	680
Задача 4.338	count-more.lisp	681
Задача 4.339	half.lisp	682
Задача 4.340	max-length-seq.lisp	683
Задача 4.341	make-odd.lisp	689
Задача 4.342	more-than-one.lisp	690
Задача 4.343	square.lisp	691
Задача 4.344	elms-positions.lisp	693
Задача 4.345	thirds.lisp	694
Задача 4.346	double.lisp	695
Задача 4.347	double-third.lisp	698
Задача 4.348	double-third-us.lisp	698
Задача 4.349	sqrt-them-all.lisp	699
Задача 4.350	deep-reverse.lisp	700
Задача 4.351	change.lisp	701
Задача 4.352	sum-numbers.lisp	702
Задача 4.353	intermittent.lisp	702
Задача 4.354	percentage.lisp	703
Задача 4.355	main-sum.lisp	703
Задача 4.356	result.lisp	704
Задача 4.357	max-span.lisp	705
Задача 4.358	conditional-mid-edges-change.lisp	706
Задача 4.359	pos+.lisp	707
Задача 4.360	matrix-main.lisp	708
Задача 4.361	quine.lisp	709

Задача 4.362	sum-square.lisp	709
Задача 4.363	first-min.lisp	710
Задача 4.364	first-min-all.lisp	711
Задача 4.365	ns.lisp	711
Задача 4.366	plural.lisp	712
Задача 4.367	inside-out.lisp	712
Задача 4.368	max-frequent.lisp	713
Задача 4.369	max-product-3.lisp	714
Задача 4.370	_remove-duplicates.lisp	716
Задача 4.371	max-days-off-month.lisp	718
Задача 4.372	edge-backward-pair.lisp	719
Задача 4.373	shift-pair.lisp	719
Задача 4.374	edge-sum.lisp	721
Задача 4.375	twone.lisp	722
Задача 4.376	negative-zero.lisp	723
Задача 4.377	drop-middle.lisp	724
Задача 4.378	ppos.lisp	724
Задача 4.379	group-same.lisp	725
Задача 4.380	eval-select.lisp	725
Задача 4.381	sum.lisp	726
Задача 4.382	level.lisp	727
Задача 4.383	list-v.lisp	729
Задача 4.384	next.lisp	729
Задача 4.385	=sum.lisp	731
Задача 4.386	rep.lisp	732
Задача 4.387	code-names.lisp	733
Задача 4.388	spec-fluency-names.lisp	734
Задача 4.389	code-surnames.lisp	735
Задача 4.390	ma.lisp	736
Задача 4.391	an.lisp	737
Задача 4.392	plane-.lisp	738
Задача 4.393	mx-string.lisp	739
Задача 4.394	clear-square.lisp	742

Задача 4.395	rev.lisp.....	743
Задача 4.396	m2p3.lisp.....	744
Задача 4.397	pairs.lisp.....	744
Задача 4.398	plus-minus.lisp.....	745
Задача 4.399	glue-edges.lisp.....	746
Задача 4.400	arrange.lisp.....	746
Задача 4.401	nums.lisp.....	747
Задача 4.402	grow.lisp.....	747
Задача 4.403	level-number-of-lists.lisp.....	749
Задача 4.404	ssort.lisp.....	751
Задача 4.405	sum-m-p.lisp.....	751
Задача 4.406	pluralize.lisp.....	754
Задача 4.407	arrange.lisp.....	756
Задача 4.408	char-codes.lisp.....	757
Задача 4.409	rems.lisp.....	757
Задача 4.410	odd-sum.lisp.....	758
Задача 4.411	ending.lisp.....	760
Задача 4.412	num.lisp.....	761
Задача 4.413	odd-min.lisp.....	763
Задача 4.414	numbers.lisp.....	763
Задача 4.415	<average<.lisp.....	763
Задача 4.416	average-percent.lisp.....	764
Задача 4.417	minus-plus.lisp.....	764
Задача 4.418	break-into-odds-evens.lisp.....	766
Задача 4.419	student-average-percent.lisp.....	767
Задача 4.420	escort-plusp-zero-p-with-asterisk.lisp.....	767
Задача 4.421	shelf.lisp.....	769
Задача 4.422	capi.lisp.....	770
Задача 4.423	cnt-elms.lisp.....	771
Задача 4.424	<<<<.lisp.....	773
Задача 4.425	syms.lisp.....	773
Задача 4.426	deep-reverse.lisp.....	773
Задача 4.427	abstract-min.lisp.....	774

Задача 4.428	revv.lisp.....	775
Задача 4.429	glue-sort-count-deep.lisp.....	776
Задача 4.430	plus-minus-zero.lisp.....	778
Задача 4.431	xenna.lisp.....	779
Задача 4.432	pair.lisp.....	780
Задача 4.433	pack.lisp.....	781
Задача 4.434	abba.lisp.....	781
Задача 4.435	maxx.lisp.....	782
Задача 4.436	maxxx.lisp.....	783
Задача 4.437	maxxxx.lisp.....	783
Задача 4.438	sum-columns.lisp.....	784
Задача 4.439	maxx.lisp.....	787
Задача 4.440	maxxx.lisp.....	788
Задача 4.441	level-sum.lisp.....	790
Задача 4.442	drop-mid.lisp.....	796
Задача 4.443	drop-empty.lisp.....	797
Задача 4.444	b.lisp.....	798
Задача 4.445	ssort.lisp.....	799
Задача 4.446	deep-double.lisp.....	799
Задача 4.447	sum-nrow.lisp.....	801
Задача 4.448	modify.lisp.....	801
Задача 4.449	drop-atoms.lisp.....	803
Задача 4.450	list-pair.lisp.....	804
Задача 4.451	adjacency-list.lisp.....	805
Задача 4.452	not-arithmetic-progression-permutations.lisp.....	805
Задача 4.453	frequency.lisp.....	806
Задача 4.454	blast.lisp.....	807
Задача 4.455	flatten.lisp.....	808
Структура 5 (функция ATOM ATOM) > ATOM.....		810
Задача 5.1	runner.lisp.....	810
Задача 5.2	average.lisp.....	810
Задача 5.3	string-right-n.lisp.....	810
Задача 5.4	add-range.lisp.....	811

Задача 5.5 disk-area.lisp	811
Задача 5.6 solid-torus.lisp.....	811
Задача 5.7 $a+a*[a+1]+a*[a+1]*[a+2]+...+a*[a+1]*...*[a+n].lisp$	812
Задача 5.8 insert-lets.lisp	813
Задача 5.9 calculate.lisp.....	814
Задача 5.10 greatest-common-divisor.lisp	814
Задача 5.11 replace-vowels.lisp	815
Задача 5.12 co-prime.lisp	815
Задача 5.13 sum-cubes.lisp.....	816
Задача 5.14 sum-expt-n-m.lisp	816
Задача 5.15 quotient.lisp.....	816
Задача 5.16 quotient-remainder.lisp	817
Задача 5.17 -expt.lisp.....	817
Задача 5.18 add.lisp	818
Задача 5.19 money-lost.lisp.....	819
Задача 5.20 brick.lisp.....	820
Задача 5.21 vowels>a.lisp.....	820
Задача 5.22 adjacent-digits-greatest-product.lisp	820
Задача 5.23 sum-primes.lisp	823
Задача 5.24 power-digit-sum.lisp	824
Задача 5.25 number-letter-counts.lisp.....	824
Задача 5.26 sum-cube.lisp	825
Задача 5.27 product.lisp.....	826
Задача 5.28 sum-input.lisp.....	827
Задача 5.29 newton.lisp	827
Задача 5.30 div-abs.lisp	828
Задача 5.31 digit.lisp	828
Задача 5.32 caesar-encipher.lisp.....	829
Задача 5.33 caesar-decipher.lisp.....	829
Задача 5.34 ionescu.lisp.....	830
Задача 5.35 concord-mask-string.lisp	831
Задача 5.36 alphabet-to-words.lisp	832
Задача 5.37 binomial.lisp.....	833

Задача 5.38	sum-cube.lisp	834
Задача 5.39	cnt.lisp.....	835
Задача 5.40	max-palindrome.lisp	835
Задача 5.41	product-primes.lisp	836
Задача 5.42	compare-triplets.lisp.....	839
Задача 5.43	birthday-cake-candles.lisp.....	840
Задача 5.44	product-row.lisp	840
Задача 5.45	count-hv.lisp.....	841
Задача 5.46	common-words.lisp.....	842
Задача 5.47	number-of-psychiatric-hospital-patients.lisp	843
Задача 5.48	pw.lisp.....	844
Структура 6 (функция АТОМ АТОМ) > (СПИСОК)		845
Задача 6.1	same-words.lisp	845
Задача 6.2	quotient&remainder.lisp.....	845
Задача 6.3	random-row.lisp	846
Задача 6.4	onion.lisp.....	846
Задача 6.5	primes.lisp.....	847
Задача 6.6	numbers.lisp.....	849
Задача 6.7	count-dictionary.lisp	849
Задача 6.8	max-divided.lisp.....	851
Задача 6.9	generate.lisp	853
Задача 6.10	onion-right.lisp.....	854
Задача 6.11	rand-add-pair.lisp	854
Задача 6.12	rand-even.lisp.....	854
Задача 6.13	list-a-n.lisp	856
Задача 6.14	add.lisp.....	856
Задача 6.15	positions.lisp	858
Задача 6.16	largest-palindrome-product.lisp	859
Задача 6.17	word-set-difference.lisp	862
Задача 6.18	eeq.lisp	863
Задача 6.19	unique-digits.lisp.....	868
Задача 6.20	range.lisp.....	869
Задача 6.21	reversed-primes.lisp	871

Задача 6.22 betwixt.lisp.....	871
Задача 6.23 set-exclusive-random.lisp.....	872
Задача 6.24 intersection-random.lisp.....	873
Задача 6.25 make-function-number-plus.lisp.....	874
Задача 6.26 primes.lisp.....	875
Задача 6.27 compose.lisp.....	876
Задача 6.28 symmetric.lisp.....	877
Задача 6.29 fizzbuzz.lisp.....	877
Задача 6.30 lace.lisp.....	878
Задача 6.31 grow-circles.lisp.....	879
Структура 7 (функция (СПИСОК) АТОМ) > АТОМ.....	880
Задача 7.1 elms-num.lisp.....	880
Задача 7.2 check-number.lisp.....	881
Задача 7.3 insider.lisp.....	882
Задача 7.4 near-last.lisp.....	882
Задача 7.5 value.lisp.....	883
Задача 7.6 count-goods.lisp.....	884
Задача 7.7 last-n.lisp.....	884
Задача 7.8 last-a.lisp.....	886
Задача 7.9 count-same.lisp.....	888
Задача 7.10 count-elm.lisp.....	888
Задача 7.11 add&length.lisp.....	889
Задача 7.12 insider.lisp.....	889
Задача 7.13 count-atoms.lisp.....	890
Задача 7.14 elm.lisp.....	890
Задача 7.15 tail-minus.lisp.....	891
Задача 7.16 member-test.lisp.....	891
Задача 7.17 pos.lisp.....	892
Задача 7.18 cnt.lisp.....	893
Задача 7.19 sum-ns.lisp.....	893
Задача 7.20 check-console-primes.lisp.....	893
Задача 7.21 check-file-primes.lisp.....	894
Задача 7.22 exceed-number.lisp.....	895

Задача 7.23	dive-count.lisp.....	895
Задача 7.24	any.lisp.....	896
Задача 7.25	count-after.lisp.....	897
Задача 7.26	rhymes.lisp.....	898
Задача 7.27	_some.lisp.....	901
Задача 7.28	n-elem-n.lisp.....	901
Задача 7.29	previous-z.lisp.....	901
Задача 7.30	max-depth-of-atom.lisp.....	903
Задача 7.31	elm-num.lisp.....	903
Задача 7.32	low.lisp.....	904
Задача 7.33	n-prime.lisp.....	905
Задача 7.34	large-sum.lisp.....	906
Задача 7.35	_every.lisp.....	910
Задача 7.36	dive-product.lisp.....	911
Задача 7.37	bin-decim.lisp.....	912
Задача 7.38	count-exept.lisp.....	913
Задача 7.39	count-bigger-than-this.lisp.....	915
Задача 7.40	counter.lisp.....	916
Задача 7.41	dive-count-list.lisp.....	917
Задача 7.42	_some.lisp.....	917
Задача 7.43	count-number.lisp.....	918
Задача 7.44	pull.lisp.....	919
Задача 7.45	summ.lisp.....	920
Задача 7.46	elt-from-end.lisp.....	921
Задача 7.47	our-position.lisp.....	922
Задача 7.48	our-nth.lisp.....	923
Задача 7.49	rems-n.lisp.....	923
Задача 7.50	cntt.lisp.....	924
Задача 7.51	deep-nth.lisp.....	925
Задача 7.52	count<=.lisp.....	925
Задача 7.53	cnt-object.lisp.....	928
Задача 7.54	inside.lisp.....	929
Задача 7.55	w-pull.lisp.....	930

Задача 7.56 diagonal-difference.lisp.....	931
Задача 7.57 twice.lisp	933
Задача 7.58 cnt.lisp.....	934
Задача 7.59 cnt-sine.lisp	935
Структура 8 (функция (СПИСОК) АТОМ) > (СПИСОК)	937
Задача 8.1 _mapcar.lisp	937
Задача 8.2 multiply.lisp	938
Задача 8.3 drop-elm.lisp	939
Задача 8.4 add-elt.lisp.....	939
Задача 8.5 group2.lisp	940
Задача 8.6 blast-els.lisp.....	940
Задача 8.7 delete-first.lisp.....	941
Задача 8.8 frequent.lisp.....	942
Задача 8.9 mapping.lisp.....	944
Задача 8.10 group-n.lisp	945
Задача 8.11 quadros.lisp	946
Задача 8.12 _remove-if-not.lisp.....	946
Задача 8.13 dive-elms.lisp	947
Задача 8.14 get-while.lisp.....	948
Задача 8.15 dump-level.lisp.....	949
Задача 8.16 addn.lisp.....	950
Задача 8.17 add.lisp	951
Задача 8.18 plunge-in.lisp.....	952
Задача 8.19 substitute-first-or-last.lisp	954
Задача 8.20 mask-list.lisp	954
Задача 8.21 repeat-elts.lisp	955
Задача 8.22 del-last-ns.lisp	955
Задача 8.23 shift-left.lisp	955
Задача 8.24 remove-i+n.lisp	956
Задача 8.25 insert-@.lisp	956
Задача 8.26 hide****.lisp	956
Задача 8.27 del-over-n.lisp	957
Задача 8.28 delete-elt.lisp.....	957

Задача 8.29	shift-right.lisp.....	957
Задача 8.30	rotate>.lisp	958
Задача 8.31	cut-n-elms.lisp.....	959
Задача 8.32	ante-del.lisp.....	959
Задача 8.33	list-a.lisp.....	960
Задача 8.34	downsize.lisp.....	960
Задача 8.35	drop-num.lisp	960
Задача 8.36	dive-atoms.lisp	961
Задача 8.37	member-down.lisp.....	963
Задача 8.38	expt-elms.lisp	963
Задача 8.39	swing-elms.lisp	964
Задача 8.40	knapsack.lisp.....	964
Задача 8.41	insert-atom.lisp.....	968
Задача 8.42	purgatory.lisp	969
Задача 8.43	not-dictionary.lisp	969
Задача 8.44	zeka.lisp	969
Задача 8.45	cska.lisp.....	970
Задача 8.46	inserta.lisp	971
Задача 8.47	dump-levels.lisp	971
Задача 8.48	keys+.lisp	972
Задача 8.49	drop-n.lisp	972
Задача 8.50	with-pairs.lisp.....	972
Задача 8.51	drop-zero-rem-a-n.lisp	973
Задача 8.52	del-upto-a.lisp	974
Задача 8.53	n-elm.lisp	975
Задача 8.54	_every.lisp.....	975
Задача 8.55	sum-n-elms.lisp	976
Задача 8.56	cut-n-elms.lisp.....	976
Задача 8.57	drop<n.lisp	977
Задача 8.58	streak.lisp	977
Задача 8.59	add-abs.lisp	978
Задача 8.60	slow-moving-n-numbers-sum.lisp.....	981
Задача 8.61	anarchy-nums.lisp	981

Задача 8.62	dropp.lisp	982
Задача 8.63	complex-dropp.lisp.....	982
Задача 8.64	overlap-a.lisp.....	983
Задача 8.65	aftershock.lisp	983
Задача 8.66	grow.lisp.....	984
Задача 8.67	filter.lisp.....	984
Задача 8.68	del-n.lisp.....	986
Задача 8.69	symb-to-depth.lisp.....	986
Задача 8.70	rem-num-sq-rest.lisp	989
Задача 8.71	shelf-life>.lisp	990
Задача 8.72	children-rhymes.lisp.....	992
Задача 8.73	bigger-than-this.lisp	993
Задача 8.74	drop.lisp	994
Задача 8.75	under.lisp.....	995
Задача 8.76	mn.lisp.....	996
Задача 8.77	deep-drop.lisp.....	997
Задача 8.78	split.lisp.....	998
Задача 8.79	p-pos.lisp	998
Задача 8.80	unveil.lisp.....	999
Задача 8.81	drop-less.lisp	999
Задача 8.82	drop.lisp	1000
Задача 8.83	shell-sort.lisp	1001
Задача 8.84	merge-sort.lisp.....	1002
Задача 8.85	n-list.lisp.....	1002
Задача 8.86	mp.lisp.....	1004
Задача 8.87	add.lisp	1004
Задача 8.88	deep-abate.lisp.....	1005
Задача 8.89	butlast-elm-from-file.lisp	1006
Задача 8.90	del-every-k.lisp	1007
Задача 8.91	exact-sum-sequence.lisp.....	1007
Задача 8.92	sub<=n.lisp.....	1008
Задача 8.93	drop.lisp	1009
Задача 8.94	compliment.lisp.....	1010

Задача 8.95	neci.lisp.....	1011
Задача 8.96	plus-minus.lisp.....	1011
Структура 9 (функция (СПИСОК) (СПИСОК)) > АТОМ		1013
Задача 9.1	count-xpr.lisp.....	1013
Задача 9.2	equalsetp.lisp.....	1013
Задача 9.3	every>.lisp.....	1014
Задача 9.4	same-elm&pos.lisp.....	1015
Задача 9.5	over-atom.lisp.....	1015
Задача 9.6	same-order.lisp.....	1016
Задача 9.7	sublistp.lisp.....	1016
Задача 9.8	more-elements.lisp.....	1017
Задача 9.9	more-atoms.lisp.....	1017
Задача 9.10	subset-next.lisp.....	1018
Задача 9.11	maxum.lisp.....	1019
Задача 9.12	miss.lisp.....	1019
Задача 9.13	reversed.lisp.....	1020
Задача 9.14	max-sum-pair.lisp.....	1020
Задача 9.15	birthday-earlier.lisp.....	1020
Задача 9.16	euclidian-distance.lisp.....	1021
Задача 9.17	longer-than-p.lisp.....	1021
Задача 9.18	sum-expt.lisp.....	1022
Задача 9.19	first-and-second.lisp.....	1023
Задача 9.20	sum-product.lisp.....	1024
Задача 9.21	sorted.lisp.....	1025
Задача 9.22	eq-exp.lisp.....	1027
Структура 10 (функция (СПИСОК) (СПИСОК)) > (СПИСОК).....		1028
Задача 10.1	priority-sort.lisp.....	1028
Задача 10.2	uni.lisp.....	1028
Задача 10.3	mix-elements.lisp.....	1029
Задача 10.4	alter-mix.lisp.....	1030
Задача 10.5	cartesian-product.lisp.....	1031
Задача 10.6	exxp.lisp.....	1032
Задача 10.7	num-v-carvcdrw.lisp.....	1033

Задача 10.8	compare.lisp	1033
Задача 10.9	mix-www.lisp.....	1034
Задача 10.10	mult-pairs.lisp	1035
Задача 10.11	our-set-difference.lisp	1035
Задача 10.12	string-set-difference.lisp.....	1036
Задача 10.13	-union.lisp	1037
Задача 10.14	apply-list.lisp.....	1040
Задача 10.15	inter-evenp.lisp.....	1040
Задача 10.16	sort-exclusive.lisp.....	1041
Задача 10.17	unite-unique.lisp.....	1041
Задача 10.18	cartesian-sort.lisp	1041
Задача 10.19	our-set-exclusive-or.lisp.....	1042
Задача 10.20	_intersection.lisp	1044
Задача 10.21	shuffle.lisp.....	1045
Задача 10.22	sum-elms.lisp	1045
Задача 10.23	drop-set.lisp.....	1046
Задача 10.24	owl.lisp.....	1047
Задача 10.25	remove-same-els.lisp.....	1048
Задача 10.26	add-pairs.lisp	1048
Задача 10.27	intersect.lisp	1049
Задача 10.28	product-elms.lisp.....	1049
Задача 10.29	insert-ordered.lisp.....	1050
Задача 10.30	exclusive-atoms.lisp	1050
Задача 10.31	exclusive-atomsxx.lisp	1051
Задача 10.32	max-pair.lisp	1053
Задача 10.33	_set-difference.lisp	1054
Задача 10.34	a-merge.lisp.....	1054
Задача 10.35	zip.lisp.....	1055
Задача 10.36	inter-mass.lisp	1055
Задача 10.37	mix-zip.lisp	1056
Задача 10.38	max-atoms.lisp	1057
Задача 10.39	max-sum.lisp	1057
Задача 10.40	extract-nths.lisp.....	1057

Задача 10.41	united.lisp.....	1058
Задача 10.42	sum-pairs.lisp	1059
Задача 10.43	construct.lisp	1060
Задача 10.44	max-inter-seq.lisp.....	1060
Задача 10.45	maxx-inter-seq.lisp.....	1062
Задача 10.46	deep-member.lisp	1062
Задача 10.47	cartesian-product-sorted.lisp	1063
Задача 10.48	bowl-sorted.lisp.....	1066
Задача 10.49	sum-/.lisp.....	1067
Задача 10.50	matrices-tensor-product.lisp	1068
Задача 10.51	ordered.lisp.....	1069
Задача 10.52	pair.lisp	1070
Задача 10.53	drop-suffixes.lisp.....	1071
Задача 10.54	prefix-infix.lisp	1071
Задача 10.55	drop-index.lisp	1072
Задача 10.56	subtract.lisp	1074
Задача 10.57	remove-elms.lisp	1075
Задача 10.58	cs.lisp	1075
Задача 10.59	srrt.lisp	1077
Задача 10.60	deep-set-difference.lisp	1078
Задача 10.61	deep-intersection.lisp	1079
Задача 10.62	our-nconc.lisp.....	1081
Задача 10.63	bind.lisp.....	1081
Задача 10.64	deep-bind.lisp.....	1082
Задача 10.65	compose.lisp.....	1084
Задача 10.66	set-distinction.lisp	1085
Задача 10.67	mx-product.lisp	1087
Задача 10.68	mxx-product.lisp	1088
Задача 10.69	msort.lisp.....	1089
Задача 10.70	summ.lisp.....	1089
Задача 10.71	m.lisp.....	1091
Задача 10.72	common-maximum-length-words.lisp	1091
Задача 10.73	common-max-length-fragments.lisp	1092

Структура 11 (функция ATOM ATOM ATOM) > ATOM	1094
Задача 11.1 sum-fibonacci.lisp	1094
Задача 11.2 common-elmsp.lisp	1094
Задача 11.3 in-range.lisp.....	1095
Задача 11.4 n-fibonacci.lisp.....	1095
Задача 11.5 replace-lets.lisp	1096
Задача 11.6 a-delete-b.lisp.....	1097
Задача 11.7 sum-n-fibonacci.lisp.....	1099
Задача 11.8 sum-fibonacci<=n.lisp.....	1099
Задача 11.9 balloon-eq.lisp.....	1100
Задача 11.10 arithmetic-n.lisp	1101
Задача 11.11 check-expt.lisp	1101
Задача 11.12 check-random.lisp	1101
Задача 11.13 triangle.lisp.....	1101
Задача 11.14 multiples-product.lisp.....	1102
Задача 11.15 sum-multiples.lisp	1102
Задача 11.16 random-number-sower.lisp.....	1103
Задача 11.17 trianglep.lisp.....	1104
Задача 11.18 product-xmultiples.lisp.....	1104
Задача 11.19 drop-xmultiples.lisp	1105
Задача 11.20 substitute-word.lisp	1105
Задача 11.21 marathon-ms.lisp	1106
Задача 11.22 sum-seven.lisp.....	1106
Задача 11.23 rectangular-cuboid-volume.lisp.....	1106
Задача 11.24 rectangular-cuboid-lateral-surface-area.lisp	1107
Задача 11.25 rectangular-cuboid-surface-area.lisp	1108
Задача 11.26 rectangular-cuboid-space-diagonal-length.lisp.....	1108
Задача 11.27 insert-word.lisp.....	1109
Задача 11.28 roofing.lisp	1110
Задача 11.29 tourist-fatigue.lisp	1111
Задача 11.30 replace-all.lisp	1113
Задача 11.31 sum-third-even.lisp.....	1113
Задача 11.32 gap-multip-345.lisp	1114

Задача 11.33 day-of-week-1753+.lisp	1115
Задача 11.34 day-of-week-1900+.lisp	1116
Структура 12 (функция АТОМ АТОМ АТОМ) > (СПИСОК).....	1117
Задача 12.1 our-list.lisp	1117
Задача 12.2 odd3+.lisp	1117
Задача 12.3 wagon.lisp	1118
Задача 12.4 square2nd-or-remove3rd.lisp	1118
Задача 12.5 trapezoid.lisp.....	1119
Задача 12.6 div-rem.lisp	1119
Задача 12.7 same-digits.lisp	1119
Задача 12.8 square-digits-times.lisp	1120
Задача 12.9 sum-digits=n.lisp.....	1121
Задача 12.10 rectungle.lisp.....	1122
Задача 12.11 random-matrix.lisp.....	1122
Задача 12.12 graftage.lisp.....	1123
Задача 12.13 process.lisp.....	1123
Задача 12.14 list&process.lisp.....	1124
Задача 12.15 min-max.lisp	1125
Задача 12.16 insulation.lisp	1126
Задача 12.17 range.lisp.....	1126
Структура 13 (функция (СПИСОК) АТОМ АТОМ) > АТОМ.....	1128
Задача 13.1 sum-sequence.lisp	1128
Задача 13.2 zeros-only.lisp.....	1129
Задача 13.3 az>a.lisp	1129
Задача 13.4 accord.lisp	1130
Задача 13.5 counter.lisp.....	1131
Задача 13.6 conditional-reverse.lisp.....	1131
Задача 13.7 check.lisp	1132
Задача 13.8 inside.lisp	1132
Структура 14 (функция (СПИСОК) АТОМ АТОМ) > (СПИСОК)	1133
Задача 14.1 sbst.lisp	1133
Задача 14.2 drop-range.lisp	1134
Задача 14.3 list-goods.lisp.....	1135

Задача 14.4	select-range.lisp.....	1135
Задача 14.5	frame-elms.lisp.....	1136
Задача 14.6	reduce-combined.lisp	1136
Задача 14.7	insert-lets.lisp	1137
Задача 14.8	replace-elts.lisp	1137
Задача 14.9	sort-cadr.lisp.....	1138
Задача 14.10	insert-on .lisp.....	1138
Задача 14.11	a-*2-change .lisp	1138
Задача 14.12	dive-n+m .lisp	1139
Задача 14.13	del-lev-n.lisp	1139
Задача 14.14	atoms-n-m.lisp.....	1141
Задача 14.15	remove-n-m.lisp	1144
Задача 14.16	meet-nums.lisp	1144
Задача 14.17	in-the-wake.lisp.....	1146
Задача 14.18	range-evenp.lisp	1147
Задача 14.19	substitute-all.lisp	1148
Задача 14.20	substitute-first.lisp.....	1149
Задача 14.21	stammer.lisp	1149
Задача 14.22	dive-next.lisp.....	1150
Задача 14.23	name.lisp	1151
Задача 14.24	change.lisp.....	1153
Задача 14.25	range.lisp.....	1153
Задача 14.26	drop.lisp	1154
Задача 14.27	swap.lisp.....	1155
Задача 14.28	blast-np-elm.lisp.....	1156
Задача 14.29	swap.lisp.....	1156
Задача 14.30	modify.lisp	1157
Задача 14.31	re.lisp.....	1157
Задача 14.32	srch.lisp	1158
Задача 14.33	alter.lisp.....	1159
Задача 14.34	insert.lisp.....	1160
Задача 14.35	sbst.lisp.....	1160
Задача 14.36	select-p.lisp	1162

Структура 15 (функция (СПИСОК) (СПИСОК) АТОМ) > АТОМ	1163
Задача 15.1 capital.lisp	1163
Структура 16 (функция (СПИСОК) (СПИСОК) АТОМ) > (СПИСОК)	1165
Задача 16.1 gen-arrangement.lisp	1165
Задача 16.2 insert-els.lisp	1167
Задача 16.3 insert-list.lisp	1168
Задача 16.4 bus-ticket.lisp	1168
Задача 16.5 run-function.lisp	1169
Задача 16.6 insert-elms.lisp	1170
Задача 16.7 insert.lisp	1170
Задача 16.8 insert-after.lisp	1171
Задача 16.9 dive-change.lisp	1172
Задача 16.10 set-diff.lisp	1173
Задача 16.11 insert-after-x.lisp	1176
Задача 16.12 inter>.lisp	1177
Структура 17 (функция (СПИСОК) (СПИСОК) (СПИСОК)) > (АТОМ)	1178
Задача 17.1 prefix-suffix.lisp	1178
Структура 18 (функция (СПИСОК) (СПИСОК) (СПИСОК)) > (СПИСОК)	1180
Задача 18.1 replace-elms.lisp	1180
Задача 18.2 remove-numbers.lisp	1181
Задача 18.3 unique-elms.lisp	1182
Задача 18.4 place-cons.lisp	1183
Задача 18.5 replace-a-b.lisp	1184
Задача 18.6 reveal.lisp	1185
Задача 18.7 replace-atom.lisp	1188
Задача 18.8 exclusive.lisp	1189
Задача 18.9 clear.lisp	1190
Задача 18.10 1-3-6.lisp	1192
Структура 19 (функция (СПИСОК) АТОМ АТОМ АТОМ) > АТОМ	1194
Задача 19.1 cocoa-bag-capacity.lisp	1194
Задача 19.2 left-rectangle.lisp	1194
Задача 19.3 right-rectangle.lisp	1195
Задача 19.4 midpoint-rectangle.lisp	1195

Задача 19.5 trapezium.lisp.....	1195
Задача 19.6 simpson.lisp.....	1196
Структура 20 (функция (СПИСОК) АТОМ АТОМ АТОМ) > (СПИСОК).....	1197
Задача 20.1 over-conditional-range.lisp.....	1197
Задача 20.2 sub-matrix-ranges.lisp.....	1198
Задача 20.3 sub-elms-ranges.lisp.....	1199
Задача 20.4 sub-matrix-coordinates.lisp.....	1199
Задача 20.5 sub-elms-coordinates.lisp.....	1199
Задача 20.6 ects-grades.lisp.....	1200
Структура 21 (функция (СПИСОК) (СПИСОК) АТОМ АТОМ) > (СПИСОК) ...	1201
Задача 21.1 balance.lisp.....	1201
Задача 21.2 run-over.lisp.....	1201
Задача 21.3 divided.lisp.....	1202
Задача 21.4 divided-infinity.lisp.....	1203
Задача 21.5 xinsert-elms.lisp.....	1203
Послесловие	1204
Литература	1205
Указатель функций.....	1211
Приложения	1234
Дж. Джеймс «Дао программирования»	1234
Л. Каганов «DOS»	1244

Предисловие

...ум наш утратил известную долю своей крепости, восприимчивости и гибкости: обязательная специализация развила в нем односторонность и парализовала его восприимчивость в других областях.

Ипполит Тэн «Наполеон Бонапарт»

...учителя должны готовить студентов к их будущему, а не к своему прошлому.

Ричард Хэмминг «Искусство науки»

Наблюдаемая в начале XXI века смена технологических подходов в научных исследованиях и образовании с неизбежностью приводит не только к количественным, но и качественным изменениям в структуре и содержании подготовки студентов, обучающихся по специальности «история».

Во-первых, универсальный характер современного университетского образования все в большей степени предполагает освоение студентами-историками компьютерных технологий для сбора, хранения, классификации, анализа и представления исторической информации, реконструкции исторических событий. Во-вторых, последнее десятилетие привело также к росту использования издательских возможностей компьютерных систем в оформлении результатов студенческих исследований и повсеместному переходу обучающихся к компьютерному набору текстов курсовых и выпускных квалификационных работ.

Условием успешности освоения историками современных компьютерных технологий являются знания, умения и навыки, приобретенные ими в ходе изучения логики, информатики, хронологии, латинского языка и современных иностранных языков. Вместе с тем, важной составляющей подготовки студентов-историков наряду со знанием языков, умением пользоваться текстовыми процессорами, электронными таблицами и базами данных, является понимание того, как составляются и выполняются компьютерные программы, а также овладение навыками разработки алгоритмов и решения

задач с применением компьютерных технологий. Умение программировать – вторая грамотность и одна из главных компетенций XXI века.

Программирование можно представить как беседу с компьютером: вы говорите команды, инструкции, запросы и ждете, что он скажет в ответ. Чтобы быть понятным компьютером, необходимо говорить с ним на его языке. В мире насчитывается 6912 живых и 8512 языков программирования. Мы познакомимся с одним из самых элегантных из них – Лиспом, который был создан выдающимся американским информатиком, основоположником функционального программирования и автором термина «искусственный интеллект», лауреатом премии Тьюринга профессором Стэнфордского университета Джоном Маккарти в 1958 году.

Критериями выбора языка для обучения решению задач были избраны гибкость и простота (элегантность). Существуют и более лаконичные – K, J, Haskell, но предлагаемый Common Lisp – стандарт ANSI INCITS 226-1994 (R2004), является языком "стройного замысла". Синтаксис его аскетичен и прост для изучения – для того, чтобы объяснить правила построения выражений, достаточно одного предложения. Все данные в Лиспе могут быть представлены в виде списка, заключенного в круглые скобки '(a b c d), если перед открывающей скобкой убрать апостроф, то список (a b c d) будет выполнен как функция a с параметрами b, c, d. Привычная математическая запись сложения $(2 + 3)$ представляется как $(+ 2 3)$.

Следует заметить, что быстрое знакомство с синтаксисом избавит вас от необходимости припоминать правила и очередность выполнения функций, но этого будет недостаточно для решения задач по программированию на Лиспе. Также, как «одно изучение правил передвижения шахматных фигур не сделает вас гроссмейстером» (Хэл Абельсон). Для того, чтобы писать программы, необходимо познакомиться со многими приемами и сотнями встроенных функций.

Существует простой и единственный быстрый способ научиться программированию – улучшать готовые решения и писать собственные программы. Данное пособие составлено с целью помочь вам приобрести вкус к решению задач и научиться программировать на Лиспе.

Установка программного обеспечения

Инструменты, которыми мы пользуемся, оказывают тонкое и глубокое влияние на наши способности мышления, а, следовательно, и на нашу способность мыслить.

Эдсгер Дейкстра

Выгоды от использования Лиспа заключаются в переживаниях и впечатлениях от использования Лиспа.

Питер Зибель «Практичный Common Lisp»

Чтобы стать кузнецом надо ковать.

Французская пословица

Следуй своему блаженству.

Джозеф Кэмпбелл «Власть мифа»

Искусство программирования заключается в формулировании пошаговых инструкций. Примером программы является последовательность шагов по установке программного обеспечения и решению следующей задачи – определить функцию, которая удваивает число, посланное ей в качестве аргумента. Далее представлено решение на языке программирования ANSI Common Lisp для операционной системы Windows 7¹:

Шаг 1. Скачать LispBox: http://www.common-lisp.net/project/lispbox/test_builds/lispbox-0.7-ccl-1.6-windowsx86.zip

Шаг 2. Разархивировать lispbox-0.7-ccl-1.6-windowsx86.zip и выполнить C:\Program Files (x86)\LispBox\lispbox.bat (в старых версиях C:\Program Files (x86)\LispBox\RunLispBox.bat).

* для OS X необходимо установить: http://common-lisp.net/project/lispbox/test_builds/Lispbox-0.7-with-ccl-1.6-darwinx86.zip;

для Linux 32-bit установить: http://common-lisp.net/project/lispbox/test_builds/lispbox-0.7-ccl-1.6-linuxx86.tar.gz, для Linux 64-bit: http://common-lisp.net/project/lispbox/test_builds/lispbox-0.7-ccl-1.6-linuxx86-64.tar.gz (Версия Linux требует GTK2 и некоторые другие общие библиотеки, и имеет проблемы с glibc на некоторых Linux дистрибутивах), а также скрипты для автоматизации загрузки, компиляции и настройки средств разработки для Лиспа: <https://github.com/LinkFly/lisp-dev-tools>

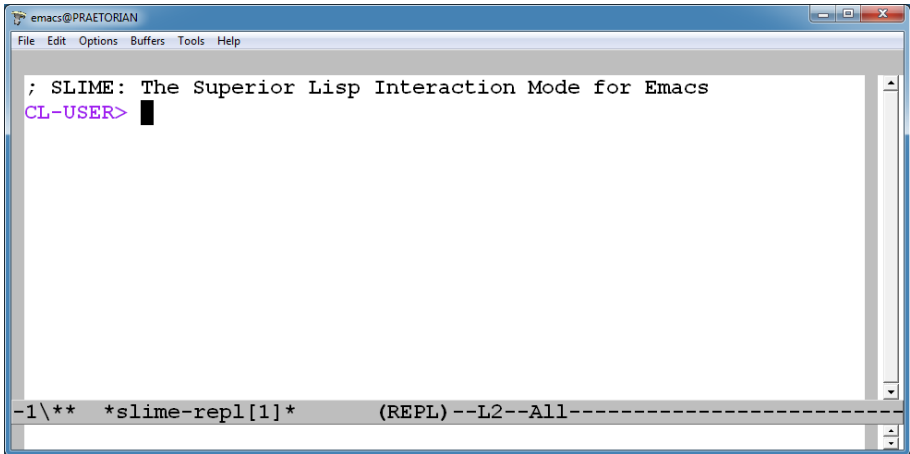


Рис. 1. Выполнение C:\Program Files (x86)\LispBox\lispbox.bat

или:

Шаг 1. Скачать Lisp Cabinet: <http://lispcabinet.sourceforge.net>

Шаг 2. Выполнить C:\Program Files (x86)\LispCabinet\LispCabinet.exe, затем нажать на клавиатуре клавишу ALT и, не отпуская ее, нажать клавишу X (в краткой записи: M-X), набрать slime и нажать клавишу ENTER. Далее:

Шаг 3. Перейти к вводу имени файла (поиску): CTRL-X CTRL-F, то есть нажать на клавиатуре клавишу CTRL и, не отпуская ее, нажать последовательно клавиши X и F:

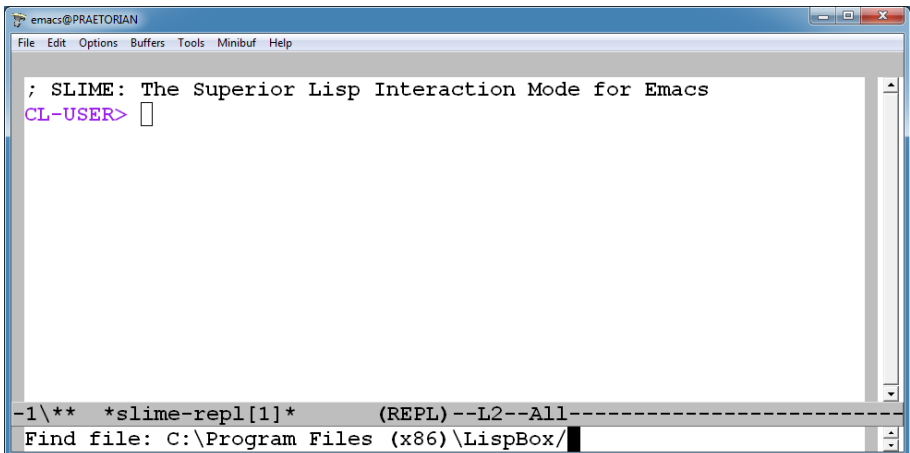


Рис. 2. Переход к вводу имени файла (поиску)

Шаг 4. Ввести имя файла, например `double.lisp` (программы в Лиспе хранятся в файлах с расширением `.lisp`) и нажать клавишу `ENTER`:

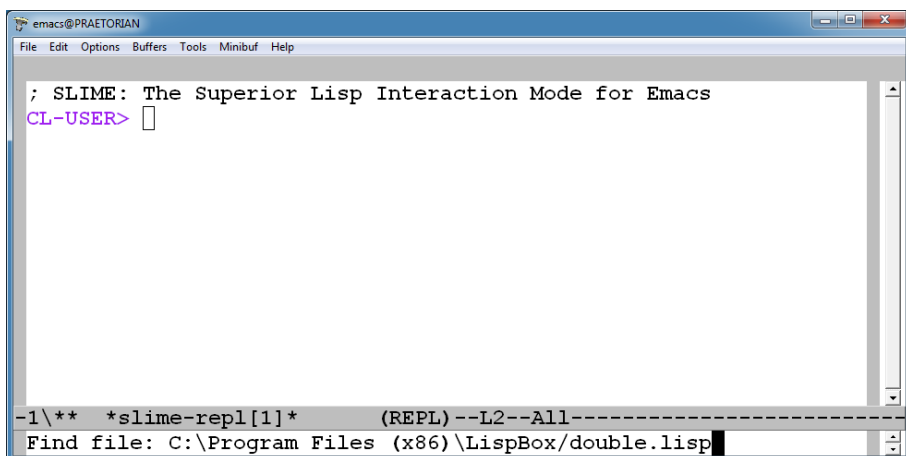


Рис. 3. Ввод имени файла

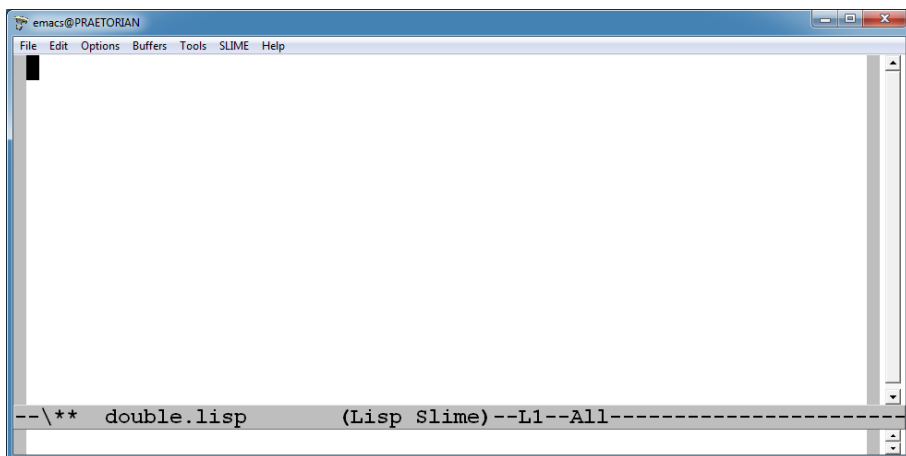


Рис. 4. Создание файла `C:\LispBox\double.lisp`

(если в папке `C:\LispBox\` нет файла с таким именем, то он будет создан `C:\LispBox\double.lisp`).

Шаг 5. Набрать программу, которая будет сохранена в файле double.lisp:

```
(defun double (x) (* x 2))
```

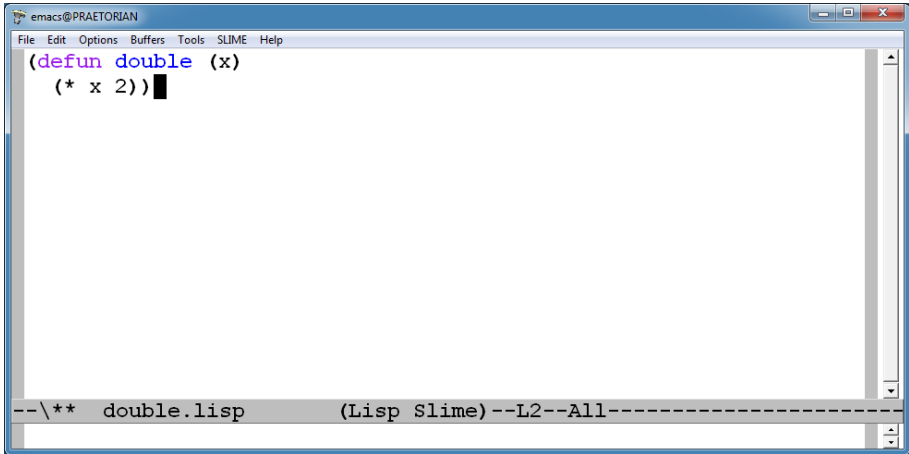


Рис. 5. Ввод текста программы

Шаг 6. Скомпилировать программу CTRL-C CTRL-K, то есть нажать на клавиатуре клавишу CTRL и, не отпуская ее, нажать последовательно клавиши C и K:

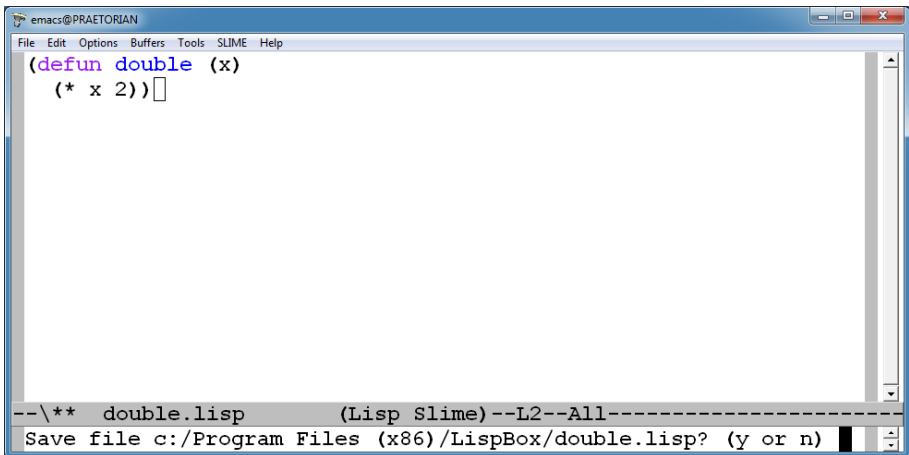


Рис. 6. Компилирование программы

и нажать клавишу Y:

The screenshot shows the Emacs editor window titled 'emacs@PRAETORIAN'. The main buffer contains the following Lisp code:

```
(defun double (x)
  (* x 2))
```

The status bar at the bottom indicates the file is 'double.lisp' and the mode is '(Lisp Slime)'. The output buffer shows the following messages:

```
--\-- double.lisp (Lisp Slime)--L2--All-----
;; Loading file C:\Program Files (x86)\LispBox\double.fas ...
;; Loaded file C:\Program Files (x86)\LispBox\double.fas
0 errors, 0 warnings

CL-USER>

-1\** *slime-repl[1]* (REPL)--L11--Bot-----
Compilation finished: 0 errors 0 warnings 0 notes [0.14 secs]
```

Рис. 7. Сохранение скомпилированной программы

или, если не поставлена закрывающая скобка:

The screenshot shows the Emacs editor window titled 'emacs@PRAETORIAN'. The main buffer contains the following Lisp code:

```
(defun double (x)
  (* x 2))
```

The status bar at the bottom indicates the file is 'double.lisp' and the mode is '(Lisp Slime)'. The output buffer shows the following error message:

```
--\-- double.lisp (Lisp Slime)--L2--All-----
READ: input stream #<INPUT BUFFERED FILE-STREAM CHARACTER
#P"C:\\Program Files (x86)\\LispBox\\double.lisp" @2> ends with
[Condition of type SYSTEM::SIMPLE-END-OF-FILE]

Restarts:
 0: [ABORT] Abort SLIME compilation.
 1: [ABORT] Abort handling SLIME request.

-1\%* *slldb [connection #1]* (slldb[1])--L6--Top-----
```

Рис. 8. Сообщение об ошибках

Шаг 7. Выйти из сообщения об ошибках, нажав на клавиатуре клавишу Q:

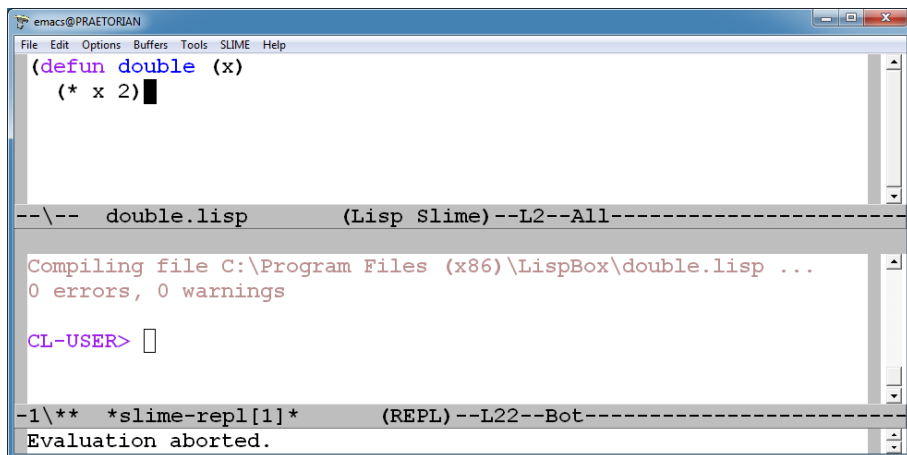


Рис. 9. Выход из сообщения об ошибках

Шаг 8. Исправить ошибку - поставить скобку и вернуться к пункту 6.

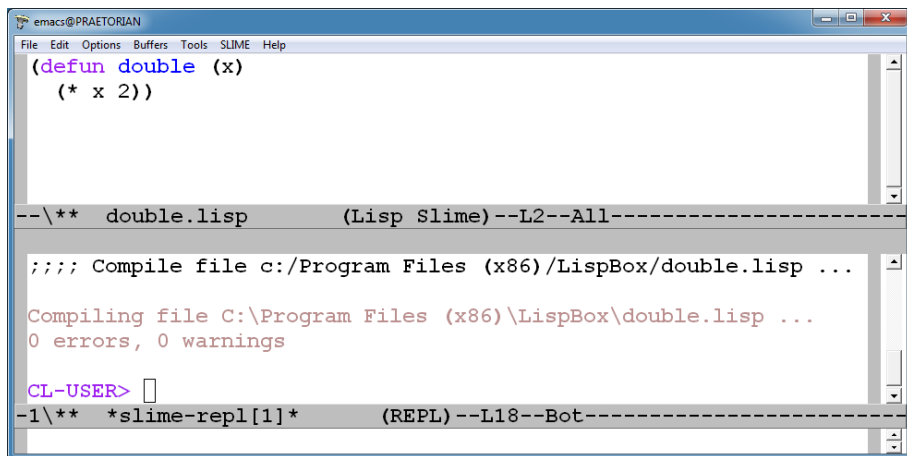


Рис. 10. Исправление ошибок

Шаг 9. Перейти в REPL (Read Evaluate Print Loop): CTRL-C CTRL-Z, то есть нажать на клавиатуре клавишу CTRL и, не отпуская ее, нажать последовательно клавиши C и Z:

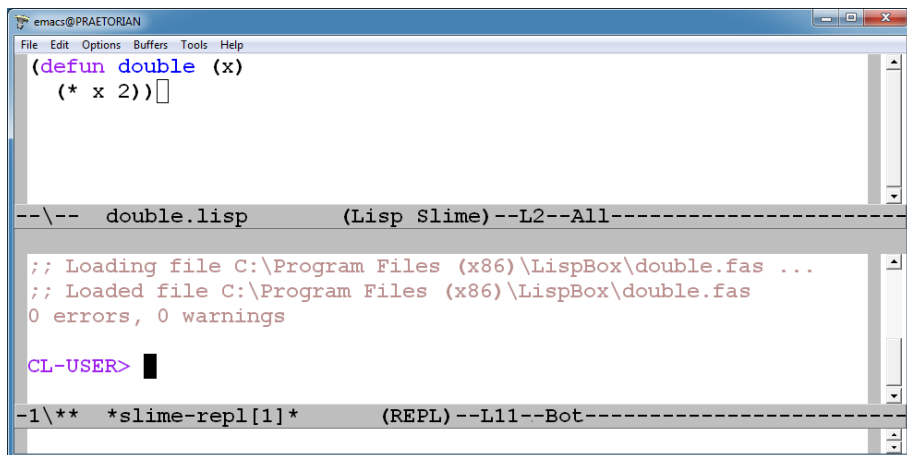


Рис. 11. Переход в REPL

Шаг 10. После приглашения CL-USER> ввести (double 100) и нажать клавишу ENTER:

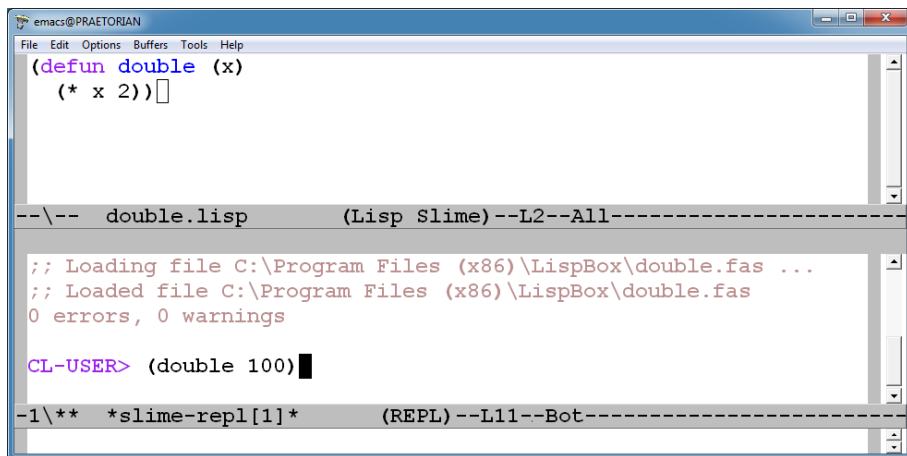


Рис. 12. Вызов функции

Программа выдаст ответ - 200:

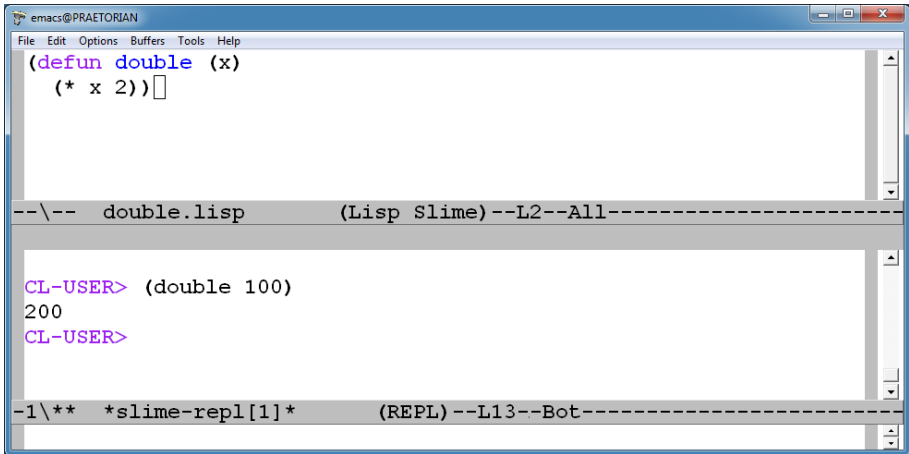


Рис. 13. Ответ

Дальнейшие шаги (переход по ссылкам):

1. Документация

<http://www.lispworks.com/documentation/lw50/CLHS/Front/Contents.htm>
http://cvberry.com/downloads/ansi_cl_standard_draft_nice.pdf
<http://filonenko-mikhail.github.io/cltl2-doc/rupdf/cltl2.pdf>
<http://najomi.org/common-lisp/>
<http://quickdocs.org>

2. Форумы

на русском:

<http://lisper.ru/forum/common-lisp>
<http://www.cyberforum.ru/lisp/>
<http://www.hardforum.ru/fl42/>
<http://lisp.ru/forums.php>

на английском:

<https://www.reddit.com/r/lisp/>
<https://stackoverflow.com/questions/tagged/common-lisp>
<https://groups.google.com/forum/#!forum/comp.lang.lisp>

Команды редактора Emacs SLIME

Vi is the god of editors. Emacs is the editor of gods.

~

Для работы с редактором Emacs в режиме SLIME (Superior Lisp Interaction Mode for Emacs) необходимо знать клавиатурные комбинации: открыть файл - c-x c-f, то есть, удерживая ctrl, нажать x, затем удерживая ctrl, нажать f, как сохранить файл - c-x c-s и как выйти из Emacs - c-x c-c (N.B. в краткой записи m - meta означает нажатие клавиши alt).

Файлы

Открыть файл	c-x c-f
Открыть файл в другом окне	c-x 4 c-f
Открыть файл только для чтения	c-x c-r
Сохранить	c-x c-s
Сохранить как	c-x c-w
Отменить	c-g
Выйти	c-x c-c

Буфер

Список буферов	c-x c-b
Следующий буфер	c-x o
Сменить буфер	c-x b
Заккрыть окно	c-x 0
Уничтожить буфер	c-x k
Оболочка (shell)	m-x
[]	c-x 1
[-]	c-x 2
[]	c-x 3

Курсор

Вперед на один значок	c-f или ->
Назад на один значок	c-b или <-
Вверх на один значок	c-p
Вниз на один значок	c-n
Вперед на одно слово	m-f
Назад на одно слово	m-b
Начало строки	c-a или home

Конец строки
 Предыдущее предложение
 Следующее предложение
 Начало параграфа
 Конец параграфа
 Страницу вверх
 Страницу вниз

c-e или end
 m-a
 m-e
 m-{
 m-}
 m-v
 c-v

Редактирование

Отменить последние изменения
 Копировать выделенный участок
 Вырезать выделенный участок
 Вставить выделенный участок
 Поменять местами значки
 Поменять местами слова
 Поменять местами s-выражения
 Поменять местами строки
 Удалить значок
 Удалить до конца слова
 Удалить до конца строки
 Удалить до начала слова
 Выделить

c-_ или c-/
 m-w
 c-w
 c-y
 c-t
 m-t
 c-m-t
 c-x c-t
 c-d
 m-d
 c-k
 c-bsp
 c-space или c-@

Поиск

Поиск вперед
 Поиск назад

c-s
 c-r

Замена

Заменить выборочно (с подтверждением)
 Заменить все вхождения

m-%
 m-x replace-string

Завершение

Завершить символ
 Показать варианты завершения символа
 Заполнить форму

m-/ или c-c c-i или c-m-i
 c-c m-i
 c-c c-s

Закрытие

Закрыть скобки

c-c c-q

Отступы

Создать отступ строки	tab
Создать отступ s-выражения	c-m-q

Компиляция

Компилировать функцию	c-c c-c
Компилировать и загрузить файл	c-c c-k
Переключиться на выводной буфер	c-c c-z

Отладчик

Выход	q

Навигация

Следующая команда в истории repl	m-n
Предыдущая команда в истории repl	m-p

Команды

Сменить директорию	,cd
Выход	quit

Макрос

c-x (	
(#\ c-f m-spc ' c-e ') c-n c-a	
c-x)	
то есть:	
Начать запись макроса	c-x (
Затем вставить символы	(#\
Перейти вперед на один символ	c-f
Удалить все пробелы кроме одного	m-spc
Вставить символ	`
Перейти в конец строки	c-e
Вставить символы	`)
Перейти в начало следующей строки	c-n c-a
Завершить запись макроса	c-x)
Выполнить макрос	c-x e
Повторить макрос 4 раза	c-x e e e e

Пакеты

Programming languages should be designed not by piling feature on top of feature, but by removing the weaknesses and restrictions that make additional features appear necessary.

Robert Findler, Jacob Matthews

Schemer: Buddha is small, clean, and serious. Lispnik: Buddha is big, has hairy armpits, and laughs.

Nikodemus Siivola

The limits of my language mean the limits of my world.

Ludwig Wittgenstein

Пакет - это лисповский объект, который связывает имена с символами и определяет область видимости символов. Текущий пакет хранится в глобальной переменной `*package*`. Сразу после старта текущим пакетом будет `common-lisp-user`.

Создать новый пакет:

```
CL-USER> (defpackage :bar (:use :common-lisp))
#<PACKAGE BAR>
```

Поместить файл в пакет:

```
CL-USER> (in-package :bar)
#<PACKAGE BAR>
BAR>
```

Переопределить функцию `+`:

```
BAR> (defun + (a b) (- a b))
+
BAR> (+ 3 2)
1
```

Переопределить функцию `cons`:

```
BAR> (defun cons (w) (reverse w))
; Evaluation aborted
DEFUN/DEFMACRO(CONS): #<PACKAGE COMMON-LISP> is locked
[Condition of type SYSTEM::SIMPLE-PACKAGE-ERROR]
```

Синтаксис

With experience we learn that simplicity is king and less is more, but it takes effort.

Gustavo Duarte

Learning lisp first is like having an incredibly good childhood. It will sabotage the rest of your life, which will always be harder and sadder.

~

...скобочки обнимают слова, и слова становятся добрее.

~

Лисп наделен однородным, последовательным и предсказуемым синтаксисом, который описывается всего двумя правилами построения выражений, которые легко запомнить:

программа = список = (атомы и подсписки)

Атомы и списки называются символьными выражениями или s-выражениями (s-expression, symbolic expression).

Список - перечень, последовательность, элементами которой являются атомы и/или подсписки. Список всегда начинается с открывающей круглой скобки и заканчивается закрывающей круглой скобкой. Элементы списка разделяются пробелами. Например, (a b c) или (a (b c)).

Таким образом, список - это многоуровневая или иерархическая структура данных, в которой открывающие и закрывающие скобки находятся в строгом соответствии.

Список, в котором нет ни одного элемента, называется пустым списком и обозначается () или символом nil. Он выполняет ту же роль, что и ноль в арифметике.

Список можно использовать для представления знаний:

(German Empire (Chancellor 1871 1890 (Otto Bismarck 1815 1898)))

Одним из основных отличий языка Лисп от традиционных языков программирования является запись в виде списков не только данных, но и программ. Например, список (+ 2 3) можно интерпретировать либо как список, состоящий из трех элементов, где quote, ' или ` блокируют вычисление выражения:

> '(+ 2 3)
(+ 2 3)

либо как действие (функцию + с аргументами 2 и 3), дающее в результате число 5:

```
> (+ 2 3)
5
```

Выборочно разблокировать s-выражение, заблокированное посредством backquote, можно с помощью запятой:

```
> `((+ 2 3) , (+ 2 3))
((+ 2 3) 5)
```

Общепринятая инфиксная нотация (запись):

```
> (2 + 5)
; Evaluation aborted
```

Синтаксис Лиспа основан на префиксной нотации - структура программы: (функция, аргумент, аргумент ...):

```
> (+ 2 5)
7
```

Префиксная нотация делает Лисп лаконичным языком, например:

```
> (+ 2 3 4 5 6 7); префиксная нотация
27
```

Правила интерпретации списков определяются первым элементом списка, которым является функция.

Синтаксис языка устанавливает простой порядок действий:

```
> (+ 2 (* 3 5))
17
```

Единообразный лаконичный синтаксис Лиспа в виде скобочной префиксной записи превращает программирование в удовольствие.

Существует и постфиксная нотация, как например, в языке программирования Forth:

```
> 2 5 +
; Evaluation aborted
```

Как вы смогли заметить - язык программирования Лисп синтаксически регулярен и концептуально элегантен (прост): все выражается функцией - даже математические операторы +, -, * и /.

ТИПЫ ДАННЫХ

Все языки программирования делятся на две большие группы, в одну из них входит Лисп, в другую – все остальные языки программирования.

Хювёнен Э., Сеппянен И. «Мир Лиспа»

Целью вычислений является понимание, а не числа.

Ричард Хэмминг

У вас есть единицы? Везучий ублюдок! Всё, что было у нас – это нули.

~

Чтобы быть уверенным, что Лисп поймет, о чем вы говорите, нужно ссылаться только на тот мир, который ему известен, и использовать понятные ему термины. В программировании на языке Лисп используются символы и построенные из них символьные структуры.

Символы

Символ – ограниченная пробелами последовательность знаков, другими словами – последовательность непробельных знаков (sempolon – греч. знак, условное обозначение, художественный образ, обозначающий какую-нибудь мысль, идею).

Символ – это имя переменной (функции), состоящее из букв без различия между прописными буквами и строчными, цифр и специальных значков !-@#\$\$%^&*_+/[{}]<>.? за исключением (),'`|":\; – которое обозначает какой-нибудь предмет, объект, вещь, действие из реального мира.

```
> 'history
HISTORY
```

Символы – это имена переменных, у которых есть свойства. Например, функция symbol-name возвращает имя символа:

```
> (symbol-name 'lisp1958)
"LISP1958"
```

В Лиспе, в отличие от большинства других языков программирования, допустимо начинать имя символа с цифры:

```
> '300spartans
300SPARTANS
```

```

> 'ansi-common-lisp-1994
ANSI-COMMON-LISP-1994

> '|santa came back|
|santa came back|

> 'january-25-2011-santa-came-back
JANUARY-25-2011-SANTA-CAME-BACK

> 'santa-came-back!-@#~$%^&*_+/[ ]{ }<>.?
SANTA-CAME-BACK!-@#~$%^&*_+/[ ]{ }<>.?

(defun !-@#~$%^&*_+/[ ]{ }<>.? (n) (+ n n))

> (!-@#~$%^&*_+/[ ]{ }<>.? 200)
400

```

Запрещено использование знаков (), ' ` | " : \ ;

```

> 'z(
; Evaluation aborted

> 'z)
; Evaluation aborted

> 'z,
; Evaluation aborted

> 'z'
; Evaluation aborted

> 'z`
; Evaluation aborted

> 'z|
; Evaluation aborted

> 'z:
; Evaluation aborted

> 'z"
; Evaluation aborted

> 'z\
|Z
|

> 'z; последний предваряет комментарии.
Z

```

Корректным именем функции, с точки зрения Лиспа, будет символ:
 $a + a[a+1] + a[a+1] * [a+2] + \dots + a[a+1] * \dots * [a+n]$ - см. Задача 5.7.

Перевод строки в символ

```
> (read-from-string "MODERN-AGE")
MODERN-AGE
10
```

Логические значения *t* и *nil*

Everything is or isn't pie.
~

Пустой горшочек, в нём можно держать что хочешь, это очень полезная вещь.

Алан Милн «Винни-Пух и все-все-все»

Символы *t* и *nil* имеют в Лиспе специальное всегда одно и то же фиксированное значение: *t* - истина - да - есть - *true*, а *nil* - ложь - нет - пусто - *false*. Символы *t* и *nil* также как и числа являются константами, их нельзя использовать в качестве имен других лисповских объектов.

```
> 750
T
```

```
> '()
NIL
```

Nil - единственный символ, который является атомом и списком одновременно - обозначает пустой список, подобно Кромвелю, для которого после принятия «Билля о самоотречении» было сделано исключение: он остался в армии, несмотря на то, что был депутатом Палаты общин.

Числа

Число - ограниченная пробелами последовательность знаков. Числа все же не являются полновесными символами, так как не могут представлять иные лисповские объекты кроме самих себя или своего числового значения, то есть числа являются константами.

```
> 5
5
```

```
> -100
-100
```

```
> 31/5
31/5
```

```
> (float 31/5)
6.2
```

Перевод строки в число

```
> (parse-integer "10")
10
```

Перевод знака в число

```
> (digit-char-p #\5)
5
```

Символы и числа представляют те простейшие объекты (атомы), из которых строятся остальные структуры.

Все символы, за исключением чисел и логических значений `t` и `nil`, если они не объявлены как константы при помощи директивы `defconstant`, исполняют роль переменных (`variables`), которые используются для обозначения других лисповских объектов.

Кроме этого система Common Lisp содержит глобальные специальные переменные, такие как символ `pi`, представляющий значение числа «пи» 3.141592653589793238510.

Буквы

Отдельная буква определяется как `#\m`. С помощью функции `char` можно как читать, так и писать знаки строки. `(char "Oliver Cromwell" n)` возвращает `n`-ый символ строки:

```
> (char "Oliver Cromwell" 10)
#\m
```

Каждой букве соответствует какое-либо простое число:

```
> (char-code #\m)
109
```

```
> (code-char 110)
#\n
```

Перевод строки в список букв

```
> (coerce "modern-age" 'list)
(#\m #\o #\d #\e #\r #\n #\ - #\a #\g #\e)
```

```
> (loop for c across "brett" collect c)
(#\b #\r #\e #\t #\t)

> (concatenate 'list "brett")
(#\b #\r #\e #\t #\t)

> (map 'list #'identity "brett")
(#\b #\r #\e #\t #\t)
```

Здесь решетка с последующей одинарной кавычкой `#'` является сокращенной записью специального оператора `function`, то есть `#'identity` эквивалентно `(function identity)` - см. Задача 3.188.

Список кодов букв строки

```
> (map 'list #'char-code "brett")
(98 114 101 116 116)
```

Строки

Строки - это векторы (направленные массивы), состоящие из буквенных значков. Для определения строки достаточно заключить набор букв в двойные кавычки:

```
> "modern age"
"modern age"
```

Сравнение строк

```
> (string-equal "modern age" "MODERN-AGE")
T

> (string= "modern age" "MODERN-AGE")
NIL
```

Перевод символа в строку

```
> (string 'modern-age)
"MODERN-AGE"
```

Перевод числа в строку

```
> (write-to-string 1792)
"1792"
```

Перевод списка букв в строку

```
> (coerce '(\a #\b #\c) 'string)
"abc"
```

Поиск подстроки в строке

```
> (search "b" "abc")
1
```

Присоединение буквы к строке

```
> (concatenate 'string (string #\a) "bc")
"abc"
```

Соединение строк

```
> (concatenate 'string "Bonny and " "Clyde")
"Bonny and Clyde"

> (format nil "~a and ~a" "Bonny" "Clyde")
"Bonny and Clyde"
```

Сортировка знаков строки

```
> (sort "Giuseppe Garibaldi" #'char<)
" G Gaabdeeiilpprsu"
```

Список слов строки

```
> (read-from-string (concatenate 'string "(" "modern age" ")"))
(MODERN AGE)
12
```

Список однобуквенных подстрок строки

```
> (mapcar 'string (coerce "brett" 'list))
("b" "r" "e" "t" "t")

> (map 'list #'string "brett")
("b" "r" "e" "t" "t")
```

```
> (loop for c across "brett" collect (string c))
("b" "r" "e" "t" "t")
```

Возврат части строки

```
> (subseq "modern age" 0 6)
"modern"
```

```
> (subseq "modern age" 7)
"age"
```

Последовательности

К последовательностям в Лиспе относятся списки и векторы (в том числе и строки).

Индексы векторов, строк и списков начинаются с нуля, так первый по счету элемент имеет индекс 0:

```
> (elt '(a b c d) 0)
A
```

```
> (elt '(a b c d) 1)
B
```

```
> (nth 1 '(a b c d))
B
```

```
> (nthcdr 1 '(a b c d))
(B C D)
```

```
> (position #\W "Wellington")
0
```

```
> (position #\n "Wellington")
5
```

```
> (position #\n "Wellington" :start 6)
9
```

```
> (position #\n "Wellington" :from-end t)
9
```

```
> (position-if #'oddp '((1) (2) (3) (4)) :start 1 :key #'car)
2
```

```
> (length '(a b c))
3
```

```

> (count 'a '(a a b))
2

> (count-if #'evenp '(1 2 3 4 5 6 7))
3

> (apply #'min '(7 4 6 4 5)) ;в CLISP до 4096 элементов, в SBCL до
4611686018427387903, иначе - reduce
4
> call-arguments-limit
4096

> (reduce #'max '(7 4 6 4 5))
7

> (search '(b) '(a b c))
1

> (find 'b '(a b c))
B

> (reverse '(c b a))
(A B C)

> (remove 'a '(a b a c))
(B C)

> (remove-if #'evenp '(1 2 3 4 5 6 7))
(1 3 5 7)

> (remove-if-not #'plusp '(-1 0 1 2 3))
(1 2 3)

> (remove-duplicates '(a a a b b c))
(A B C)

> (substitute 'z 'a '(a a a b b c))
(Z Z Z B B C)

```

В следующем разделе мы продолжим знакомиться с функциональным программированием на Лиспе. В отличие от процедурного подхода программирование при помощи функций соответствует следующим правилам:

- программа состоит из вызова функций, то есть функция принимает значение другой функции в качестве параметра;
- значения переменным не присваиваются, а передаются в качестве параметров.

ФУНКЦИИ

```
Lisp is lisp, it has no future and no past,  
it neither endorses nor combats competitors  
and it is entirely self sufficient.
```

~

```
My analyst warned me, but metaprogramming  
was so beautiful I got another analyst.
```

~

Лисп – это набор функций. Функцией в математике называется отображение (mapping), которое однозначно отображает одни значения на другие. Например, запись $y = f(x)$ ставит в соответствие каждому элементу x единственный элемент y из множества значений функции f . Это соответствие также можно записать в следующем виде $f(x) \rightarrow y$, где функция f от аргумента x имеет значение $y = f(x)$.

Функция в программировании – один из видов подпрограммы. Особенность, отличающая её от другого вида подпрограмм – процедуры, состоит в том, что функция возвращает значение, а её вызов может использоваться в программе как выражение.

У функции может быть произвольное количество аргументов (в том числе их может не быть совсем, и она будет иметь постоянное значение).

Примером является арифметическая функция сложения:

```
> (+ 2 3)  
5
```

Базовые функции

Мы познакомились с функцией сложения, однако, с самого начала Лисп рассматривался в качестве языка для создания искусственного интеллекта и был предназначен для работы с символами и списками.

Основные функции обработки списков

В Лиспе для построения, разбора и анализа списков существуют очень простые базовые функции. Простота базовых функций и их малое число – это характерные черты Лиспа. С ними связана математическая элегантность языка.

Функция создания списка – cons (конструирующая функция) включает новый элемент в начало списка:

```
> (cons 'a '())  
(a)
```

```
> (cons 'a '(b))
(A B)
```

Второй аргумент обязательно должен быть списком, иначе результатом будет точечная пара:

```
> (cons 'a 'b)
(A . B)
```

Функции разбора списка – `car` и `cdr` (селективные функции):

`car` – возвращает в качестве значения первый элемент списка (`head` – голову списка)

```
> (car '(a b c))
A
```

`cdr` – возвращает в качестве значения список, получаемый из исходного списка после удаления из него первого элемента (`tail` – хвост списка)

```
> (cdr '(a b c))
(B C)
```

Функции `car` и `cdr` являются сокращениями от наименований регистров вычислительной машины IBM 605 – `Contents of Address Register` и `Contents of Decrement Register`. Автор Лиспа Джон Маккарти реализовал первую Лисп-систему именно на этой машине и использовал регистры `car` и `cdr` для хранения головы и хвоста списка.

В Common Lisp наряду с именами `car` и `cdr` используют и более громоздкие имена `first` (первый) и `rest` (остаток).

```
> (first '(a b c))
A
> (rest '(a b c))
(B C)
```

Функции `car` и `cdr` определены лишь для аргументов, являющихся списками:

```
> (car 'a)
; Evaluation aborted
> (cdr 'a)
; Evaluation aborted
```

Функции проверки

Функции опознавания и анализа или базовые функции-предикаты, которые проверяют наличие некоторого свойства и возвращают `T` или `NIL`:

Базовая функция-предикат `atom` проверяет, является ли аргумент атомом:

```

> (atom 'a)
T
> (atom 234)
T
> (atom '(a b))
NIL
> (atom nil)
T
> (atom t)
T

```

Базовая функция-предикат `eq` проверяет тождественность двух символов:

```

> (eq 'Madrid 'Madrid)
T
> (eq 'Athens 'Madrid)
NIL
> (eq '() nil)
T
> (eq '(a b) '(a b))
NIL
> (eq 3.14 3.14)
NIL

```

Стандартные функции

Lisps are only good when they're huge!

Grue

Начнем знакомство со стандартными арифметическими функциями:

```

> (+ 1000 5)
1005
> (- 1000 5)
995
> (* 1000 5)
5000
> (/ 1000 5)
200

```

Функция возведения в степень - `expt`:

```

> (expt 2 3)
8

```

Некоторые встроенные функции (предикаты) проверяют выполнение некоторого условия и возвращают `T` или `NIL`.

Функция-предикат `listp` идентифицирует списки:

```

> (listp '(a b))

```

```
T
> (listp 'a)
NIL
```

Функция `null` проверяет на пустой список:

```
> (null '())
T
> (null '(a b))
NIL
```

Функция-предикат `=` сравнивает числа различных типов:

```
> (= 5 5.0)
T
> (= 5 10/2)
T
> (= 5 5.0 10/2)
T
```

Функция-предикат `eq1` сравнивает числа одинаковых типов:

```
> (eq1 3.14 3.14)
T
```

Функция-предикат `equal` проверяет одинаковость двух списков:

```
> (equal '(a b) '(a b))
T
> (equal 2 'cat)
NIL
```

Функция-предикат `equalp` сравнивает произвольные лисповские объекты. Этот предикат может потребоваться, когда нет уверенности в типе сравниваемых объектов или в корректности использования других предикатов сравнения:

```
> (equalp '(a b) '(a b))
T
```

Недостатком универсальных предикатов типа `equalp` является то, что их применение требует от системы несколько большего объема вычислений, чем использование специализированных предикатов.

```
> (numberp 1775)
T
> (numberp 'Orange)
NIL
```

```
> (symbolp 'Grant)
T
> (symbolp '1861)
```

NIL

> (zerop 0)

T

> (zerop 7)

NIL

> (evenp '1054)

T

> (evenp '1125)

NIL

> (oddp '1805)

T

> (oddp '1812)

NIL

> (atom 1852)

T

> (atom '(Washington Dublin London Canberra))

NIL

> (< 2 3)

T

> (< 3 2)

NIL

> (> 3 2)

T

> (> 2 3)

NIL

> (< 3 5 7)

T

> (> 8 6 4)

T

Вложение функций

> (/ 31 5)

31/5

> (float 31/5)

6.2

> (float (/ 31 5))

6.2

> (cons '+ '(2 3))

(+ 2 3)

> (* (cons '+ '(2 3)) 4)

; Evaluation aborted

> (* (eval (cons '+ '(2 3))) 4)

20

Например, найти наименьшее среди тех элементов первого списка, которые не входят во второй список:

```
> (apply #'min (set-difference '(3 5 7) '(3 4))
5
```

Стандартные функции (продолжение)

Обработку списков всегда можно свести к описанным ранее базовым функциям: `cons`, `car`, `cdr`, `atom` и `eq`, но программирование лишь с их использованием было бы очень сложным. Поэтому в Лисп-систему включено свыше 700 встроенных функций для различных действий и различных ситуаций.

Функция `list` создает список из элементов:

```
> (list 'a 'b 'c)
(A B C)
> (cons 'a (cons 'b (cons 'c nil)))
(A B C)
```

Вложенные вызовы `car` и `cdr` можно записывать в сокращенном виде:

```
> (car (cdr '(a b c)))
B
> (cadr '(a b c))
B
```

или

```
> (car (cdr (cdr (cdr '(a b c d)))))
D
> (caddr '(a b c d))
D
```

или

```
> (car (cdr (cdr (car '((a b c d) (e f g h))))))
C
> (caddr '((a b c d) (e f g h)))
C
```

Функция `member` проверяет, входит ли элемент в список:

```
> (member 'b '(a b c))
(B C)
```

Функция `nth` выделяет n -й элемент списка:

```
> (nth 2 '(Washington Dublin London Canberra))
LONDON
```

Функция `nthcdr` возвращает элементы списка, начиная с *n*-го элемента:

```
> (nthcdr 2 '(Washington Dublin London Canberra))
(LONDON CANBERRA)
```

Функция `append` соединяет списки:

```
> (append '(a b) '(c d) '(e))
(A B C D E)
```

Функция `revappend` переворачивает список и соединяет его со вторым:

```
> (revappend '(a b c) '(d e f))
(C B A D E F)
```

Функция `last` возвращает последний элемент списка:

```
> (last '(a b c d))
(D)
```

Функция `butlast` возвращает все элементы списка кроме последнего:

```
> (butlast '(a b c d))
(A B C)
```

Функция `length` возвращает длину списка:

```
> (length '(a b c d e f g h i j k l m n o p q r s t u v w x y z))
26
```

Функция `subseq` возвращает подпоследовательность:

```
> (subseq '(a b c d) 1 2)
(B)
> (subseq '(a b c d) 1)
(B C D)
```

Функция `reverse` переворачивает список:

```
> (reverse '(a b c))
(C B A)
```

Если элемент не входит в список, то функция `adjoin` ставит его на первое место:

```
> (adjoin 'c '(a b c))
(A B C)
> (adjoin 'h '(a b c))
(H A B C)
```

Функция `union` объединяет множества:

```
> (union '(a b c) '(c b d))
```

(A C B D)

Функция `intersection` возвращает пересечение множеств:

```
> (intersection '(a b c) '(b b c))
(B C)
```

Функция `set-difference` возвращает разность множеств:

```
> (set-difference '(a b c d e) '(b c))
(A D E)
```

Функция `set-exclusive-or` возвращает элементы списков, которые входят в один из списков, но не входят в оба списка:

```
> (set-exclusive-or '(a b c d e) '(b e))
(A C D)
```

Функция `mismatch` возвращает позицию первого элемента, начиная с которого списки отличаются:

```
> (mismatch '(a b c) '(a b d))
2
```

Функция `merge` соединяет и сортирует последовательности:

```
> (merge 'list '(1 2 3) '(1 2 5) #'<)
(1 1 2 2 3 5)

> (merge 'string "bank" "account" #'string<)
"abaccnkount"
```

Функция `every` проверяет условие для всех элементов списка:

```
> (every #'oddp '(1 3 5 7))
T

> (every #'> '(1 3 5) '(0 2 4))
T
```

Функция `some` проверяет условие для некоторых элементов списка:

```
> (some #'evenp '(1 2 5 7))
T
```

Функция `apply` применяет функцию к одному и более аргументу, последним из которых должен быть список:

```
> (apply #'* '(8 9 10))
720
```

```
> (apply #'* 8 9 '(10))
720
```


Функция `funcall` вызывает функцию с аргументами:

```
> (funcall #'* 8 9 10)
720
```

Среди функций, принимающих функции в качестве аргументов, часто применяются функции сортировки `sort` и чистки `remove-if`.

Первая является функцией сортировки общего назначения. Она принимает список аргументов и предикат, а возвращает список, отсортированный пропускаянием через предикат каждой пары элементов:

```
> (sort '(4 2 5 1 7 6 3) #'<)
(1 2 3 4 5 6 7)
```

Функция чистки списка `remove-if` принимает функцию и список, а возвращает все элементы списка, для которых функция вернула `false`:

```
> (remove-if #'evenp '(1 2 3 4 5 6 7))
(1 3 5 7)
```

Логические функции

```
> (and 5 (< 1 2))
T

> (or (< 2 1) (< 1 2))
T

> (not (< 2 1))
T
```

Ветвление вычислений

```
> (when (< 1 2) 'yes)
YES

> (if (< 1 2) 'yes 'no)
YES

(defun check (n)
  (cond ((zerop n) 'zero)
        ((plusp n) 'plus)
        ('minus)))

> (check 0)
ZERO
```

Создание (определение) функций

Lisp isn't a language, it's a building material.

Alan Kay

В Лиспе программы состоят из функций, которые вызывают друг друга. Определение функции состоит из имени функции, за которым в скобках следуют формальные параметры функции (формальность параметров означает, что их можно поменять на любые другие символы, и это не отразится на вычислениях, определяемых функцией), и тела функции.

Дать имя и определить новую функцию можно с помощью макроса `defun`: `(defun add-two (a b) (+ a b))` – определяет функцию с именем `add-two` и списком аргументов `(a b)`; `(+ a b)` – тело функции, которое состоит из последовательности выражений. Функция возвращает результат последнего выражения, в приведенном примере `(+ a b)`.

```
> (add-two 25 4000)
4025
```

```
(defun div-two (a b) (/ a b))
```

```
> (div-two 8000 5)
1600
```

Аналог встроенной функции `seventh`:

```
(defun -seventh (w) (car (cdr (cdr (cdr (cdr (cdr (cdr w))))))))
```

```
> (-seventh '(1 2 3 4 5 6 7 8 9 10))
7
```

Сумма элементов одноуровневого числового списка:

```
(defun sum-list (w)
  (if (null w) 0 (+ (car w) (sum-list (cdr w)))))
```

```
> (sum-list '(1 2 3 4))
10
```

Аналог встроенной функции `member`:

```
(defun -member (m w)
  (cond ((atom w) nil)
        ((eql (car w) m) w)
        ((-member m (cdr w)))))
```

```
> (-member 'c '(a b c d e))
(C D E)
```

Рекурсивные функции

À la guerre comme à la guerre.

французская пословица

Война это ... война!

Григорий Горин «Тот самый Мюнхгаузен»

Рекурсивной называется функция, которая для повторяющихся вычислений вызывает саму себя. Примерами рекурсии являются: матрешка, смерть Кошечки, Зазеркалье, временная петля, взаимность, два телената, девочка, прыгающая на скакалке, и собака, носящаяся за своим хвостом.

Функция вычисления положительной целой степени является рекурсивной:

```
(defun -expt (m n)
  (if (= n 1) m
      (* m (-expt m (1- n)))))

> (-expt 2 3)
8
```

Функция `-expt` вызывает себя до тех пор, пока ее аргумент `n` (счетчик) не уменьшится до 1 (то есть `n - 1` раз), после чего функция вернет значение `m`. При каждой передаче значения `m` на предыдущий уровень, результат умножается на `m`. Так `m` окажется перемноженным на себя `n` раз.

Другим примером рекурсии является функция нахождения факториала (произведения данного числа на все целые положительные числа, меньшие данного). Например, факториал 4 есть $1 * 2 * 3 * 4 = 24$.

```
(defun factorial (n)
  (if (zerop n) 1
      (* n (factorial (- n 1)))))

> (factorial 4)
24
```

Рекурсивные функции хорошо подходят для работы со списками, поскольку списки часто рекурсивно содержат подписки.

Простая рекурсия

Рекурсия является простой, если вызов функции встречается в некоторой ветви лишь один раз:

```
(defun x-length (w)
  (if w (1+ (x-length (cdr w)))
      0))
```

```

> (x-length '(a b c d))
4

(defun insert (a w)
  (cond ((null w) (cons a nil))
        ((< a (car w)) (cons a w))
        ((cons (car w) (insert a (cdr w))))))

> (insert 4 '(1 3 7))
(1 3 4 7)

```

Хвостовая рекурсия

Хвостовой рекурсией является функция, чей рекурсивный вызов самой себя является ее последней операцией:

```

(defun t-length (w &optional (ac 0))
  (if w (t-length (cdr w) (1+ ac))
      ac))

> (t-length '(a b c d))
4

```

&optional - указатель на то, что далее следуют необязательные (опциональные) параметры.

Параллельная рекурсия

Рекурсия называется параллельной, если она встречается одновременно в нескольких аргументах функции.

Следующая функция принимает два списка-аргумента *w* и *v*, возвращает *t*, если *v* является подсписком *w*.

Элементами списков могут быть атомы и списки любой вложенности:

```

(defun subp (v w)
  (cond ((null w) nil)
        ((equal v (car w)) t)
        ((atom (car w)) (subp v (cdr w)))
        (t (subp v (car w)) (subp v (cdr w)))))

> (subp '(a) '((a) b))
T

```

Здесь рекурсия применяется как к голове, так и к хвосту списка.

Взаимная рекурсия

Я оглянулся посмотреть, не оглянулась ли
она, чтоб посмотреть, не оглянулся ли я.

Максим Леонидов «Девочка-видение»

Волки от испуга скушали друг друга.

Корней Чуковский «Тараканище»

Рекурсия является взаимной между двумя или более функциями, если они вызывают друг друга. Таковыми являются функции `reverse-list` и `rearrange`, которые обращают элементы на всех уровнях списка:

```
(defun reverse-list (w) (if (atom w) w (rearrange w nil)))

(defun rearrange (w ac)
  (if w (rearrange (cdr w) (cons (reverse-list (car w)) ac)) ac))

> (reverse-list '((a b) c (d e) f))
(F (E D) C (B A))
```

Функция MAPCAR

Функция `mapcar` принимает в качестве аргументов функцию и один или более списков (один на каждый параметр функции) и применяет функцию последовательно к элементам каждого списка:

```
> (mapcar #'1- '(100 100 100))
(99 99 99)

> (mapcar #'list '(a b c) '(1 2 3))
((A 1) (B 2) (C 3))

> (mapcar #'list '(a b c d) '(1 2 3))
((A 1) (B 2) (C 3))

> (mapcar #'+ '(10 100 1000) '(1 2 3))
(11 102 1003)

> (mapcar #'car '((Ottawa CA) (Canberra AU) (Wellington NZ)))
(OTTAWA CANBERRA WELLINGTON)

> (mapcar #'cadr '((Ottawa CA) (Canberra AU) (Wellington NZ)))
(CA AU NZ)
```

Достаточно описать в виде функции, что нужно сделать с элементом списка и затем применить последовательно эту функцию ко всем элементам списка.

Макрос LAMBDA

- Как ловко ты перебил веревку. Отличный выстрел!
- Ну... Вообще-то, я метил в палача...

Мел Брукс «Робин Гуд: Мужчины в трико»

Мы уже видели, насколько удобно иметь возможность обращаться с функциями, как с данными. Во многих языках, даже если мы могли бы передать функцию в качестве аргумента чему-то, вроде `mapcar`, она бы, тем не менее, была функцией, определенной заранее в каком-нибудь исходном файле. Если бы требовался именно цельный код, добавляющий 10 к каждому элементу списка, мы бы определили функцию с именем `10+` или каким-то подобным только для этого единственного применения:

```
(defun 10+ (n) (+ n 10))

> (mapcar #'10+ '(1 2 3 4 5))
(11 12 13 14 15)
```

Для этой цели в Лиспе существует макрос `lambda`, при помощи которого вызывается безымянная (анонимная) временная функция, определяемая в месте использования:

```
> (mapcar #'(lambda (n) (+ n 10)) '(1 2 3 4 5))
(11 12 13 14 15)
```

Макрос LOOP

При помощи макроса `loop` организуются циклы:

```
(defun row (n)
  (loop for i from 1 to n collect i))

> (row 10)
(1 2 3 4 5 6 7 8 9 10)

(defun factorial (n)
  (reduce #'* (loop for a from 1 to n collect a)))

> (factorial 4)
24

(defun cube-row (n)
  (loop for i from 1 to n collect (expt i 3)))

> (cube-row 10)
(1 8 27 64 125 216 343 512 729 1000)

(defun 10+ (w)
```

```

(loop for i in w collect (+ i 10)))

> (10+ '(1 2 3 4))
(11 12 13 14)

(defun repeat (a n)
  (loop repeat n collect a))

> (repeat 2 10)
(2 2 2 2 2 2 2 2 2 2)

(defun rand (r n)
  (loop for i from 1 to n collect (random r)))

> (rand 10 5)
(5 3 4 4 6)

(defun sum (n)
  (loop for i from 1 to n sum i))

> (sum 8725)
38067175

(defun even/odd (w)
  (loop for i in w
    if (evenp i)
      collect i into evens
    else collect i into odds
    finally (return (values evens odds))))

> (even/odd '(1 2 3 4 5 6))
(2 4 6)
(1 3 5)

```

Функция *FORMAT*

Год создания Лиспа:

```

> (format nil "~r" 1958)
"one thousand, nine hundred and fifty-eight"
> (format nil "~:r" 1958)
"one thousand, nine hundred fifty-eighth"
> (format nil "~:@r" 1958)
"MDCCCCLVIII"
> (format nil "~@r" 1958)
"MCMLVIII"

```

Год падения Западной Римской империи (Imperium Romanum Occidentale):

```

> (format nil "~:@r" 476)
"CCCCLXXVI"
> (format nil "~@r" 476)
"CDLXXVI"

```

Легенда об изобретателе шахматной игры - количество зерен на 64-й клетке шахматной доски:

```
> (format nil "~r" (1- (expt 2 64)))
"eighteen quintillion, four hundred and forty-six quadrillion, seven
hundred and forty-four trillion, seventy-three billion, seven hundred
and nine million, five hundred and fifty-one thousand, six hundred and
fifteen"
```

```
> (format nil "~r" (1- (expt 10 66)))
"nine hundred and ninety-nine vigintillion, nine hundred and ninety-
nine novemdecillion, nine hundred and ninety-nine octodecillion, nine
hundred and ninety-nine septendecillion, nine hundred and ninety-nine
sexdecillion, nine hundred and ninety-nine quindecillion, nine hundred
and ninety-nine quattuordecillion, nine hundred and ninety-nine tre-
decillion, nine hundred and ninety-nine duodecillion, nine hundred and
ninety-nine undecillion, nine hundred and ninety-nine decillion, nine
hundred and ninety-nine nonillion, nine hundred and ninety-nine octil-
lion, nine hundred and ninety-nine septillion, nine hundred and nine-
ty-nine sextillion, nine hundred and ninety-nine quintillion, nine
hundred and ninety-nine quadrillion, nine hundred and ninety-nine
trillion, nine hundred and ninety-nine billion, nine hundred and nine-
ty-nine million, nine hundred and ninety-nine thousand, nine hun-
dred and ninety-nine"
```

Множественное число -s:

```
> (format nil "~r war~:p" 1)
"one war"
> (format nil "~r war~:p" 2)
"two wars"
```

Множественное число -y или -ies:

```
> (format nil "~r histor~:@p" 1)
"one history"
> (format nil "~r histor~:@p" 2)
"two histories"
```

В императивных языках программа представляет собой набор команд (среди которых могут находиться и вызовы функций). В функциональных языках программа представляет собой совокупность функций. Функции определяются через другие функции.

Функциональное программирование позволяет представить всю программу в виде одной функции, которая вызывает другие функции. Примеры применения функций к решению задач представлены в следующем разделе.

Примеры решений

Perfection is achieved, not when there is nothing more to add, but when there is nothing left to take away.

Antoine de Saint-Exupery

Вещи, сделанные без примеров, таковы, что их сущности надо бояться.

Вильям Шекспир

Структура 1 (функция АТОМ) > АТОМ

У меня в голове опилки. Длинные слова меня только расстраивают.

Алан Милн «Винни-Пух и все-все-все»

Далее, для краткости и единообразия: а - атом, n и m - числа, s - строка, w - больший список, v - меньший список, асс - больший список-аккумулятор, ас - меньший список-аккумулятор.

Задача 1.1 *inc.lisp*

Определить функцию, которая увеличивает число на 1.

Решение 1.1.1

```
(defun inc (n)
  (+ n 1))
```

```
> (inc 100)
101
```

Задача 1.2 *dec.lisp*

Определить функцию, которая уменьшает число на 1.

Решение 1.2.1

```
(defun dec (n)
  (- n 1))
```

```
> (dec 100)
```

Задача 1.3 1000+.lisp

Определить функцию, которая увеличивает число на 1000.

Решение 1.3.1

```
(defun 1000+ (n)
  (+ n 1000))
```

```
> (1000+ 10)
1010
```

Задача 1.4 triple-number.lisp

Определить функцию, которая умножает число на 3.

Решение 1.4.1

```
(defun triple-number (number)
  (* number 3))
```

```
> (triple-number 1000)
3000
```

Решение 1.4.2

```
(defun triple-number (number)
  (+ number number number))
```

```
> (triple-number 1000)
3000
```

Задача 1.5 years-seconds.lisp

Дано количество лет и средняя продолжительность года - 31556926 секунд. Определить функцию, возвращающую количество секунд.

Решение 1.5.1

```
(defun years-seconds (n)
  (* n 31556926))
```

```
> (years-seconds 1)
31556926
> (format nil "~r" (years-seconds 1))
"thirty-one million, five hundred and fifty-six thousand, nine hundred
and twenty-six"
> (years-seconds 100)
3155692600
```

```
> (format nil "~r" (years-seconds 100))
"three billion, one hundred and fifty-five million, six hundred and
ninety-two thousand, six hundred"
> (years-seconds 2014)
63555648964
> (format nil "~r" (years-seconds 2014))
"sixty-three billion, five hundred and fifty-five million, six hundred
and forty-eight thousand, nine hundred and sixty-four"
```

Задача 1.6 *julian-years-seconds.lisp*

Дано количество лет и средняя продолжительность года в юлианском календаре – 31557600 секунд СИ (точно). Определить функцию, возвращающую количество секунд.

Решение 1.6.1

```
(defun julian-years-seconds (n)
  (* n 31557600))

> (julian-years-seconds 1)
31557600
> (format nil "~r" (julian-years-seconds 1))
"thirty-one million, five hundred and fifty-seven thousand, six hun-
dred"
> (julian-years-seconds 100)
3155760000
> (format nil "~r" (julian-years-seconds 100))
"three billion, one hundred and fifty-five million, seven hundred and
sixty thousand"
> (julian-years-seconds 2014)
63557006400
> (format nil "~r" (julian-years-seconds 2014))
"sixty-three billion, five hundred and fifty-seven million, six thou-
sand, four hundred"
```

Задача 1.7 *gregorian-years-seconds.lisp*

Дано количество лет и средняя продолжительность года в григорианском календаре – 365,2425 дня (точно). Определить функцию, возвращающую количество секунд.

Решение 1.7.1

```
(defun gregorian-years-seconds (n)
  (/ (* n 3652425 24 60 60) 10000))

> (gregorian-years-seconds 1)
31556952
> (format nil "~r" (gregorian-years-seconds 1))
"thirty-one million, five hundred and fifty-six thousand, nine hundred"
```

```

and fifty-two"
> (gregorian-years-seconds 100)
3155695200
> (format nil "~r" (gregorian-years-seconds 100))
"three billion, one hundred and fifty-five million, six hundred and
ninety-five thousand, two hundred"
> (gregorian-years-seconds 2014)
63555701328
> (format nil "~r" (gregorian-years-seconds 2014))
"sixty-three billion, five hundred and fifty-five million, seven hun-
dred and one thousand, three hundred and twenty-eight"

```

Задача 1.8 digits-expt-125-100.lisp

Найти из скольких цифр состоит число 125 в степени 100.

Решение 1.8.1

```

> (expt 125 100)
4909093465297726553095771954986275642975215512499449565111549117187105
2547217158564600978840373319522771835715651318785131679186104247189028
0751482410896345225310546445986192853894181098439730703830718994140625
> (write-to-string (expt 125 100))
"490909346529772655309577195498627564297521551249944956511154911718710
5254721715856460097884037331952277183571565131878513167918610424718902
8075148241089634522531054644598619285389418109843973070383071899414062
5"
> (loop for a across (write-to-string (expt 125 100)) collect (string
a))
("4" "9" "0" "9" "0" "9" "3" "4" "6" "5" "2" "9" "7" "7" "2" "6" "5"
"5" "3" "0" "9" "5" "7" "7" "1" "9" "5" "4" "9" "8" "6" "2" "7" "5"
"6" "4" "2" "9" "7" "5" "2" "1" "5" "5" "1" "2" "4" "9" "9" "4" "4"
"9" "5" "6" "5" "1" "1" "1" "5" "4" "9" "1" "1" "7" "1" "8" "7" "1"
"0" "5" "2" "5" "4" "7" "2" "1" "7" "1" "5" "8" "5" "6" "4" "6" "0"
"0" "9" "7" "8" "8" "4" "0" "3" "7" "3" "3" "1" "9" "5" "2" "2" "7"
"7" "1" "8" "3" "5" "7" "1" "5" "6" "5" "1" "3" "1" "8" "7" "8" "5"
"1" "3" "1" "6" "7" "9" "1" "8" "6" "1" "0" "4" "2" "4" "7" "1" "8"
"9" "0" "2" "8" "0" "7" "5" "1" "4" "8" "2" "4" "1" "0" "8" "9" "6"
"3" "4" "5" "2" "2" "5" "3" "1" "0" "5" "4" "6" "4" "4" "5" "9" "8"
"6" "1" "9" "2" "8" "5" "3" "8" "9" "4" "1" "8" "1" "0" "9" "8" "4"
"3" "9" "7" "3" "0" "7" "0" "3" "8" "3" "0" "7" "1" "8" "9" "9" "4"
"1" "4" "0" "6" "2" "5")
> (length (loop for a across (write-to-string (expt 125 100)) collect
(string a)))
210

```

Решение 1.8.2

```

(defun digits-expt-125-100 ()
  (length (loop for a across (write-to-string (expt 125 100))
    collect (string a))))

```

```
> (digits-expt-125-100)
210
```

Решение 1.8.3

```
(defun digits-expt-100 (n)
  (length (loop for a across (write-to-string (expt n 100))
                collect (string a))))
```

```
> (digits-expt-100 125)
210
```

Решение 1.8.4

```
(defun digits-expt (n m)
  (length (loop for a across (write-to-string (expt n m))
                collect (string a))))
```

```
> (digits-expt 125 100)
210
```

Решение 1.8.5

```
(defun digits-expt-125-100 ()
  (loop for a across (write-to-string (expt 125 100))
        counting a))
```

```
> (digits-expt-125-100)
210
```

Решение 1.8.6

```
(defun digits-expt (n m)
  (loop for a across (write-to-string (expt n m))
        count a))
```

```
> (digits-expt 125 100)
210
```

Решение 1.8.7

```
(defun digits-expt (n m)
  (count-digits (expt n m)))

(defun count-digits (a)
  (cond ((< a 1) 0)
        (t (1+ (count-digits (/ a 10))))))
```

```
> (digits-expt 125 100)
210
```

Решение 1.8.8

```
(defun digits-expt (n m)
  (count-digits (expt n m)))

(defun count-digits (a)
  (if (< a 1) 0 (1+ (count-digits (/ a 10)))))

> (digits-expt 125 100)
210
```

Решение 1.8.9

```
(defun digits-expt (n m)
  (labels ((count-digits (a)
            (if (< a 1) 0 (1+ (count-digits (/ a 10)))))
    (count-digits (expt n m))))

> (digits-expt 125 100)
210
```

Решение 1.8.10

```
(defun digits-expt (n m)
  (length (write-to-string (expt n m))))

> (digits-expt 125 100)
210
```

Задача 1.9 romanp.lisp

Для представления римских цифр используются символы: I - один, V - пять, X - десять, L - пятьдесят. С - сто, D - пятьсот, M - тысяча. Для изображения числа с помощью римских цифр используются общеизвестные правила. Так, например, 482 - CDLXXXII. 1999 - MCMXCIX. Напишите функцию (f R), которая проверяет, правильна ли запись некоторого числа римскими цифрами R. Например, (f IIIC) = nil.

Решение 1.9.1

```
(defun romanp (r &optional (n 3999))
  (cond ((zerop n) nil)
        ((string= (string r) (format nil "~@r" n)) t)
        ((romanp r (1- n)))))

> (romanp 'CDLXXVI)
T
> (romanp 'IIIC)
NIL
```

Решение 1.9.2

```
(defun romanp (r)
  (loop for a from 1 to 3999
```

```

when (string= (string r) (format nil "~@r" a)) collect a))

> (romanp 'CDLXXVI)
(476)
> (romanp 'IIIC)
NIL

```

Решение 1.9.3

```

(defun romanp (r &optional (n 3999))
  (cond ((zerop n) nil)
        ((string= (string r) (format nil "~@r" n)) n)
        ((romanp r (1- n)))))

> (romanp 'CDLXXVI)
476
> (romanp 'IIIC)
NIL

```

Решение 1.9.4

```

(defun romanp (r)
  (labels ((roman (r n)
            (cond ((zerop n) nil)
                  ((string= (string r) (format nil "~@r" n)) n)
                  ((roman r (1- n)))))
    (roman r 3999)))

> (romanp 'CDLXXVI)
476
> (romanp 'IIIC)
NIL

```

Решение 1.9.5

```

(defun romanp (n)
  (member n (mapcar #'roman (loop for i from 1 to 3999 collect i))))

(defun roman (n)
  (read-from-string (format nil "~@r" n)))

> (romanp 'MMXII)
(MMXII MMXIII MMXIV MMXV MMXVI MMXVII MMXVIII MMXIX MMXX MMXXI MMXXII
 MMXXIII MMXXIV MMXXV MMXXVI MMXXVII MMXXVIII MMXXIX MMXXX MMXXXI
 MMXXXII MMXXXIII MMXXXIV MMXXXV MMXXXVI MMXXXVII MMXXXVIII MMXXXIX
 MMXL MMXLI MMXLII MMXLIII MMXLIV MMXLV MMXLVI MMXLVII MMXLVIII MMXLIX
 MML MMLI MMLII MMLIII MMLIV MMLV MMLVI MMLVII MMLVIII MMLIX MMLX MMLXI
 MMLXII MMLXIII MMLXIV MMLXV MMLXVI MMLXVII MMLXVIII MMLXIX MMLXX
 MMLXXI MMLXXII MMLXXIII MMLXXIV MMLXXV MMLXXVI MMLXXVII MMLXXVIII
 MMLXXIX MMLXXX MMLXXXI MMLXXXII MMLXXXIII MMLXXXIV MMLXXXV ... )

> (romanp 'IIIC)
NIL

```

Решение 1.9.6 (для XLISP)

```
(defun romanp (n)
  (member n '(I II III IV V VI VII VIII IX X XI XII XIII XIV XV XVI
XVII XVIII XIX XX XXI XXII XXIII XXIV XXV XXVI XXVII XXVIII XXIX XXX
XXXI XXXII XXXIII XXXIV XXXV XXXVI XXXVII XXXVIII XXXIX XL XLI XLII
XLIII XLIV XLV XLVI XLVII XLVIII XLIX L LI LII LIII LIV LV LVI LVII
LVIII LIX LX LXI LXII LXIII LXIV LXV LXVI LXVII LXVIII LIX LXX LXXI
LXXII LXXIII LXXIV LXXV LXXVI LXXVII LXXVIII LXXIX LXXX ... MMMCMLXXI
MMMCMLXXII MMMCMLXXIII MMMCMLXXIV MMMCMLXXV MMMCMLXXVI MMMCMLXXVII
MMMCMLXXVIII MMMCMLXXIX MMMCMLXXX MMMCMLXXXI MMMCMLXXXII MMMCMLXXXIII
MMMCMLXXXIV MMMCMLXXXV MMMCMLXXXVI MMMCMLXXXVII MMMCMLXXXVIII
MMCMCMLXXXIX MMCMCMXC MMCMCMXCI MMCMCMXCII MMCMCMXCIII MMCMCMXCIV MMCMCMXCV
MMCMCMXCVI MMCMCMXCVII MMCMCMXCVIII MMCMCMXCIX)))

> (romanp 'MMXII)
(MMXII MMXIII MMXIV MMXV MMXVI MMXVII MMXVIII MMXIX MMXX MMXXI MMXXII
MMXXIII MMXXIV MMXXV MMXXVI MMXXVII MMXXVIII MMXXIX MMXXX MMXXXI
MMXXXII MMXXXIII MMXXXIV MMXXXV MMXXXVI MMXXXVII MMXXXVIII MMXXXIX
MMXL MMXLI MMXLII MMXLIII MMXLIV MMXLV MMXLVI MMXLVII MMXLVIII MMXLIX
MML MMLI MMLII MMLIII MMLIV MMLV MMLVI MMLVII MMLVIII MMLIX MMLX MMLXI
MMLXII MMLXIII MMLXIV MMLXV MMLXVI MMLXVII MMLXVIII MMLXIX MMLXX
MMLXXI MMLXXII MMLXXIII MMLXXIV MMLXXV MMLXXVI MMLXXVII MMLXXVIII
MMLXXIX MMLXXX MMLXXXI MMLXXXII MMLXXXIII MMLXXXIV MMLXXXV ... )

> (romanp 'IIIC)
NIL
```

Задача 1.10 special-pythagorean-triplet-product.lisp

It will give you insight into other languages. It's the mountain-top from which you can see everything. In some sense, every other language is clumsy corruption of Lisp.

Christopher Reiss

A Pythagorean triplet is a set of three natural numbers, $a < b < c$, for which, $a^2 + b^2 = c^2$. For example, $3^2 + 4^2 = 5^2$. There exists exactly one Pythagorean triplet for which $a + b + c = 1000$. Find the product abc .

Тройка Пифагора или пифагоровы числа – названы в честь Пифагора Самосского, 570–490 гг. до н. э., древнегреческого философа, математика и мистика, создателя религиозно-философской школы пифагорейцев, хотя открыты значительно раньше,

Наиболее известной в развитых древних культурах была тройка 3, 4 и 5, которая позволяла древним строить прямые углы. римский архитектор и механик Витрувий (I век до н. э.) считал эту тройку высшим достижением математики, а древнегреческий философ Платон, ученик Сократа, учитель

Аристотеля, — символом супружества, что говорит о большом значении, которое придавали древние тройке 3, 4 и 5.

В архитектуре древнемесопотамских надгробий встречается равнобедренный треугольник, составленный из двух прямоугольных со сторонами 9, 12 и 15 локтей. Пирамиды фараона Снофру (XXVII век до н. э.) построены с использованием треугольников со сторонами 20, 21 и 29, а также 18, 24 и 30 десятков египетских локтей.

Пифагоровы числа — три натуральных числа $a < b < c$, для которых выполняется равенство $a^2 + b^2 = c^2$. Например, $3^2 + 4^2 = 5^2$. Существует только одна тройка Пифагора, для которой $a + b + c = 1000$. Найдите произведение abc .

Решение 1.10.1

По условию, a должно быть меньше 333 (иначе $333 + 334 + 335 = 1002$) и b должно быть меньше 500 (иначе $1 + 500 + 501 = 1002$).

```
(defun special-pythagorean-triplet (n)
  (loop for a from 1 to (/ n 3)
        nconc (loop for b from 1 to (/ n 2)
                    when (= (+ (expt a 2) (expt b 2))
                           (expt (- n a b) 2))
                    nconc (list a b (- n a b))))))

(defun special-pythagorean-triplet-product (n)
  (reduce #'* (special-pythagorean-triplet n)))

> (special-pythagorean-triplet 1000)
(200 375 425)
> (special-pythagorean-triplet-product 1000)
31875000
```

Решение 1.10.2

```
(defun special-pythagorean-triplet (n)
  (loop for a from 1 to (/ n 3)
        nconc (loop for b from 1 to (/ n 2)
                    when (= (+ (* a a) (* b b))
                           (expt (- n a b) 2))
                    nconc `(,a ,b ,(- n a b))))))

(defun special-pythagorean-triplet-product (n)
  (reduce #'* (special-pythagorean-triplet n)))

> (special-pythagorean-triplet 1000)
(200 375 425)
> (special-pythagorean-triplet-product 1000)
31875000
```

Задача 1.11 trim-words.lisp

Создать программу, которая удаляет первую букву каждого слова в предложении.

Решение 1.11.1

```
(defun cut (s)
  (mapcar #'(lambda (a) (subseq a 1))
    (mapcar #'string (read-from-string
      (concatenate 'string "(" s ")")))))

(defun glue (w)
  (cond ((null w) nil)
    ((concatenate 'string (car w) " " (glue (cdr w))))))

(defun trim-words(s)
  (string-right-trim " " (glue (cut s))))

> (trim-words "abc abc abc")
"BC BC BC"
```

Решение 1.11.2

```
(defun cut (s)
  (mapcar #'(lambda (a) (subseq a 1))
    (mapcar #'string (read-from-string
      (concatenate 'string "(" s ")")))))

(defun glue (w)
  (when w (concatenate 'string (car w) " " (glue (cdr w)))))

(defun trim-words(s)
  (string-right-trim " " (glue (cut s))))

> (trim-words "abc abc abc")
"BC BC BC"
```

Решение 1.11.3

```
(defun cut (s)
  (mapcar #'(lambda (a) (subseq a 1))
    (mapcar #'string (read-from-string
      (concatenate 'string "(" s ")")))))

(defun glue (w)
  (format nil "~{~a~^ ~}" w))

(defun trim-words(s)
  (glue (cut s)))

> (trim-words "abc abc abc")
"BC BC BC"
```

Задача 1.12 remove-extra-spaces.lisp

В заданном тексте слова разделены пробелами. Удалить лишние пробелы так, чтобы между словами осталось только по одному пробелу.

Решение 1.12.1

```
(defun remove-extra-spaces (s)
  (concat (mapcar #'string
                  (read-from-string
                   (concatenate 'string "(" s ")")))))

(defun concat (w &optional (ac ""))
  (cond ((null w) (string-right-trim '("#\space") ac))
        ((concat (cdr w) (concatenate 'string ac (car w) " "))))))

> (remove-extra-spaces "a ab  abc")
"A AB ABC"
```

Решение 1.12.2

```
(defun remove-extra-spaces (s)
  (concat (mapcar #'string
                  (read-from-string
                   (concatenate 'string "(" s ")")))))

(defun concat (w &optional (ac ""))
  (if w
      (concat (cdr w) (concatenate 'string ac (car w) " ")
              (string-right-trim '("#\space") ac)))
      ac))

> (remove-extra-spaces "a ab  abc")
"A AB ABC"
```

Решение 1.12.3

```
(defun remove-extra-spaces (s)
  (concat (mapcar #'string
                  (read-from-string
                   (concatenate 'string "(" s ")")))))

(defun concat (w)
  (labels ((conc (v ac)
             (if v (conc (cdr v)
                          (concatenate 'string ac (car v) " ")
                          (string-right-trim '("#\space") ac)))
                 (conc w ""))))
    (conc w "")))

> (remove-extra-spaces "a ab  abc")
"A AB ABC"
```

Решение 1.12.4

```
(defun remove-extra-spaces (s)
  (concat (mapcar #'string
                  (read-from-string
                   (concatenate 'string "(" s ")")))))

(defun concat (w)
  (format nil "~{~a~^ ~}" w))

> (remove-extra-spaces "a ab abc")
"A AB ABC"
```

Задача 1.13 max-lets.lisp

Найти в заданном тексте самое длинное слово.

Решение 1.13.1

```
(defun max-lets (s)
  (caar (sort (mapcar #'(lambda (a) (list a (length a)))
                    (mapcar #'string (read-from-string
                                     (concatenate 'string "(" s ")"))))
              #'> :key #'cadr)))
```

```
> (max-lets "a ab abc")
"ABC"
```

Комментарии (по функциям):

```
> (concatenate 'string "(" "a ab abc" ")")
"(a ab abc)"
> (read-from-string "(a ab abc)")
(A AB ABC)
10
> (mapcar #'string '(A AB ABC))
("A" "AB" "ABC")
> (mapcar #'(lambda (a) (list a (length a))) '("A" "AB" "ABC"))
(("A" 1) ("AB" 2) ("ABC" 3))
> (sort '(("A" 1) ("AB" 2) ("ABC" 3)) #'> :key #'cadr)
(("ABC" 3) ("AB" 2) ("A" 1))
> (caar '(("ABC" 3) ("AB" 2) ("A" 1)))
"ABC"
```

Решение 1.13.2

```
(defun max-lets (s)
  (caar
   (sort
    (mapcar
     #'(lambda (a) (list a (length a)))
     (mapcar
      #'string
```

```

(read-from-string
 (concatenate
  'string "("
  (remove-if-not #'(lambda (x)
                      (or (alpha-char-p x)
                          (equal x #\space)
                          (equal x #\-)))
                  s) ")"))))
#'> :key #'cadr)))

> (max-lets "a ab abc")
"ABC"
> (max-lets "jsadf .")
"JSADF"
> (max-lets "(asdf jlkj) lk")
"ASDF"
> (max-lets "#.(print (user-homedir-pathname))")
"USER-HOMEDIR-PATHNAME"

```

Решение 1.13.3

```

(defun max-lets (s)
  (caar
   (sort
    (mapcar
     #'(lambda (a) (list a (length a)))
     (mapcar
      #'string
      (read-from-string
       (concatenate
        'string "("
        (loop for c across s
              if (or (alpha-char-p c)
                    (equal c #\-))
                collect c
                else collect #\space) ")"))))
     #'> :key #'cadr)))

> (max-lets "a ab abc")
"ABC"
> (max-lets "jsadf .")
"JSADF"
> (max-lets "(asdf jlkj) lk")
"ASDF"
> (max-lets "#.(print (user-homedir-pathname))")
"USER-HOMEDIR-PATHNAME"
> (max-lets "foo+bar")
"FOO"
> (max-lets "f+bar")
"BAR"

```

Решение 1.13.4

```

(defun max-lets (s)
  (caar
   (sort
    (mapcar
     #'(lambda (a) (list a (length a)))
     (mapcar
      #'string
      (read-from-string
       (concatenate
        'string "("
        (substitute-if-not
         #\space
         #'(lambda (a) (or (alpha-char-p a)
                           (equal a #\-)))
         s) ")"))))
    #'> :key #'cadr)))

> (max-lets "a ab abc")
"ABC"
> (max-lets "jsadf .")
"JSADF"
> (max-lets "(asdf jlkj) lk")
"ASDF"
> (max-lets "#.(print (user-homedir-pathname))")
"USER-HOMEDIR-PATHNAME"
> (max-lets "foo+bar")
"FOO"
> (max-lets "f+bar")
"BAR"

```

Решение 1.13.5

```

(defun check (a)
  (or (alpha-char-p a) (equal a #\-)))

(defun aim (b r)
  (if (or (not b) (and r (> (length r) (length b)))) r b))

(defun max-lets (s
                 &optional
                 b
                 &aux
                 (n (position-if #'check s))
                 (m (when n (position-if-not #'check s :start n)))
                 (r (when n (if m (subseq s n m) (subseq s n)))))
  (cond ((not n) b)
        ((not m) (aim b r))
        ((max-lets (subseq s m) (aim b r)))))

> (max-lets "a ab abc")
"ABC"
> (max-lets "jsadf .")
"JSADF"

```

```

> (max-lets "(asdf jlkj) lk")
"ASDF"
> (max-lets "#.(print (user-homedir-pathname))")
"USER-HOMEDIR-PATHNAME"
> (max-lets "foo+bar")
"FOO"
> (max-lets "f+bar")
"BAR"

```

Решение 1.13.6

```

(defun check (a)
  (or (alpha-char-p a) (equal a #\-)))

(defun aim (b r)
  (if (or (not b) (and r (> (length r) (length b)))) r b))

(defun max-lets (s)
  (labels ((maxes (s
                  b
                  &aux
                  (n (position-if #'check s))
                  (m (when n (position-if-not #'check s :start n)))
                  (r (when n (if m (subseq s n m) (subseq s n)))))
    (cond ((not n) b)
          ((not m) (aim b r))
          ((maxes (subseq s m) (aim b r))))))
    (maxes s nil)))

> (max-lets "a ab abc")
"ABC"
> (max-lets "jsadf .")
"JSADF"
> (max-lets "(asdf jlkj) lk")
"ASDF"
> (max-lets "#.(print (user-homedir-pathname))")
"USER-HOMEDIR-PATHNAME"
> (max-lets "foo+bar")
"FOO"
> (max-lets "f+bar")
"BAR"

```

Задача 1.14 *dump-duplicates.lisp*

Из набора атомов вычеркнуть дубликаты, т.е. повторяющиеся: (1 2 3 4 5 1 2) > (1 2 3 4 5).

Решение 1.14.1

```

(defun dump-duplicates (w)
  (cond ((null w) nil)
        ((member (car w) (cdr w)) (dump-duplicates (cdr w)))

```

```
(t (cons (car w) (dump-duplicates (cdr w)))))
```

```
> (dump-duplicates '(1 2 3 4 5 1 2))
(3 4 5 1 2)
```

Решение 1.14.2

```
(defun dump-duplicates (w)
  (when w (if (member (car w) (cdr w))
              (dump-duplicates (cdr w))
              (cons (car w) (dump-duplicates (cdr w))))))
```

```
> (dump-duplicates '(1 2 3 4 5 1 2))
(3 4 5 1 2)
```

Решение 1.14.3

```
(defun dump-duplicates (w &aux (a (car w)) (d (cdr w)))
  (when w (if (member a d)
              (dump-duplicates d)
              (cons a (dump-duplicates d)))))
```

```
> (dump-duplicates '(1 2 3 4 5 1 2))
(3 4 5 1 2)
```

Макрос `when` проверяет список на непустоту, если список пуст, то вернет `nil`, то есть функция `dump-duplicates` добежит до конца списка `w` и уткнется в возвращенный `nil` - пустой список `()`, возможно включит последний элемент в начало пустого списка, затем включит предпоследний в начало возвращенного списка из одного (последнего) элемента и так далее, пока не включит первый элемент в начало возвращаемого списка.

Решение 1.14.4

```
(defun dump-duplicates (w)
  (when w ((lambda (a d)
              (if (member a d)
                  (dump-duplicates d)
                  (cons a (dump-duplicates d)))))
           (car w) (cdr w))))
```

```
> (dump-duplicates '(1 2 3 4 5 1 2))
(3 4 5 1 2)
```

```
(dump-duplicates "abcbddefdcf")
```

Evaluation aborted

P.S. в Common Lisp - `remove-duplicates` и `delete-duplicates` работают с любыми последовательностями, в том числе и строками:

```
(remove-duplicates "abcbddefdcf")
"abedcfg"
```


Задача 1.15 chess-prize.lisp

Написать функцию, которая вычисляет, сколько будет зёрен на шахматной доске, если, положив на первую клетку доски одно зерно, класть на каждую следующую клетку вдвое больше зёрен, чем на предыдущую.

Решение 1.15.1

```
(defun chess-prize (number-of-squares)
  (1- (expt 2 number-of-squares)))

> (chess-prize 64)
18446744073709551615

> (format nil "~r" (chess-prize 64))
"eighteen quintillion, four hundred and forty-six quadrillion, seven
hundred and forty-four trillion, seventy-three billion, seven hundred
and nine million, five hundred and fifty-one thousand, six hundred and
fifteen"
```

Решение 1.15.2

```
(defun chess-prize-recursive (number-of-squares &optional (acc 1))
  (cond ((zerop number-of-squares) (1- acc))
        ((chess-prize-recursive (1- number-of-squares) (* acc 2)))))

> (chess-prize-recursive 64)
18446744073709551615
```

Решение 1.15.3

```
(defun chess-prize-recursive (number-of-squares)
  (labels ((chess-prize (n ac)
            (cond ((zerop n) (1- ac))
                  ((chess-prize (1- n) (* ac 2))))))
    (chess-prize number-of-squares 1)))

> (chess-prize-recursive 64)
18446744073709551615
```

Задача 1.16 sort-vowels<.lisp

Дан текст. Записать слова текста в порядке возрастания количества гласных букв в слове.

Решение 1.16.1

```
(defun split (s &optional ac &aux (p (position #\Space s)))
  (cond ((zerop (length s)) ac)
        (p (split (subseq s (1+ p)) (cons (subseq s 0 p) ac)))
        ((cons (subseq s 0) ac))))
```

```

(defun ws (w)
  (mapcar #'(lambda (a) (coerce a 'list)) w))

(defun cn (w &optional (n 0))
  (cond ((null w) n)
        ((member (car w) '(#\a #\e #\i #\o #\u))
         (cn (cdr w) (1+ n)))
        ((cn (cdr w) n))))

(defun ez (w)
  (mapcar #'(lambda (e) (list e (cn e))) w))

(defun ts (w)
  (sort w #'< :key #'cadr))

(defun join (w)
  (apply #'concatenate 'string (mapcar #'string w)))

(defun dm (w)
  (cond ((null w) nil)
        ((cons (join (caar w)) (dm (cdr w))))))

(defun ju (w)
  (format nil "~{~A~^ ~}" w))

(defun sort-vowels< (w)
  (ju (dm (ts (ez (ws (split w)))))))

> (sort-vowels< "aaaw aaw aww w")
"w aww aaw aaaw"

```

Задача 1.17 select-words.lisp

Learning Lisp will make you smarter even if
you never use Lisp.

Luis Argerich

В строке оставить только слова, содержащие заданную комбинацию букв.

Решение 1.17.1

```

(defun occur (x s)
  (delete-if-not #'(lambda (a) (search x a)) s))

(defun split (s)
  (loop for i = 0 then (1+ j)
        as j = (position #\Space s :start i)
        collect (subseq s i j)
        while j))

(defun join (w)
  (format nil "~{~A~^ ~}" w))

```

```
(defun select-words (c s)
  (join (occur c (split s))))

> (select-words "bc" "abc bcd cde")
"abc bcd"
```

Решение 1.17.2 (без использования циклов - рекурсия или/и функционалы)

```
(defun occur (x s)
  (delete-if-not #'(lambda (a) (search x a)) s))

(defun split (s &optional ac &aux (p (position #\Space s)))
  (cond ((zerop (length s)) ac)
        (p (split (subseq s (1+ p)) (cons (subseq s 0 p) ac)))
        ((cons (subseq s 0) ac))))

(defun join (w)
  (format nil "~{~A~^ ~}" w))

(defun select-words (c s)
  (join (occur c (reverse (split s)))))

> (select-words "bc" "abc bcd cde")
"abc bcd"
```

Задача 1.18 string-upcase.lisp

Написать функцию изменения регистра строки: прописные, строчные или как в предложении.

Решение 1.18.1

```
> (string-upcase "ab cd")
"AB CD"

> (string-downcase "AB CD")
"ab cd"

> (string-capitalize "ab cd" :end 1)
"Ab cd"
```

Задача 1.19 2exptp.lisp

Написать функцию, которая для аргумента-числа проверяет, является ли оно степенью двойки.

Решение 1.19.1

```
(defun 2exptp (n)
  (cond ((= n 2) t)
```

```
((< n 2) nil)
((2exptp (/ n 2))))
```

```
> (2exptp 4)
T
```

Решение 1.19.2

```
(defun 2exptp (n)
  (cond ((> n 2) (2exptp (/ n 2)))
        ((= n 2))))
```

```
> (2exptp 4)
T
```

Задача 1.20 factorial.lisp

Computer languages are weapons. But the power of computer programs decided by one who write it, not by what it made of. To write lisp program we need more effort, so it is more powerful the lisp programmers are, not the lisp language is.

~

Определить функцию, возвращающую факториал числа n (лат. *factorialis* — действующий, производящий, умножающий; обозначается $n!$, произносится эн факториал) — произведение всех натуральных чисел от 1 до n включительно.

Факториал определён только для целых неотрицательных чисел.

Последовательность факториалов неотрицательных целых чисел начинается так: 1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800, 39916800, 479001600, 6227020800, 87178291200, 1307674368000, 20922789888000, 355687428096000, 6402373705728000, 121645100408832000, 2432902008176640000, ...

Решение 1.20.1

```
(defun factorial (n)
  (if (zerop n) 1 (* n (factorial (- n 1)))))
```

```
> (factorial 4)
24
```

```
> (defun factorial (n)
  (if (zerop n) 1 (* n (factorial (- n 1)))))
FACTORIAL
> (trace factorial)
;; Tracing function FACTORIAL.
(FACTORIAL)
> (factorial 4)
```



```

24
> (untrace factorial)
(FACTORIAL)
> (time (factorial 20))
Real time: 0.0 sec.
Run time: 0.0 sec.
Space: 232 Bytes
2432902008176640000
> (time (factorial 200))
Real time: 0.0 sec.
Run time: 0.0 sec.
Space: 19380 Bytes
7886578673647905035523632139321850622951359776871732632947425332443594
4996340334292030428401198462390417721213891963883025764279024263710506
1926624952829931113462857270763317237396988943922445621451664240254033
2918641312274282948532775242424075739032403212574055795686602260319041
70324062351700858796178922222789623703897374720000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000

```

Решение 1.20.3

```

(defun factorial (n)
  (loop for i from 1 to n
        for fac = 1 then (* fac i)
        finally (return fac)))

> (factorial 4)
24
> (defun factorial (n)
  (loop for i from 1 to n
        for fac = 1 then (* fac i)
        finally (return fac)))
FACTORIAL
> (trace factorial)
;; Tracing function FACTORIAL.
(FACTORIAL)
> (factorial 4)
1. Trace: (FACTORIAL '4)
1. Trace: FACTORIAL ==> 24
24
> (untrace factorial)
(FACTORIAL)
> (time (factorial 20))
Real time: 0.0 sec.
Run time: 0.0 sec.
Space: 152 Bytes
2432902008176640000
> (time (factorial 200))
Real time: 0.0 sec.
Run time: 0.0 sec.
Space: 15792 Bytes
7886578673647905035523632139321850622951359776871732632947425332443594
4996340334292030428401198462390417721213891963883025764279024263710506

```

```

1926624952829931113462857270763317237396988943922445621451664240254033
2918641312274282948532775242424075739032403212574055795686602260319041
70324062351700858796178922227896237038973747200000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000

```

Задача 1.21 factorial+.lisp

Определить функцию, которая вычисляет сумму промежуточных значений нахождения факториала.

Решение 1.21.1

```

(defun factorial (n)
  (if (zerop n) 1 (* (factorial (1- n)) n)))

(defun factorial+ (n)
  (if (zerop n) 0 (+ (factorial+ (1- n)) (factorial n))))

> (factorial+ 4)
33
> (defun factorial (n)
  (if (zerop n) 1 (* (factorial (1- n)) n)))
FACTORIAL
> (trace factorial)
;; Tracing function FACTORIAL.
(FACTORIAL)
> (factorial 4)
1. Trace: (FACTORIAL '4)
2. Trace: (FACTORIAL '3)
3. Trace: (FACTORIAL '2)
4. Trace: (FACTORIAL '1)
5. Trace: (FACTORIAL '0)
5. Trace: FACTORIAL ==> 1
4. Trace: FACTORIAL ==> 1
3. Trace: FACTORIAL ==> 2
2. Trace: FACTORIAL ==> 6
1. Trace: FACTORIAL ==> 24
24
> (untrace factorial)
(FACTORIAL)
> (factorial 4)
24

```

Решение 1.21.2

```

(defun factorial+ (n)
  (loop for a from 1 to n
        for m = a then (* a m) sum m))

> (factorial+ 4)
33
> (macroexpand-1

```

```

      '(loop for a from 1 to 4
        for m = a then (* a m) sum m))
(MACROLET ((LOOP-FINISH NIL (SYSTEM::LOOP-FINISH-ERROR)))
  (BLOCK NIL
    (LET ((A 1))
      (PROGN
        (LET ((M NIL))
          (LET ((#:G1382 0))
            (MACROLET ((LOOP-FINISH NIL '(GO SYSTEM::END-LOOP)))
              (TAGBODY (WHEN (> A 4) (LOOP-FINISH)) (SETQ M A) SYSTEM::BEGIN-
LOOP
                (PROGN (SETQ #:G1382 (+ #:G1382 M)) (PSETQ A (+ A 1))
                  (WHEN (> A 4) (LOOP-FINISH)) (PSETQ M (* A M)) (GO SYS-
TEM::BEGIN-LOOP)
                  SYSTEM::END-LOOP
                  (MACROLET
                    ((LOOP-FINISH NIL (SYSTEM::LOOP-FINISH-WARN) '(GO SYS-
TEM::END-LOOP)))
                    (RETURN-FROM NIL #:G1382))))))))))
T

```

Задача 1.22 char-sum-code+.lisp

If you are interested in Lisp, you should learn it. Either it will colonize your brain and you will become a Lisp hacker, or it will be incompatible with your brain and you will not. If you become a Lisp hacker, you will be happier and more productive, and will henceforth understand all computation in terms of Lisp.

Mike Levins

Определить функцию, которая вычисляет сумму кодов знаков строки и возвращает значок, кодом которого является найденная сумма.

Решение 1.22.1

```

(defun char-sum-code (s)
  (code-char (reduce #'+ (map 'list #'char-code s))))

> (char-sum-code "p3")
#\POUND_SIGN
> (char-sum-code "p3js3")
#\LATIN_CAPITAL_LETTER_Y_WITH_HOOK

```

Решение 1.22.1

```

(defun char-sum-code (s)
  (code-char (reduce #'+ (map 'list #'char-code (string s)))))

> (char-sum-code '|p3|)

```



```
#\POUND_SIGN
> (char-sum-code '|p3Js3|)
#\LATIN_CAPITAL_LETTER_G_WITH_HOOK
```

Задача 1.23 $2/1+3/2...+[n+1]/n.lisp$

Определить функцию, которая вычисляет сумму ряда $2/1+3/2...+(n+1)/n$.

Решение 1.23.1

```
(defun 2/1+3/2...+[n+1]/n (n)
  (if (zerop n) 0 (+ (/ (1+ n) n) (2/1+3/2...+[n+1]/n (1- n)))))
> (2/1+3/2...+[n+1]/n 2)
7/2
```

Решение 1.23.2

```
(defun 2/1+3/2...+[n+1]/n (n)
  (loop for a from 1 to n summing (/ (1+ a) a)))
> (2/1+3/2...+[n+1]/n 2)
7/2
```

Задача 1.24 num-magic-ticket-9999.lisp

Найти число "счастливых" билетов (сумма первых двух цифр равна сумме последних двух цифр) с номерами от 0000 до 9999 включительно.

Решение 1.24.1

```
(defun num-magic-ticket-9999 (n)
  (row-ticket (add-nine n) n))

(defun row-ticket (nn n &aux (m (int-digit nn)) (z (add-zero m n)))
  (if (>= nn 0)
      (if (check-balance z (/ n 2))
          (1+ (row-ticket (1- nn) n))
          (row-ticket (1- nn) n))
      0))

(defun add-nine (n &optional w)
  (if (< (length w) n) (add-nine n (cons 9 w)) (list-num w)))

(defun list-num (w)
  (parse-integer (apply #'concatenate 'string
                        (mapcar #'write-to-string w))))

(defun add-zero (w n)
  (if (< (length w) n) (add-zero (cons 0 w) n) w))
```

```
(defun int-digit (n)
  (loop with r do (setf (values n r) (truncate n 10))
    collect r into ac until (zerop n)
    finally (return (nreverse ac))))

(defun check-balance (w n)
  (= (apply #'+ (subseq w 0 n)) (apply #'+ (nthcdr n w))))

> (num-magic-ticket-9999 4)
670
```

Решение 1.24.2

```
(defun num-magic-ticket-9999 (n)
  (row-ticket (add-nine n) n))

(defun row-ticket (nn n &aux (m (int-digit nn)) (z (add-zero m n)))
  (if (>= nn 0)
    (if (check-balance z (/ n 2))
      (1+ (row-ticket (1- nn) n))
      (row-ticket (1- nn) n))
    0))

(defun add-nine (n &optional w)
  (if (< (length w) n) (add-nine n (cons 9 w)) (list-num w)))

(defun list-num (w)
  (parse-integer (apply #'concatenate 'string
    (mapcar #'write-to-string w))))

(defun add-zero (w n)
  (if (< (length w) n) (add-zero (cons 0 w) n) w))

(defun int-digit (n &optional ac)
  (if (zerop n) ac (int-digit (truncate n 10) (cons (rem n 10) ac))))

(defun check-balance (w n)
  (= (apply #'+ (subseq w 0 n)) (apply #'+ (nthcdr n w))))

> (num-magic-ticket-9999 4)
670
```

Задача 1.25 *sum-odd-plus.lisp*

Определить функцию, которая находит сумму положительных нечётных чисел меньше 100.

Решение 1.25.1

```
(defun sum-odd-plus (n)
  (cond ((zerop n) 0)
        ((oddp n) (+ n (sum-odd-plus (1- n)))))
```

```
((sum-odd-plus (1- n))))
> (sum-odd-plus 99)
2500
```

Задача 1.26 longest-palindrome.lisp

Составить программу, которая в заданном предложении находит самое длинное симметричное слово.

Решение 1.26.1

```
(defun palindrome (w) (equalp w (reverse w)))

(defun palindromes (w)
  (loop for a in w when (palindrome a) collect a))

(defun cleave (s)
  (mapcar #'string (read-from-string
                    (concatenate 'string "(" s ")"))))

(defun longest-palindrome (s)
  (car (sort (palindromes (cleave s)) #'> :key #'length)))

> (longest-palindrome "anna hannah")
"HANNAH"
```

Решение 1.26.2

```
(defun dromes (w)
  (loop for a in w when (equal a (reverse a))
        collect a))

(defun longest-palindrome (s)
  (car (sort (dromes (mapcar #'string s))
            #'> :key #'length)))

> (longest-palindrome '(anna hannah))
"HANNAH"
```

Решение 1.26.3

```
(defun dromes (w)
  (loop for a in w when (equal a (reverse a))
        collect a))

(defun longest-palindrome (s)
  (read-from-string
   (car (sort (dromes (mapcar #'string s))
             #'> :key #'length))))

> (longest-palindrome '(anna hannah))
```

HANNAH

6

Решение 1.26.4

```
(defun dromes (w)
  (loop for a in w when (equal a (reverse a))
        collect a))

(defun longest (w &optional b &aux (a (car w)))
  (if w
      (if (> (length a) (length b))
          (longest (cdr w) a)
          (longest (cdr w) b))
      b))

(defun longest-palindrome (s)
  (read-from-string (longest (dromes (mapcar #'string s))))))

> (longest-palindrome '(anna hannah))
HANNAH
6
```

Решение 1.26.5

```
(defun dromes (w)
  (loop for a in w when (equal a (reverse a))
        collect a))

(defun longest (w &optional b &aux (a (car w)))
  (cond ((null w) b)
        ((> (length a) (length b)) (longest (cdr w) a))
        ((longest (cdr w) b))))

(defun longest-palindrome (s)
  (read-from-string (longest (dromes (mapcar #'string s))))))

> (longest-palindrome '(anna hannah))
HANNAH
6
```

Задача 1.27 remove-repetition.lisp

Дана строка, содержащая предложение. В предложении некоторые из слов записаны подряд несколько раз (предложение заканчивается точкой или знаком восклицания). Получить в новой строке отредактированный текст, в котором удалены подряд идущие вхождения слов в предложении.

Решение 1.27.1

```
(defun drop-words (w)
  (cond ((null w) nil)
```

```

      ((eq (car w) (cadr w)) (drop-words (cdr w)))
      ((cons (car w) (drop-words (cdr w)))))

(defun glue (w m)
  (cond ((null (cdr w)) (concatenate 'string (car w) m))
        ((concatenate 'string (car w) " " (glue (cdr w) m)))))

(defun list-word (s)
  (read-from-string (concatenate 'string "(" s ")")))

(defun remove-repetition (s &aux
                          (n (1- (length s)))
                          (z (subseq s 0 n))
                          (m (subseq s n)))
  (glue (mapcar 'string (drop-words (list-word z))) m))

> (remove-repetition "aa aa aa bb bb.")
"AA BB."

```

Решение 1.27.2

```

(defun drop-words (w)
  (cond ((null w) nil)
        ((eq (car w) (cadr w)) (drop-words (cdr w)))
        ((cons (car w) (drop-words (cdr w)))))

(defun glue (w m)
  (if (cdr w)
      (concatenate 'string (car w) " " (glue (cdr w) m))
      (concatenate 'string (car w) m)))

(defun list-word (s)
  (read-from-string (concatenate 'string "(" s ")")))

(defun remove-repetition (s &aux
                          (n (1- (length s)))
                          (z (subseq s 0 n))
                          (m (subseq s n)))
  (glue (mapcar 'string (drop-words (list-word z))) m))

> (remove-repetition "aa aa aa bb bb.")
"AA BB."

```

Решение 1.27.3

```

(defun drop-words (w)
  (cond ((null w) nil)
        ((eq (car w) (cadr w)) (drop-words (cdr w)))
        ((cons (car w) (drop-words (cdr w)))))

(defun glue (w)
  (if (cdr w)
      (concatenate 'string (car w) " " (glue (cdr w)))

```

```

(car w)))

(defun list-word (s)
  (read-from-string (concatenate 'string (" s "))))

(defun remove-repetition (s &aux
                          (n (1- (length s)))
                          (z (subseq s 0 n))
                          (m (subseq s n)))
  (concatenate 'string
    (glue (mapcar 'string
                  (drop-words (list-word z))))
    m))

> (remove-repetition "aa aa aa bb bb.")
"AA BB."

```

Задача 1.28 *set-length.lisp*

Подсчитать, сколько различных символов содержатся в строке.

Решение 1.28.1

```

(defun set-length (s) (length (remove-duplicates s)))

> (set-length "aabbc")
3

```

Задача 1.29 *display.lisp*

Средствами текстового редактора подготовить файл, содержащий два столбца числовых значений (значения аргумента и функции), и написать программу, выводящую эти числа на экран.

Решение 1.29.1

```

d:/old.txt:
11 12
13 14

(defun display (path)
  (with-open-file (s path :direction :input)
    (do ((line (read-line s nil :eof) (read-line s nil :eof)))
        ((eql line :eof)) (print line))))

> (display "d:/old.txt")
"11 12"
"13 14"
NIL

```

Задача 1.30 num-substr.lisp

Напишите функцию, которая в заданной строке определяет количество вхождений в нее заданной подстроки (слова).

Решение 1.30.1 * возвращает число разобщенных (дизъюнктных) подстрок

```
(defun num-substr
  (a s &optional (m (length a)) (z 0) &aux (n (search a s)))
  (if n (num-substr a (subseq s (+ n m)) m (1+ z)) z))

> (num-substr "jar" "jar jar")
2
> (num-substr "ja" "jar jar")
2
> (num-substr "r" "jar jar")
2
```

Решение 1.30.2 * возвращает число связанных (конъюнктивных) подстрок

```
(defun num-substr
  (a s &optional (z 0) &aux (n (search a s)))
  (if n (num-substr a (subseq s (1+ n)) (1+ z)) z))

> (num-substr "ja" "jar jar")
2
> (num-substr "aa" "aaa")
2
> (num-substr "aa" "aaaa")
3
```

Задача 1.31 sum-divisors.lisp

Определить функцию, которая принимает целое положительное число и возвращает сумму делителей этого числа.

Решение 1.31.1

```
(defun sum-divisors (n)
  (loop for a from 1 to (truncate (/ n 2))
        when (zerop (rem n a))
        sum a))

> (sum-divisors 100)
117
> (sum-divisors 99)
57
```

Решение 1.31.2

```
(defun sum-divisors (n &aux (m (truncate (/ n 2))))
  (cond ((zerop m) 0)
```

```

      ((zerop (rem n m)) (+ m (sum-divisors n (1- m))))
      ((sum-divisors n (1- m)))))

> (sum-divisors 100)
117
> (sum-divisors 99)
57

```

Решение 1.31.3

```

(defun sum-divisors (n &aux (m (truncate (/ n 2))))
  (if (zerop m)
      0
      (if (zerop (rem n m))
          (+ m (sum-divisors n (1- m)))
          (sum-divisors n (1- m)))))

> (sum-divisors 100)
117
> (sum-divisors 99)
57

```

Задача 1.32 min-shuffle.lisp

Определить функцию, которая переставляет цифры целого числа так, чтобы получилось минимальное число.

Решение 1.32.1

```

(defun digitize (n &optional m)
  (cond ((zerop n) m)
        (t (multiple-value-bind (q r) (truncate n 10)
              (digitize q (cons r m))))))

(defun undigitize (w &optional (n 1))
  (if w (+ (* (car w) n) (undigitize (cdr w) (* n 10))) 0))

(defun min-shuffle (n)
  (undigitize (reverse (sort (digitize n) #'<)))))

> (min-shuffle 34534534)
33344455

```

Решение 1.32.2

```

(defun digitize (n &optional m)
  (if (zerop n)
      m
      (multiple-value-bind (q r) (truncate n 10)
        (digitize q (cons r m)))))

(defun undigitize (w &optional (n 1))

```



```
(if w (+ (* (car w) n) (undigitize (cdr w) (* n 10))) 0))

(defun min-shuffle (n)
  (undigitize (reverse (sort (digitize n) #'<))))

> (min-shuffle 34534534)
33344455
```

Решение 1.32.3

```
((defun min-shuffle (n)
  (parse-integer (sort (write-to-string n) #'char<)))

> (min-shuffle 34534534)
33344455
8
```

Задача 1.33 *capitalize-line.lisp*

If you are a professional writer - i.e., if someone else is getting paid to worry about how your words are formatted and printed - Emacs outshines all other editing software in approximately the same way that the noon-day sun does the stars. It is not just bigger and brighter; it simply makes everything else vanish.

Neal Stephenson «In the beginning... was the command line»

Создайте с помощью текстового редактора текстовый файл, каждая строка которого представляет одно предложение. Напишите программу, которая заменяет первую строчную букву предложения на заглавную.

Решение 1.33.1

```
old.txt:
abc abc
abc abc
```

```
(defun capitalize-line
  (&optional (in "d:/old.txt") (out "d:/new.txt") )
  (with-open-file (s in :direction :input)
    (with-open-file (r out :direction :output :if-exists :supersede)
      (do ((line (read-line s) nil :eof) (read-line s nil :eof)))
          ((eql line :eof))
          (if (zerop (length line))
              (format r "~%")
              (format r "~a~%"
                (string-capitalize line :start 0 :end 1)))))))
```

```
> (capitalize-line)
NIL

new.txt:
Abc abc
Abc abc
```

Задача 1.34 *deci-to-hex.lisp*

Определить функцию для перевода числа из десятичной системы в шестнадцатеричную.

Решение 1.34.1

```
(defun deci-to-hex (n) (format nil "~x" n))

> (deci-to-hex 9)
"9"
> (deci-to-hex 10)
"A"
> (deci-to-hex 24)
"18"
> (deci-to-hex 255)
"FF"
```

Решение 1.34.2

```
(defun deci-to-hex (n) (write-to-string n :base 16))

> (deci-to-hex 9)
"9"
> (deci-to-hex 10)
"A"
> (deci-to-hex 24)
"18"
> (deci-to-hex 255)
"FF"
```

Задача 1.35 *hex-to-bin.lisp*

Определить функцию для перевода чисел из шестнадцатеричной системы в двоичную. Ввод и вывод реализовать через файлы.

```
hex.in
A
18
```

Решение 1.35.1

```
(defun hexa-to-bin (&optional (in "d:/hex.in") (out "d:/bin.in"))
  (with-open-file (s in :direction :input)
```

```

(with-open-file (r out :direction :output :if-exists :supersede)
  (do ((line (read-line s nil :eof) (read-line s nil :eof)))
      ((eql line :eof))
      (if (zerop (length line))
          (format r "~%")
          (format r "~a~%"
                  (write-to-string
                    (parse-integer line :radix 16) :base 2))))))

> (hex-to-bin)
NIL

```

Решение 1.35.2

```

(defun hexa-to-bin (&optional (in "d:/hex.in") (out "d:/bin.out"))
  (with-open-file (s in :direction :input)
    (with-open-file (r out :direction :output :if-exists :supersede)
      (do ((line (read-line s nil :eof) (read-line s nil :eof)))
          ((eql line :eof))
          (if (zerop (length line))
              (format r "~%")
              (format r "~b~%" (parse-integer line :radix 16))))))

> (hex-to-bin)
NIL

bin.out
1010
11000

```

Задача 1.36 *ration.lisp*

Определить рациональное ли число, если да - то возвести его в квадрат, если нет, то вернуть это число.

Решение 1.36.1

```

(defun ration (n) (if (rationalp n) (expt n 2) n))

> (ration 7/3)
49/9
> (ration 1.7)
1.7

```

Решение 1.36.2

```

(defun ration (n)
  (if (rationalp n) (expt (float n) 2) n))

> (ration 7/3)
5.444444
> (ration 1.7)

```

1.7

Решение 1.36.3

```
(defun ration (n)
  (if (rationalp n) (float (expt n 2)) n))

> (ration 7/3)
5.4444447
> (ration 1.7)
1.7
```

Задача 1.37 extract-nums.lisp

Выбрать все числа из текста и записать их в новую строку.

Решение 1.37.1

```
(defun extract-nums (s)
  (string-trim "#\\(\\)"
    (write-to-string
      (delete-if-not
        #'numberp
        (read-from-string
          (concatenate 'string "(" s ")"))))))

> (extract-nums "a 1 b 2c 3")
"1 3"
```

Задача 1.38 count-divisors.lisp

Определить функцию, которая принимает целое положительное число и возвращает число делителей этого числа.

Решение 1.38.1

```
(defun count-divisors (n)
  (loop for a from 1 to (truncate (/ n 2))
        when (zerop (rem n a))
        count a))

> (count-divisors 100)
8
> (count-divisors 99)
5
```

Решение 1.38.2

```
(defun count-divisors (n &optional (m (truncate (/ n 2))))
  (cond ((zerop m) 0)
        ((zerop (rem n m))
```

```
(1+ (count-divisors n (1- m)))
((count-divisors n (1- m))))
```

```
> (count-divisors 100)
8
> (count-divisors 99)
5
```

Решение 1.38.3

```
(defun count-divisors (n &optional (m (truncate (/ n 2))))
  (if (plusp m)
      (if (zerop (rem n m))
          (1+ (count-divisors n (1- m)))
          (count-divisors n (1- m)))
      0))
```

```
> (count-divisors 100)
8
> (count-divisors 99)
5
```

Решение 1.38.4

```
(defun count-divisors (n)
  (labels ((cnt (n m)
            (cond ((zerop m) 0)
                  ((zerop (rem n m)) (1+ (cnt n (1- m))))
                  ((cnt n (1- m))))))
    (cnt n (truncate (/ n 2)))))
```

```
> (count-divisors 100)
8
> (count-divisors 99)
5
```

Задача 1.39 *adjust-subtitles.lisp*

Внести изменения в субтитры фильма "White Men Can't Jump (1992).txt".

```
1
00:00:37,200 --> 00:00:40,100
Presbyterians are called
"God's frozen people."

2
00:00:40,400 --> 00:00:42,400
Wouldn't swing if you hung them.
```

чтобы получилось:

```
11
```

```
00:00:44,200 --> 00:00:47,100
Presbyterians are called
"God's frozen people."
```

```
12
00:00:47,400 --> 00:00:49,400
Wouldn't swing if you hung them.
```

Решение 1.39.1

```
(defun adjust-subtitles (path s+ &optional (f+ 0) &aux (n 1))
  (with-open-file (s path :direction :input)
    (with-open-file
      (r "c:/nst/new.txt" :direction :output :if-exists :supersede)
      (do ((line (read-line s nil :eof) (read-line s nil :eof)))
          ((eql line :eof))
          (cond ((zerop (length line)) (format r "~%"))
                ((equal (format nil "~a" n) line)
                 (format r "~a~%" (+ n f+)) (incf n))
                ((char= (elt line 0) #\0)
                 (format r "~a~%" (at (at line s+ 6) s+ 23)))
                ((format r "~a~%" line)))))))

(defun at (line s+ pos
          &aux (z (+ (parse-integer
                     line :start pos :end (+ pos 2)) s+)))
  (cond ((> z 59) (re line (mod z 60) pos)
        (at line (floor z 60) (- pos 3)))
        ((re line z pos)))
  line)

(defun re (line u pos)
  (replace line (format nil "~2,'0d" u)
            :start1 pos :end1 (+ pos 2)))

> (adjust-subtitles "c:/nst/wmcj.txt" 7 10)
```

```
11
00:00:44,200 --> 00:00:47,100
Presbyterians are called
"God's frozen people."

12
00:00:47,400 --> 00:00:49,400
Wouldn't swing if you hung them.
```

Задача 1.40 decode-phone-n.lisp

Написать программу декодирования телефонного номера для АОН. По запросу АОНа АТС посылает телефонный номер, используя следующие правила:

- Если цифра повторяется менее 2 раз, то это помеха и она должна быть отброшена;

- Каждая значащая цифра повторяется минимум 2 раза;
 - Если в номере идут несколько цифр подряд, то для обозначения «такая же цифра как предыдущая» используется идущий 2 или более подряд раз знак #;
 Например, входящая строка 4434###552222311333661 соответствует номеру 4452136.

Решение 1.40.1

```
(defun decode-phone-n (s)
  (glue (squeeze (cleave s))))

(defun cleave (s)
  (map 'list #'string s))

(defun squeeze (w &optional ac q b z &aux (a (car w)) (d (cdr w)))
  (cond ((null w) (nreverse ac))
        ((or (string/= a b) (equal b q))
         (squeeze d ac b a z))
        ((and (equal a b) (equal b "#"))
         (squeeze d (push z ac) b a z))
        ((squeeze d (push a ac) b a a))))

(defun glue (v)
  (reduce
   #'(lambda (x &optional y)
       (concatenate 'string x y)) v :initial-value ""))

> (decode-phone-n "4434###552222311333661")
"4452136"
> (decode-phone-n "###4434###552222311333661")
"4452136"
> (decode-phone-n "5###4434###552222311333661")
"4452136"
```

Решение 1.40.2 * (decode-phone-n "22122")

```
(defun decode-phone-n (s)
  (glue (squeeze (cleave s))))

(defun cleave (s)
  (map 'list #'string s))

(defun squeeze (w &optional ac q b z &aux (a (car w)) (d (cdr w)))
  (cond ((null w) (nreverse ac))
        ((or (string/= a b) (equal b q) (equal a z))
         (squeeze d ac b a z))
        ((and (equal a b) (equal b "#"))
         (squeeze d (push z ac) b a z))
        ((squeeze d (push a ac) b a a))))

(defun glue (v)
  (reduce
```

```

    #'(lambda (x &optional y)
      (concatenate 'string x y)) v :initial-value ""))

> (decode-phone-n "5###4434###552222311333661")
"4452136"
> (decode-phone-n "22122")
"2"

Решение 1.40.3

(defun decode-phone-n (s)
  (glue (squeeze (cleave s))))

(defun cleave (s)
  (map 'list #'string s))

(defun squeeze (w &optional ac q b z &aux (a (car w)) (d (cdr w))
  (n (or (string/= a b) (equal b q) (equal a z)))
  (m (and (equal a b) (equal b "#"))))
  (if w (squeeze
    d
    (cond (n ac)
      (m (push z ac))
      (t (push a ac)))
    b
    a
    (if (or n m) z a))
    (nreverse ac)))

(defun glue (v)
  (reduce
    #'(lambda (x &optional y)
      (concatenate 'string x y)) v :initial-value ""))

> (decode-phone-n "5###4434###552222311333661")
"4452136"
> (decode-phone-n "22122")
"2"

```

Задача 1.41 reverse-digits.lisp

Определить функцию, которая выводит цифры числа в обратном порядке.

Решение 1.41.1

```

(defun reverse-digits (n &optional (m 0))
  (cond ((zerop n) m)
        (t (multiple-value-bind (q r) (truncate n 10)
              (reverse-digits q (+ (* m 10) r))))))

> (reverse-digits 1234)
4321

```


Решение 1.41.2

```
(defun reverse-digits (n)
  (labels ((rd (n m)
            (cond ((zerop n) m)
                  (t (multiple-value-bind (q r) (truncate n 10)
                      (rd q (+ (* m 10) r)))))))
    (rd n 0)))

> (reverse-digits 1234)
4321
```

Решение 1.41.3 (автор Chris Jester-Young codereview.stackexchange.com)

```
(defun reverse-digits (n)
  (labels ((next (n v)
            (if (zerop n) v
                (multiple-value-bind (q r)
                    (truncate n 10)
                    (next q (+ (* v 10) r))))))
    (next n 0)))

> (reverse-digits 34567)
76543
```

Решение 1.41.4

```
(defun reverse-digits (n)
  (parse-integer (reverse (write-to-string n))))

> (reverse-digits 1234)
4321
4
```

Задача 1.42 *remove-extra-spaces.lisp*

В заданной строке слова разделены несколькими пробелами. Удалить все лишние пробелы.

Решение 1.42.1

```
(defun remove-extra (s)
  (format nil "~{~s ~}"
    (read-from-string
     (concatenate
      'string "("
      (delete-if-not #'(lambda (x)
                          (or (alpha-char-p x)
                              (equal x #\space)
                              (equal x #\-)))
                      s) ")")))))
```

```
(defun remove-extra-spaces (s)
  (string-right-trim " " (remove-extra s)))
```

```
> (remove-extra-spaces "aa  bb")
"AA BB"
```

Решение 1.42.2

```
(defun remove-extra-spaces (w)
  (loop for c across w
        unless (and n (char= c (car (last n)) #\space))
        collect c into n
        finally (return (string-trim " " (concatenate 'string n)))))
```

```
> (remove-extra-spaces "aa  bb")
"AA BB"
```

Задача 1.43 swap-lets.lisp

Написать программу, которая в каждом слове предложения меняет местами первую и последнюю букву.

Решение 1.43.1

```
(defun cut (s)
  (mapcar #'swap
    (mapcar #'string (read-from-string
      (concatenate 'string "(" s ")")))))
```

```
(defun swap (s &aux (v (concatenate 'list s)))
  (if (cdr v) (nconc (last v) (butlast (cdr v)) `',(car v)) v))
```

```
(defun glue (w)
  (when w (concatenate 'string (car w) " " (glue (cdr w)))))
```

```
(defun swap-lets(s) (string-right-trim " " (glue (cut s))))
```

```
> (swap-lets "a ab abc")
"A BA CBA"
> (swap-lets "a ab #.(print 'security-hole) abc")
SECURITY-HOLE
"A BA EECURITY-HOLS CBA"
```

Решение 1.43.2

```
(defun cut (s)
  (mapcar #'swap (word (map 'list #'string s))))
```

```
(defun word (w &optional acc)
  (cond ((null w) (list (apply #'concatenate 'string (reverse acc))))
        ((equal (car w) " ")
         (equal (car w) " ")
```

```

      (cons (apply #'concatenate 'string (reverse acc))
            (word (cdr w) nil)))
    ((word (cdr w) (cons (car w) acc)))))

(defun swap (s &aux (v (concatenate 'list s)))
  (if (cdr v) (nconc (last v) (butlast (cdr v)) `((, (car v))) v))

(defun glue (w)
  (when w (concatenate 'string (car w) " " (glue (cdr w)))))

(defun swap-lets(s) (string-right-trim " " (glue (cut s))))

> (swap-lets "a ab abc")
"A BA CBA"
> (swap-lets "a ab #.(print 'security-hole) abc")
"a ba t.(prin# )security-hole' cba"

```

Задача 1.44 factorial-of.lisp

Определить, факториал какого числа равен передаваемому значению.

Решение 1.44.1

```

(defun factorial-of (m &optional (n 1) &aux (f (factorial n)))
  (cond ((> f m) nil)
        ((= f m) n)
        (t (factorial-of m (1+ n)))))

(defun factorial (n) (if (zerop n) 1 (* n (factorial (- n 1)))))

> (factorial-of 479001600)
12
> (factorial-of 479001601)
NIL

```

Решение 1.44.2

```

(defun factorial-of (m &optional (n 1) &aux (f (factorial n)))
  (when (<= f m) (if (= f m) n (factorial-of m (1+ n)))))

(defun factorial (n)
  (if (zerop n) 1 (* n (factorial (- n 1)))))

> (factorial-of 479001600)
12
> (factorial-of 479001601)
NIL

```

Решение 1.44.3

```

(defun factorial-of (m &optional (n 2) &aux (z (/ m n)))
  (cond ((= m 1) 1)

```

```

      ((= m 2) 2)
      ((= (1+ n) z) z)
      ((< (1+ n) z) (factorial-of Z (1+ n)))
      (t nil)))

> (factorial-of 479001600)
12
> (factorial-of 479001601)
NIL

```

Решение 1.44.4

```

(defun factorial-of (m &optional (n 2) &aux (z (/ m n)))
  (cond ((or (= m 2) (= m 1)) m)
        ((= (1+ n) z) z)
        ((< (1+ n) z) (factorial-of Z (1+ n)))
        (t nil)))

> (factorial-of 479001600)
12
> (factorial-of 479001601)
NIL

```

Решение 1.44.5

```

(defun factorial-of (m &optional (n 2) &aux (z (/ m n)))
  (cond ((= m 1) m)
        ((= z 1) n)
        ((< n z) (factorial-of Z (1+ n)))
        (nil)))

> (factorial-of 479001600)
12
> (factorial-of 479001601)
NIL

```

Решение 1.44.6

```

(defun factorial-of (m &optional (n 2) &aux (z (/ m n)))
  (cond ((= m 1) m)
        ((= z 1) n)
        ((< n z) (factorial-of Z (1+ n)))))

> (factorial-of 479001600)
12
> (factorial-of 479001601)
NIL

```

Решение 1.44.7

```

(defun factorial-of (m)
  (labels ((factor (m n &aux (z (/ m n)))
            (cond ((= m 1) m)

```

```

                ((= z 1) n)
                ((< n z) (factorial z (1+ n))))))
(factorial m 2)))

> (factorial-of 479001600)
12
> (factorial-of 479001601)
NIL

```

Решение 1.44.8 (автор - Catstail, www.cyberforum.ru)

```

(defun factorial-of (m &optional (z 1) (n 1))
  (cond ((= z m) n)
        ((> z m) nil)
        ((factorial-of m (* z (1+ n)) (1+ n)))))

> (factorial-of 479001600)
12
> (factorial-of 479001601)
NIL

```

Решение 1.44.9

```

(defun factorial-of (m &optional (z 1) (n 1))
  (when (<= z m) (if (= z m) n (factorial-of m (* z (1+ n)) (1+ n)))))

> (factorial-of 479001600)
12
> (factorial-of 479001601)
NIL

```

Решение 1.44.10

```

(defun factorial-of (m &optional (z 1) (n 1))
  (unless (> z m) (if (= z m)
                      n
                      (factorial-of m (* z (1+ n)) (1+ n)))))

> (factorial-of 479001600)
12
> (factorial-of 479001601)
NIL

```

Решение 1.44.11

```

(defun factorial-of (m)
  (labels ((factor (m z n)
             (unless (> z m)
               (if (= z m) n
                   (factor m (* z (1+ n)) (1+ n))))))
    (factor m 1 1)))

> (factorial-of 479001600)

```

```
12
> (factorial-of 479001601)
NIL
```

Задача 1.45 min-network-connections-length.lisp

Четыре города расположены по вершинам квадрата со стороной n . Найти какая из возможных дорожных сетей, связывающих города, имеет минимальную длину. Задача является частным случаем задачи Штейнера, состоящей в поиске минимального дерева Штейнера — кратчайшей сети, соединяющей заданный конечный набор точек плоскости. Свое название получила в честь берлинского геометра Якоба Штейнера (1796–1863).

Решение 1.45.1 * большая окружность

```
(defun min-network-connections-length (n)
  (* pi (sqrt (* (expt n 2) 2))))

> (min-network-connections-length 1)
4.442882862121663463L0
```

Решение 1.45.2 * зигзаг

```
(defun min-network-connections-length (n)
  (+ n n (sqrt (* (expt n 2) 2))))

> (min-network-connections-length 1)
3.4142137
```

Решение 1.45.3 * большая дуга

```
(defun min-network-connections-length (n)
  (* (* pi (sqrt (* (expt n 2) 2))) 0.75))

> (min-network-connections-length 1)
3.3321621465912475975L0
```

Решение 1.45.4 * три стороны

```
(defun min-network-connections-length (n)
  (* (* pi (sqrt (* (expt n 2) 2))) 0.75))

> (min-network-connections-length 1)
3.3321621465912475975L0
```

Решение 1.45.5 * две дуги

```
(defun min-network-connections-length (n)
  (* n pi))

> (min-network-connections-length 1)
3.1415926535897932385L0
```

Решение 1.45.6 * диагональный крест

```
(defun min-network-connections-length (n)
  (* (sqrt (* (expt n 2) 2)) 2))

> (min-network-connections-length 1)
2.828427
```

Решение 1.45.7 * 5 отрезков под углом 120 (пчелиные соты)

```
(defun min-network-connections-length (n) (float (* (/ n 2) 5)))

> (min-network-connections-length 1)
2.5
```

Задача 1.46 fibonacci.lisp

I think every one who aspires to be a complete programmer should learn Lisp or some variant of Lisp. It's like taking a class in classical acoustic guitar when learning to be a hard rock guitarist or taking a yoga class when learning to be a football player - you will learn to look at problems in a very different, compact way and I believe this will give you a competitive edge.

Eswar Priyadarshan

Написать функцию, возвращающую n-е число Фибоначчи.

Решение 1.46.1

```
(defun fibonacci (n)
  (if (> n 1) (+ (fibonacci (- n 1)) (fibonacci (- n 2))) n))

> (time (fibonacci 40))
Real time: 128.78036 sec.
Run time: 127.34361 sec.
Space: 84 Bytes
102334155
```

Решение 1.46.2

```
(defun fibonacci (z) (f z 0 1))

(defun f (z n m)
  (if (zerop z) n (f (1- z) (+ n m) n)))

> (time (fibonacci 40))
Real time: 0.0 sec.
```

Run time: 0.0 sec.
 Space: 48 Bytes
 102334155

Решение 1.46.3

```
(defun fibonacci (z)
  (labels ((f (z n m)
            (if (zerop z) n (f (1- z) (+ n m) n))))
    (f z 0 1)))

> (time (fibonacci 40))
Real time: 0.0 sec.
Run time: 0.0 sec.
Space: 48 Bytes
102334155
```

Решение 1.46.4

```
(defun fibonacci (z &optional (n 0) (m 1))
  (if (zerop z) n (fibonacci (1- z) (+ n m) n)))

> (time (fibonacci 40))
Real time: 0.0 sec.
Run time: 0.0 sec.
Space: 48 Bytes
102334155
```

Решение 1.46.5

```
(defun fibonacci (z)
  (labels ((fib (z n m)
            (if (zerop z) n
                (fib (1- z) (+ n m) n))))
    (fib z 0 1)))

> (time (fibonacci 40))
Real time: 0.0 sec.
Run time: 0.0 sec.
Space: 48 Bytes
102334155
```

Решение 1.46.6

```
(defun fibonacci (n)
  (loop repeat n
        for a = 0 then b
        and for b = 1 then (+ a b)
        finally (return b)))

> (time (fibonacci 40))
Real time: 0.0010001 sec.
Run time: 0.0 sec.
```


Space: 48 Bytes
102334155

Задача 1.47 *ctg.lisp*

Определить функцию для вычисления котангенса.

Решение 1.47.1

```
(defun ctg (a) (/ (cos a) (sin a)))
```

```
> (ctg 90)
-0.5012028
```

Решение 1.47.2

```
(defun ctg (a) (/ 1.0 (tan a)))
```

```
> (ctg 90)
-0.50120276
```

Решение 1.47.3

```
(defun ctg (a) (/ (tan a)))
```

```
> (ctg 90)
-0.50120276
```

Задача 1.48 *circle-area.lisp*

Известна длина окружности. Найти площадь круга, ограниченного этой окружностью.

Решение 1.48.1

```
(defun circle-area (c) (/ (* pi (expt (/ c pi) 2)) 4))
```

```
> (circle-area 10)
7.957747154594766788L0
```

Решение 1.48.2

```
(defun circle-area (c) (* pi (/ c pi 2) (/ c pi 2)))
```

```
> (circle-area 10)
7.957747154594766788L0
```

Решение 1.48.3

```
(defun circle-area (c) (* (/ c 2) (/ c pi 2)))
```

```
> (circle-area 10)
7.957747154594766788L0
```

Решение 1.48.4

```
(defun circle-area (c) (/ (* c c) 4 pi))
```

```
> (circle-area 10)
7.957747154594766788L0
```

Решение 1.48.5

```
(defun circle-area (c) (/ (expt c 2) 4 pi))
```

```
> (circle-area 10)
7.957747154594766788L0
```

Решение 1.48.6

```
(defun circle-area (c) (/ (* c c 1/4) pi))
```

```
> (circle-area 10)
7.957747154594766788L0
```

Решение 1.48.7

```
(defun circle-area (c) (/ (* pi (expt (/ c pi) 2)) 4))
```

```
> (circle-area 10)
7.957747154594766788L0
```

Задача 1.49 dot-lowercase.lisp

Дана строка знаков. Требуется определить количество сочетаний знаков точки и следующей буквы в нижнем регистре.

Решение 1.49.1

```
(defun dot-lowercase (s
                      &aux
                      (butlast-s (subseq s 0 (1- (length s))))
                      (cdr-s (subseq s 1)))
  (loop for a across butlast-s
        for b across cdr-s
        when (and (eq a #\.) (lower-case-p b))
        count a))

> (dot-lowercase "a.bc.defg.h.i.j.k.Lm.nopq.Rs.tuvwx.y.z")
10
```

Решение 1.49.2

```
(defun dot-lowercase (s)
  (loop for a across (subseq s 0 (1- (length s)))
        for b across (subseq s 1)
        when (and (eq a #\.) (lower-case-p b))
        count a))

> (dot-lowercase "a.bc.defg.h.i.j.k.Lm.nopq.Rs.tuvwx.y.z")
10
```

Решение 1.49.3

```
(defun dot-lowercase (s)
  (dot-low (coerce s 'list)))

(defun dot-low (w)
  (cond ((null (cdr w)) 0)
        ((and (eql (car w) #\.)
               (lower-case-p (cadr w)))
         (1+ (dot-low (cdr w))))
        ((dot-low (cdr w)))))

> (dot-lowercase "a.bc.defg.h.i.j.k.Lm.nopq.Rs.tuvwx.y.z")
10
```

Решение 1.49.4

```
(defun dot-lowercase (s)
  (dot-low (coerce s 'list)))

(defun dot-low (w)
  (if (cdr w)
      (if (and (eql (car w) #\.)
                (lower-case-p (cadr w)))
          (1+ (dot-low (cdr w)))
          (dot-low (cdr w)))
      0))

> (dot-lowercase "a.bc.defg.h.i.j.k.Lm.nopq.Rs.tuvwx.y.z")
10
```

Решение 1.49.5

```
(defun dot-lowercase (s)
  (dot-low (coerce s 'list)))

(defun dot-low (w)
  (if (cdr w)
      (+ (if (and (eql (car w) #\.)
                  (lower-case-p (cadr w)))
              1
              0)
         (dot-low (cdr w)))
      0))
```

```
> (dot-lowercase "a.bc.defg.h.i.j.k.Lm.nopq.Rs.tuvwx.y.z")
10
```

Задача 1.50 *sum-natural.lisp*

Написать рекурсивную функцию, считающую сумму натуральных чисел от 1 до n.

Решение 1.50.1

```
(defun sum-natural (n)
  (cond ((zerop n) 0)
        (t (+ n (sum-natural (1- n))))))

> (sum-natural 10)
55
```

Решение 1.50.2

```
(defun sum-natural (n)
  (if (zerop n)
      0
      (+ n (sum-natural (1- n)))))

> (sum-natural 10)
55
```

Задача 1.51 *sum-natural-even.lisp*

Написать рекурсивную функцию, считающую сумму четных натуральных чисел от 1 до n.

Решение 1.51.1

```
(defun sum-natural-even (n)
  (cond ((= n 1) 0)
        ((evenp n) (+ n (sum-natural-even (1- n))))
        (t (sum-natural-even (1- n)))))

> (sum-natural-even 10)
30
```

Решение 1.51.2

```
(defun sum-natural-even (n)
  (cond ((zerop n) 0)
        ((evenp n) (+ n (sum-natural-even (1- n))))
        ((sum-natural-even (1- n)))))

> (sum-natural-even 10)
```

30

Решение 1.51.3

```
(defun sum-natural-even (n)
  (if (plusp n)
      (if (evenp n)
          (+ n (sum-natural-even (1- n)))
          (sum-natural-even (1- n)))
      0))
```

```
> (sum-natural-even 10)
30
```

Решение 1.51.4

```
(defun sum-natural-even (n)
  (if (plusp n)
      (+ (if (evenp n) n 0) (sum-natural-even (1- n)))
      0))
```

```
> (sum-natural-even 10)
30
```

Задача 1.52 single-elimination-wins.lisp

Написать рекурсивную функцию, считающую минимальное число туров, разыгранных в соревнованиях по олимпийской системе или плей-офф (англ. playoff), при которой участник выбывает из турнира после первого же проигрыша, в зависимости от числа участников.

Решение 1.52.1

```
(defun single-elimination-wins (n &optional (m 0))
  (if (> n 1) (single-elimination-wins (/ n 2) (1+ m)) m))
```

```
> (single-elimination-wins 7)
3
> (single-elimination-wins 70)
7
> (single-elimination-wins 7000000000)
33
```

Решение 1.52.2

```
(defun single-elimination-wins (n)
  (labels ((single (n m)
             (if (> n 1) (single (/ n 2) (1+ m)) m)))
    (single n 0)))
```

```
> (single-elimination-wins 7)
3
```

```
> (single-elimination-wins 70)
7
> (single-elimination-wins 7000000000)
33
```

Задача 1.53 largest-prime-factor.lisp

The prime factors of 13195 are 5, 7, 13 and 29. What is the largest prime factor of the number 600851475143?

Простые делители числа 13195 – это 5, 7, 13 и 29. Каков самый большой делитель числа 600851475143, являющийся простым числом?

Решение 1.53.1

```
(defun primep (n)
  (loop for a from 2 to (isqrt n) never (zerop (mod n a))))

(defun largest-prime-factor (n &optional (m 2) &aux (z (/ n m)))
  (if (and (integerp z) (primep z))
      z
      (largest-prime-factor n (1+ m))))

> (time (largest-prime-factor 600851475143))
Real time: 250.87993 sec.
Run time: 240.59375 sec.
Space: 2274004336 Bytes
GC: 4337, GC time: 24.546875 sec.
6857
```

Решение 1.53.2

```
(defun primep (n)
  (loop for a from 2 to (isqrt n)
        never (zerop (mod n a))))

(defun largest-prime-factor (n)
  (labels ((largest (n m &aux (z (/ n m)))
            (if (and (integerp z) (primep z))
                z
                (largest n (1+ m)))))
    (largest n 2)))

> (time (largest-prime-factor 600851475143))
Real time: 268.7622 sec.
Run time: 239.60938 sec.
Space: 2274004336 Bytes
GC: 4338, GC time: 25.0 sec.
6857
```

Решение 1.53.3 (автор – rdt, www.cyberforum.ru)

```
(defun largest-prime-factor (n)
  (do ((a 2 (1+ a)))
      ((> (* a a) n) n)
      (if (zerop (mod n a))
          (return (largest-prime-factor (/ n a))))))

> (time (largest-prime-factor 600851475143))
Real time: 0.0019994 sec.
Run time: 0.0 sec.
Space: 7180 Bytes
6857
```

Решение 1.53.4

```
(defun largest-prime-factor (n)
  (loop for a from 2
        if (> (* a a) n) return n
        else if (zerop (mod n a))
        return (largest-prime-factor (/ n a))))

> (time (largest-prime-factor 600851475143))
Real time: 0.0014536 sec.
Run time: 0.0 sec.
Space: 7180 Bytes
6857
```

Задача 1.54 sum-square-difference.lisp

The sum of the squares of the first ten natural numbers is 385 The square of the sum of the first ten natural numbers is 3025. Hence the difference between the sum of the squares of the first ten natural numbers and the square of the sum is $3025 - 385 = 2640$. Find the difference between the sum of the squares of the first one hundred natural numbers and the square of the sum.

Сумма квадратов первых десяти натуральных чисел - 385. Квадрат суммы первых десяти натуральных чисел - 3025. Следовательно, разность между суммой квадратов и квадратом суммы первых десяти натуральных чисел составляет $3025 - 385 = 2640$. Найдите разность между суммой квадратов и квадратом суммы первых ста натуральных чисел.

Решение 1.54.1

```
(defun sum-square-difference
  (n &aux (w (loop for a from 1 to n collect a)))
  (- (expt (reduce #'+ w) 2)
      (reduce #'+ (mapcar #'(lambda (a) (expt a 2)) w))))

> (sum-square-difference 10)
2640
> (sum-square-difference 100)
25164150
```

Решение 1.54.2

```
(defun sum-square-difference (n)
  (- (expt (loop for a from 1 to n sum a) 2)
     (loop for a from 1 to n sum (expt a 2))))

> (sum-square-difference 10)
2640
> (sum-square-difference 100)
25164150
```

Решение 1.54.3

```
(defun sum-square-difference (n)
  (loop for a from 1 to n
        sum a into b
        sum (* a a) into c
        finally (return (- (* b b) c))))

> (sum-square-difference 10)
2640
> (sum-square-difference 100)
25164150
```

Задача 1.55 *count-word.lisp*

В заданной строке определить количество слов (слова могут разделяться одним или несколькими пробелами).

Решение 1.55.1

```
(defun count-word (s)
  (length (read-from-string (concatenate 'string "(" s ")"))))

> (count-word "aa aa aa")
3
```

Задача 1.56 *archive-string.lisp*

Есть строка, в ней надо заменить подстроки, которые состоят из 4х и больше идущих подряд одинаковых символов на знак ! + количество подряд идущих одинаковых символов + сам символ. Например:

```
> (f "aabbbaaabbcsdkAAAAbbbbbb")
"aabbbaaabbcsdk!4A!5b"
```

Решение 1.56.1

```
(defun ar (n ac)
  (concatenate 'string "!" (write-to-string n) (car ac)))
```



```
(defun chive (w &optional ac acc (n 0) &aux (a (car w)) (d (cdr w)))
  (cond ((null w)
        (reverse (if (> n 3) (cons (ar n ac) acc) (nconc ac acc))))
        ((equal a (car ac)) (chive d (cons a ac) acc (+ n 1)))
        ((> n 3) (chive d (list a) (cons (ar n ac) acc) 1))
        ((chive d (list a) (nconc ac acc) 1))))

(defun archive-string (s)
  (apply #'concatenate 'string (chive (map 'list #'string s))))

> (archive-string "aabbaaabbbcsdkAAAAbbbbbb")
"aabbaaabbbcsdk!4A!5b"
```

Задача 1.57 reap-reverse.lisp

It is not how much you do, but how much love
you put in the doing.

~

Напечатать в обратном порядке заданный во входном файле текст (за текстом следует точка).

Решение 1.57.1

```
d:/test.txt
abc de
1234.

(defun reap-reverse (path &aux u)
  (with-open-file (s path :direction :input)
    (do ((line (read-line s nil :eof) (read-line s nil :eof)))
        ((eql line :eof))
        (setf u (concatenate 'string u line))))
  (reverse u))

> (reap-reverse "d:/test.txt")
".4321 ed cba"
```

Задача 1.58 sum-square-reaper.lisp

Написать программу, которая находит сумму квадратов чисел, содержащихся во входном файле.

Решение 1.58.1

```
d:/test.txt
75 93 74 74 46 51 29 93 10 38 25 8 39 44 80 40 71 42 41 56

(defun sum-square-reaper (path &aux u)
  (with-open-file (s path :direction :input)
    (do ((line (read-line s nil :eof) (read-line s nil :eof)))
```

```

      ((eq1 line :eof))
      (setf u (concatenate 'string u line)))
    (sum-square (read-from-string
                  (concatenate
                    'string "(" u ")")))))

(defun sum-square (w) (loop for a in w sum (* a a)))

> (sum-square-reaper "d:/test.txt")
64745

```

Решение 1.58.2

```

d:/test.txt
56 30 25 10 12 18 29 98 75 17 40 0 82 7 71 92 35 15 39 47

(defun sum-square-reaper (path &aux u)
  (with-open-file (s path :direction :input)
    (do ((line (read-line s nil :eof) (read-line s nil :eof)))
        ((eq1 line :eof))
        (setf u (concatenate 'string u line)))
    (loop for a in (read-from-string
                    (concatenate
                      'string "(" u ")"))
          sum (* a a))))

> (sum-square-reaper "d:/test.txt")
48646

```

Задача 1.59 *deci.lisp*

Programming is not easy like Sunday morning,
it is silent poetry.

Waseem Latif

Определить функцию для перевода числа из q -ичной системы в десятичную ($q < 10$).

Решение 1.59.1

```

(defun deci (n q)
  (if (< n q)
      n
      (+ (* (deci (floor n 10) q) q)
          (mod n 10))))

> (deci 1010 2)
10
> (deci 2357 8)
1263

```

Решение 1.59.2

```
(defun deci (n q) (parse-integer (write-to-string n) :radix q))

> (deci 1010 2)
10
4
> (deci 2357 8)
1263
4
```

Задача 1.60 count-reversible-numbers.lisp

The craft of programming begins with empathy, not formatting or languages or tools or algorithms or data structures.

Kent Beck

Some positive integers n have the property that the sum $[n + \text{reverse}(n)]$ consists entirely of odd (decimal) digits. For instance, $36 + 63 = 99$ and $409 + 904 = 1313$. We will call such numbers reversible; so 36, 63, 409, and 904 are reversible. Leading zeros are not allowed in either n or $\text{reverse}(n)$. There are 120 reversible numbers below one-thousand. How many reversible numbers are there below one-billion?

Некоторые положительные целые числа n обладают следующим свойством: сумма $[n + \text{перевернутое}(n)]$ состоит исключительно из нечетных (десятичных) цифр. К примеру, $36 + 63 = 99$ и $409 + 904 = 1313$. Такие числа будем называть переворачиваемыми. Таким образом, 36, 63, 409, и 904 являются переворачиваемыми. Ни n , ни перевернутое(n) не могут начинаться с нулей. В пределах одной тысячи существует 120 переворачиваемых чисел. Сколько переворачиваемых чисел существует в пределах одного миллиарда?

Решение 1.60.1

```
(defun count-reversible-numbers (d)
  (loop for a from 2 to d
        if (evenp a)
          sum (* 20 (expt 30 (1- (floor a 2))))
        else if (= 3 (mod a 4))
          sum (* 5 20 (expt (* 20 25) (floor a 4)))))

> (count-reversible-numbers 9)
608720
```

Решение 1.60.2

```
(defun count-reversible-numbers (d)
  (loop for a from 2 to d
        if (evenp a)
```

```

sum (* 20 (expt 30 (1- (floor a 2))))
else if (= 3 (mod a 4))
sum (* 5 20 (expt (* 20 25) (floor a 4))))

> (count-reversible-numbers 9)
608720

```

Задача 1.61 reverse-sentence.lisp

The only way to learn a new programming language is by writing programs in it.

Kernighan and Ritchie

Given an input string, reverse the string word by word.

Определить функцию, меняющую порядок слов в предложении на обратный.

Решение 1.61.1

```

(defun cleave (s)
  (mapcar #'string (read-from-string
                    (concatenate 'string "(" s ")"))))

(defun reverse-sentence (s)
  (string-right-trim " " (format nil "~{~a ~}"
                                (reverse (cleave s)))))

> (reverse-sentence "ab cd")
"CD AB"

```

Решение 1.61.2

```

(defun cleave (s)
  (mapcar #'string (read-from-string
                    (concatenate 'string "(" s ")"))))

(defun reverse-sentence (s)
  (string-right-trim '(#\Space)
                    (format nil "~{~a ~}"
                            (reverse (cleave s)))))

> (reverse-sentence "ab cd")

"CD AB"

```

Решение 1.61.3

```

(defun cleave (s)
  (mapcar #'string (read-from-string
                    (concatenate
                     'string "("

```

```

(delete-if-not #'(lambda (x)
                  (or (alpha-char-p x)
                      (equal x #\space)
                      (equal x #\-)
                      (equal x #\_))))
s) "))))))

(defun reverse-sentence (s)
  (string-right-trim '(#\Space)
    (format nil "~{~a ~}"
      (reverse (cleave s)))))

> (reverse-sentence "ab, cd; ef")
"EF CD AB"

```

Решение 1.61.4

```

(defun cleave (s)
  (mapcar #'string (read-from-string
    (concatenate
      'string "("
      (delete-if #'(lambda (x)
                    (or (equal x #\()
                        (equal x #\))
                        (equal x #\,)
                        (equal x #\' )
                        (equal x #\' )
                        (equal x #\| )
                        (equal x #\| )
                        (equal x #\: )
                        (equal x #\\ )
                        (equal x #\; ))))
      s) "))))))

(defun reverse-sentence (s)
  (string-right-trim '(#\Space)
    (format nil "~{~a ~}"
      (reverse (cleave s)))))

> (reverse-sentence "a, b; c@ d# e$ f% g^ h& i* j_ k+ l/ m[ n] o{ p}
q< r> s. t? u( v) z, z' z` z| z: z\ z;")
"Z Z Z Z Z Z V U T? S. R> Q< P} O{ N] M[ L/ K+ J_ I* H& G^ F% E$ D#
C@ B A"

```

Задача 1.62 sum-evenn.lisp

Определить функцию, возвращающую значение суммы ряда целых четных чисел от 2 до n.

Решение 1.62.1

```

(defun sum-evenn (n)
  (reduce #'+ (loop for a from 2 to n when (evenp a) collect a)))

```

```
> (sum-evenn 11)
30
```

Задача 1.63 *replace-ing-ed.lisp*

Определить функцию, заменяющую суффикс "ing" на "ed" всех словах предложения.

Решение 1.63.1

```
(defun replace-ie (s a b &aux (n (search a s)))
  (if n (replace-ie (replace s b :start1 n) a b) s))

(defun replace-inged (s)
  (replace-ie
   (replace-ie
    (replace-ie s
                "ing."
                "ed. ")
    "ing, "
    "ed, ")
  "ing "
  "ed "))

> (replace-inged "inglewood laughing, and dancing.")
"inglewood laughed, and danced. "
```

Задача 1.64 *product-even.lisp*

In Lisp the data is list, program is list, basically you are writing lists, inside lists inside lists.

Ruchir Hajela

Дано число n . Определить функцию, возвращающую значение произведения ряда целых нечётных чисел от 1 до n .

Решение 1.64.1

```
(defun product-even (n)
  (cond ((= n 1) 1)
        ((evenp n) (* n (product-even (1- n)))))
        ((product-even (1- n)))))

> (product-even 5)
8
```

Решение 1.64.2

```
(defun product-even (n)
```

```
(if (= n 1) 1 (* (if (evenp n) n 1) (product-even (1- n)))))

> (product-even 5)
8
```

Решение 1.64.3

```
(defun product-even (n)
  (reduce #'* (loop for a from 1 to n when (evenp a) collect a)))

> (product-even 5)
8
```

Решение 1.64.4

```
(defun product (n m p)
  (if (> m n) p (product n (+ m 2) (* p k))))

(defun product-even (n) (product n 2 1))

> (product-even 5)
8
```

Задача 1.65 *n!+n²⁰.lisp*

Вычислить $n!+n^{20}$, где $n > 20$.

Решение 1.65.1

```
(defun n!+n^20 (n)
  (+ (factorial n) (expt n 20)))

(defun factorial (n)
  (if (zerop n) 1 (* (factorial (1- n)) n)))

> (n!+n^20 53)
4274883284060025564298013753389399680275971079215018641463848989276401
```

Задача 1.66 *last-number.lisp*

Определить функцию, возвращающую последнюю цифру в строке.

Решение 1.66.1

```
(defun last-number (s)
  (find-if
   #'numberp
   (mapcar
    #'read-from-string
    (map 'list #'string s)
    :from-end t)))
```

```
> (last-number "alb2c")
2
```

Решение 1.66.2

```
(defun last-number (s)
  (find-if
   #'numberp
   (loop for c across s
         collect (read-from-string (string c)))
   :from-end t))
```

```
> (last-number "alb2c")
2
```

Задача 1.67 eeeq.lisp

Дано натуральное четырёхзначное число, т.е. в диапазоне от 1000 до 9999. Определить функцию, проверяющую, что три из четырёх цифр совпадают.

Решение 1.67.1

```
(defun int-digits (n)
  (loop for c across (write-to-string n)
        collect (digit-char-p c)))
```

```
(defun eeeq (n)
  (let* ((m (int-digits n))
         (a (car m))
         (b (cadr m))
         (c (caddr m))
         (d (cadddr m)))
    (or (= a b c)
        (= a b d)
        (= b c d)
        (= a c d))))
```

```
> (eeeq 1112)
T
> (eeeq 1122)
NIL
```

Решение 1.67.2

```
(defun int-digits (n)
  (mapcar #'digit-char-p (coerce (write-to-string n) 'list)))

(defun eeeq (n)
  &aux (m (int-digits n))
        (a (car m))
```



```

        (b (cadr m))
        (c (caddr m))
        (d (cadddr m)))
(or (= a b c)
    (= a b d)
    (= b c d)
    (= a c d)))

> (eeeq 1112)
T
> (eeeq 1122)
NIL

```

Решение 1.67.3

```

(defun int-digits (n)
  (mapcar #'digit-char-p (concatenate 'list (write-to-string n))))

(defun eeeq (n
             &aux (m (int-digits n))
                  (a (car m))
                  (b (cadr m))
                  (c (caddr m))
                  (d (cadddr m)))
  (or (= a b c)
      (= a b d)
      (= b c d)
      (= a c d)))

> (eeeq 1112)
T
> (eeeq 1122)
NIL

```

Решение 1.67.4

```

(defun int-digits (n)
  (mapcar #'digit-char-p (map 'list #'identity (write-to-string n))))

(defun eeeq (n
             &aux (m (int-digits n))
                  (a (car m))
                  (b (cadr m))
                  (c (caddr m))
                  (d (cadddr m)))
  (or (= a b c)
      (= a b d)
      (= b c d)
      (= a c d)))

> (eeeq 1112)
T
> (eeeq 1122)

```

NIL

Решение 1.67.5

```
(defun int-digits (n)
  (mapcar #'parse-integer (map 'list #'string (write-to-string n))))

(defun eeeq (n
            &aux (m (int-digits n))
                 (a (car m))
                 (b (cadr m))
                 (c (caddr m))
                 (d (caddr m)))
  (or (= a b c)
      (= a b d)
      (= b c d)
      (= a c d)))

> (eeeq 1112)
T
> (eeeq 1122)
NIL
```

Решение 1.67.6

```
(defun int-digits (n)
  (mapcar #'parse-integer (map 'list #'string (format nil "~D" n))))

(defun eeeq (n
            &aux (m (int-digits n))
                 (a (car m))
                 (b (cadr m))
                 (c (caddr m))
                 (d (caddr m)))
  (or (= a b c)
      (= a b d)
      (= b c d)
      (= a c d)))

> (eeeq 1112)
T
> (eeeq 1122)
NIL
```

Решение 1.67.7

```
(defun int-digits (n) (map 'list #'digit-char-p (prin1-to-string n)))

(defun eeeq (n)
  (let* ((m (int-digits n))
        (a (car m))
        (b (cadr m))
        (c (caddr m)))
```

```

      (d (caddr m)))
    (or (= a b c)
        (= a b d)
        (= b c d)
        (= a c d))))

```

```

> (eeeq 1112)
T
> (eeeq 1122)
NIL

```

Решение 1.67.8

```

(defun int-digits (n &optional ac)
  (if (zerop n) ac
      (int-digits (truncate n 10) (cons (rem n 10) ac))))

(defun eeeq (n
             &aux (m (int-digits n))
                  (a (car m))
                  (b (cadr m))
                  (c (caddr m))
                  (d (caddr m)))
  (or (= a b c)
      (= a b d)
      (= b c d)
      (= a c d)))

```

```

> (eeeq 1112)
T
> (eeeq 1122)
NIL

```

Решение 1.67.9

```

(defun int-digits (n)
  (labels ((id (n ac)
             (if (zerop n) ac
                 (id (truncate n 10)
                     (cons (rem n 10) ac)))))
    (id n nil)))

(defun eeeq (n
             &aux (m (int-digits n))
                  (a (car m))
                  (b (cadr m))
                  (c (caddr m))
                  (d (caddr m)))
  (or (= a b c)
      (= a b d)
      (= b c d)
      (= a c d)))

```

```
> (eeeq 1112)
T
> (eeeq 1122)
NIL
```

Решение 1.67.10

```
(defun int-digits (n)
  (loop with r do (setf (values n r) (truncate n 10))
    collect r into ac until (zerop n)
    finally (return (nreverse ac))))

(defun eeeq (n
             &aux (m (int-digits n))
                  (a (car m))
                  (b (cadr m))
                  (c (caddr m))
                  (d (caddddr m)))
  (or (= a b c)
      (= a b d)
      (= b c d)
      (= a c d)))

> (eeeq 1112)
T
> (eeeq 1122)
NIL
```

Задача 1.68 min-lets.lisp

В заданной строке определить самое короткое слово.

Решение 1.68.1

```
(defun min-lets (s)
  (caar (sort (mapcar #'(lambda (a) (list a (length a)))
                    (mapcar #'string (read-from-string
                                       (concatenate 'string "(" s ")"))))
             #'< :key #'cadr)))

> (min-lets "a ab abc")
"A"
```

Решение 1.68.2

```
(defun min-lets (s)
  (caar
   (sort
    (mapcar
     #'(lambda (a) (list a (length a)))
     (mapcar
      #'string
```

```

(read-from-string
 (concatenate
  'string "("
  (remove-if-not #'(lambda (x)
                      (or (alpha-char-p x)
                          (equal x #\space)
                          (equal x #\-)))
    s) ")"))))
#'< :key #'cadr)))

> (min-lets "a ab abc")
"A"
> (min-lets "(asdf jlkj) lk")
"LK"
> (min-lets "jsadf .")
"JSADF"
> (min-lets "#.(print (user-homedir-pathname))")
"PRINT"

```

Решение 1.68.3

```

(defun min-lets (s)
  (caar
   (sort
    (mapcar
     #'(lambda (a) (list a (length a)))
     (mapcar
      #'string
      (read-from-string
       (concatenate
        'string "("
        (loop for c across s
              if (or (alpha-char-p c)
                    (equal c #\-))
                collect c
                else collect #\space) ")"))))
     #'< :key #'cadr)))

> (min-lets "a ab abc")
"A"
> (min-lets "(asdf jlkj) lk")
"LK"
> (min-lets "jsadf .")
"JSADF"
> (min-lets "#.(print (user-homedir-pathname))")
"PRINT"
> (min-lets "foo+bar")
"FOO"
> (min-lets "foo+b")
"B"

```

Решение 1.68.4

```

(defun min-lets (s)
  (caar
   (sort
    (mapcar
     #'(lambda (a) (list a (length a)))
     (mapcar
      #'string
      (read-from-string
       (concatenate
        'string "("
        (substitute-if-not
         #\space
         #'(lambda (a) (or (alpha-char-p a)
                           (equal a #\-)))
         s) ")"))))
    #'< :key #'cadr)))

> (min-lets "a ab abc")
"A"
> (min-lets "(asdf jlkj) lk")
"LK"
> (min-lets "jsadf .")
"JSADF"
> (min-lets "#.(print (user-homedir-pathname))")
"PRINT"
> (min-lets "foo+bar")
"FOO"
> (min-lets "foo+b")

```

Решение 1.68.5

```

(defun check (a) (or (alpha-char-p a) (equal a #\-)))

(defun aim (b r)
  (if (or (not b) (and r (< (length r) (length b)))) r b))

(defun min-lets (s
  &optional
  b
  &aux
  (n (position-if #'check s))
  (m (when n (position-if-not #'check s :start n)))
  (r (when n (if m (subseq s n m) (subseq s n)))))
  (cond ((not n) b)
        ((not m) (aim b r))
        ((min-lets (subseq s m) (aim b r)))))

> (min-lets "a ab abc")
"A"
> (min-lets "(asdf jlkj) lk")
"LK"
> (min-lets "jsadf .")
"JSADF"

```

```
> (min-lets "#.(print (user-homedir-pathname))")
"PRINT"
> (min-lets "foo+bar")
"FOO"
> (min-lets "foo+b")
```

Задача 1.69 *bis-sexтус.lisp*

Определите функцию, проверяющую, что данный год является високосным.

Високосный по юлианскому календарю (он же «старый стиль», по которому отмечается «Старый Новый год») тот, который кратен 4; по григорианскому календарю (начиная с 1582 года страны стали переходить на него) тот, который кратен 400 или тот, который кратен 4, но в тоже время не кратен 100, «кратен» означает, что число делится без остатка.

Решение 1.69.1

```
(defun bis-sexтус (n)
  (destructuring-bind (z400 z100 z4)
    (mapcar #'(lambda (a) (zerop (mod n a))) '(400 100 4))
    (or z400 (unless z100 z4))))

> (bis-sexтус 2015)
NIL
> (bis-sexтус 2016)
T
```

Решение 1.69.2

```
(defun bis-sexтус (n)
  (or (zerop (rem n 400))
      (and (zerop (rem n 4))
            (not (zerop (rem n 100))))))

> (bis-sexтус 2015)
NIL
> (bis-sexтус 2016)
T
```

Решение 1.69.3

```
(defun bis-sexтус (n)
  (or (zerop (rem n 400))
      (unless (zerop (rem n 100)) (zerop (rem n 4)))))

> (bis-sexтус 2015)
NIL
> (bis-sexтус 2016)
T
```

Задача 1.70 newton-2.lisp

Programming languages have two boundaries such as C and Lisp. Everything else is in-between. C is the language which you can further dig into the hardware by pushing limits of performance. Lisp is the language which you can further dig into abstractions by using macros/code as data. It makes you to think ideas freely without any boundaries.

Tamil Selvan

Определить функцию, возвращающую корень числа с точностью до сотых.

Решение 1.70.1

```
(defun newton-2 (x &optional (e .001) (z 1))
  (if (> (abs (- x (* z z))) e)
      (newton-2 x e (* (+ z (/ x z)) .5))
      (format t "~,2f" z)))
```

```
> (newton-2 2)
1.41
NIL
```

Решение 1.70.2

```
(defun newton-2 (x &optional (e .001) (z 1))
  (if (> (abs (- x (* z z))) e)
      (newton-2 x e (* (+ z (/ x z)) .5))
      (format t "~$" z)))
```

```
> (newton-2 2)
1.41
NIL
```

Задача 1.71 newton-3.lisp

Определить функцию, возвращающую корень числа с точностью до тысячных.

Решение 1.71.1

```
(defun newton-3 (x &optional (e .0001) (z 1))
  (if (> (abs (- x (* z z))) e)
      (newton-3 x e (* (+ z (/ x z)) .5))
      (format t "~,3f" z)))
```

```
> (newton-3 2)
1.414
NIL
```


Задача 1.72 +digit.lisp

The standard practice in Lisp and related languages is to define abstractions relevant to the problem domain and keep re-using them (almost like a Domain Specific language with a FULL language behind it). Using a combination of macros and features that have evolved over the last 50 years, creating these abstractions is second-nature to experienced Lispsers. As a result, they can do far far more in a finite amount of time than non-lispsers. The way it manifests itself is that a single line of Lisp code can end up being equivalent to hundreds of lines of hand-written code. The industry average for defects per line of code is about 15-50 per KLOC. This is generally independent of programming language because this is human error. A programming language that replaces 100's of lines with a single line of code automatically reduces the defects in software by a proportional amount. Since most programmers spend most of their time tracing and fixing defects (even during development), it automatically makes the programmer that much more productive. For the longest time, this methodology of programming was unmatched in terms of its productivity. It was also VERY hard for 80% of programmers to understand and be effective at. This is also why you don't see that many people using Lisp.

Anurag Mendhekar

Определите функцию, вычисляющую сумму цифр натурального числа.

Решение 1.72.1

```
(defun +digit (n)
  (if (< n 10) n (+ (rem n 10) (+digit (truncate n 10)))))

> (+digit 1234)
10
```

Задача 1.73 ++digit.lisp

Найти сумму цифр целого числа без учета знака.

Решение 1.73.1

```
(defun ++digit (n)
  (cond ((minusp n) (++digit (abs n)))
        ((< n 10) n)
        ((+ (rem n 10) (++digit (truncate n 10))))))

> (++digit 12345)
15
> (++digit -12345)
15
```

Решение 1.73.2

```
(defun ++digit (n) (+d (abs n)))

(defun +d (n)
  (if (>= n 10)
      (+ (rem n 10) (+d (truncate n 10)))
      n))

> (++digit 12345)
15
> (++digit -12345)
15
```

Решение 1.73.3

```
(defun ++digit (n) (+d (abs n)))

(defun +d (n)
  (if (< n 10) n (+ (rem n 10) (+d (truncate n 10)))))

> (++digit 12345)
15
> (++digit -12345)
15
```

Задача 1.74 punctuation-marks.lisp

Дано предложение в виде строки. Определить функцию, которая возвращает копию этого предложения строки, в которой все знаки препинания кроме последнего заменены на заданный знак препинания.

Решение 1.74.1

```
(defun punctuation-marks (s
  p
  &aux
    (v '(#\. #\, #\: #\; #\! №\? #\(-))
    (n (1- (length s)))
    (u (subseq s 0 n))
    (z (subseq s n)))
  (coerce (nconc (loop for a across u
```

```

        if (member a v) collect p
        else collect a)
      `,(char z 0)))
    'string))

```

```

> (punctuation-marks "abra: ca, dabra!" #\?)
"abra? ca? dabra!"

```

Решение 1.74.2

```

(defun punctuation-marks (s
  p
  &aux
    (v '(#\. #\, #\: #\; #\! #\? #\-))
    (n (position-if
      #'(lambda (a) (member a v))
      s
      :from-end t))
    (u (subseq s 0 n))
    (z (subseq s n)))
  (concatenate 'string
    (substitute-if
      p
      #'(lambda (a) (member a v))
      u)
    z))

```

```

> (punctuation-marks "abra: ca, dabra!" #\?)
"abra? ca? dabra!"
> (punctuation-marks "abra: ca, da! bra" #\?)
"abra? ca? da! bra"

```

Задача 1.75 reverse-parts.lisp

Дана строка, состоящая из четного количества символов. Определить функцию, возвращающую строку, составленную из символов первой половины заданной строки, записанных в обратном порядке, после которых идут символы второй половины заданной строки, также записанные в обратном порядке.

Решение 1.75.1

```

(defun reverse-parts (s &aux (m (/ (length s) 2)))
  (concatenate 'string
    (reverse (subseq s 0 m))
    (reverse (subseq s m))))

> (reverse-parts "abcdef")
"cbafed"

```

Задача 1.76 *fahrenheit-celsius.lisp*

Определить функцию для перевода температуры из шкалы Фаренгейта в шкалу Цельсия.

Историческая справка: Градус Фаренгейта — единица измерения температуры. Назван в честь немецкого учёного Габриеля Фаренгейта, предложившего в 1724 году шкалу для измерения температуры.

На шкале Фаренгейта точка таяния льда равна $+32^{\circ}\text{F}$, а точка кипения воды $+212^{\circ}\text{F}$. Диапазон $0\dots+100^{\circ}\text{F}$ по шкале Фаренгейта примерно соответствует диапазону $-18\dots+38^{\circ}\text{C}$ по шкале Цельсия.

Шкала Фаренгейта сейчас повсеместно применяется только в США.

Приблизительное преобразование градусов по Фаренгейту в градусы Цельсия: вычтите 30 и полученное число умножьте на два. Эта формула дает ответ правильный в пределах $\pm 2,8^{\circ}\text{C}$ от 1°F до 100°F и точно при 50°F .

Для точного перевода температуры из шкалы Фаренгейта в шкалу Цельсия нужно от исходного числа отнять 32 и умножить результат на $5/9$.

Решение 1.76.1

```
(defun fahrenheit-celsius (n) (float (* (- n 32) 5/9)))

> (fahrenheit-celsius 100)
37.77778
> (fahrenheit-celsius 0)
-17.777779
> (fahrenheit-celsius -40)
-40.0
```

Задача 1.77 *celsius-fahrenheit.lisp*

Определить функцию для перевода температуры из шкалы Цельсия в шкалу Фаренгейта.

Историческая справка: В 1665 году голландский физик Христиан Гюйгенс вместе с английским физиком Робертом Гуком впервые предложили использовать в качестве отсчётных точек температурной шкалы точки таяния льда и кипения воды.

В 1742 году шведский астроном, геолог и метеоролог Андерс Цельсий (1701–1744) на основе этой идеи разработал новую температурную шкалу.

Первоначально в ней за ноль была принята точка кипения воды, а за 100°C — температура замерзания воды (точка плавления льда).

Позже, уже после смерти Цельсия, его современники и соотечественники ботаник Карл Линней и астроном Мортен Штремер использовали эту шкалу в

перевернутом виде (за 0 °C стали принимать температуру таяния льда, а за 100 °C — кипения воды).

Приблизительное преобразование градусов по Цельсию в градусы Фаренгейта: умножьте на два и добавьте 30. Эта формула дает ответ правильный в пределах $\pm 2,8$ °C от 1 °F до 100 °F и точно при 50 °F.

Для точного перевода температуры из шкалы Цельсия в шкалу Фаренгейта нужно умножить исходное число на 9/5 и прибавить 32.

Решение 1.77.1

```
(defun celsius-fahrenheit (n) (float (+ (* n 9/5) 32)))

> (celsius-fahrenheit 36.6)
97.88
> (celsius-fahrenheit 20)
68.0
> (celsius-fahrenheit 0)
32.0
> (celsius-fahrenheit -10)
14.0
```

Задача 1.78 calculate.lisp

On the surface, Lisp is a nice, simple functional programming language. There's almost no syntax, and all the pieces fit together in logical ways. If you dig a little deeper, read SICP, and write a metacircular evaluator, you discover two things: One, the whole interpreter (given just a few primitives) is just barely a page of code, and two, the relationship between code and data allows for elegant programming techniques. Once you've fully absorbed this, it feels like other languages are set in stone when they only allow you to say a few things. Lisp can build any abstraction at all if you can define syntax and semantics for it.

Josh Lee

Написать функцию, которая принимает ввод двух чисел и операции между ними. Выход из функция командой quit.

Решение 1.78.1

```
(defun calculate (&aux cmd)
  (loop (print "Enter: Number Operator Number")
        (setq cmd (read-from-string
                     (concatenate 'string "(" (read-line) ")"))))
```

```

      (when (eql (car cmd) 'quit) (return 'ok))
      (print (eval `',(cadr cmd) ,(car cmd) ,(caddr cmd))))))

> (calculate)
"Enter: Number Operator Number" 2 + 3
5
"Enter: Number Operator Number" quit
OK

```

Решение 1.78.2

```

(defun calculate (&aux w)
  (loop (print '>)
        (setq w (read-from-string
                  (concatenate 'string "(" (read-line) ")"))))
        (when (eql (car w) 'quit) (return 'ok))
        (print (eval `',(cadr w) ,(car w) ,(caddr w))))))

> (calculate)
> 2 + 3
5
> quit
OK

```

Задача 1.79 *sum.lisp*

Определить функцию, вычисляющую сумму положительных чётных чисел меньше n .

Решение 1.79.1

```

(defun sum (n)
  (loop for a from 1 below n when (evenp a) sum a))

> (sum 9)
20

```

Задача 1.80 *num-digit.lisp*

Определить функцию, возвращающую количество цифр натурального числа.

Решение 1.80.1

```

(defun num-digit (n)
  (if (> n 9) (1+ (num-digit (floor n 10))) 1))

> (num-digit 1017)
4

```

Решение 1.80.2

```
(defun num-digit (n)
  (length (write-to-string n)))

> (num-digit 1017)
4
```

Задача 1.81 cnt.lisp

A Lisp program tends to provide a much clearer mapping between your ideas about how the program works and the code you actually write.

Peter Seibel

Определить в заданной строке количество знаков арифметических действий.

Решение 1.81.1

```
(defun cnt (w s) (loop for a in w sum (count a s)))

> (cnt '(#\+ #\ - #\* #\/) "a+ b- c* d/")
4
```

Решение 1.81.2

```
(defun cnt (s) (loop for a in '(#\+ #\ - #\* #\/) sum (count a s)))

> (cnt "a+ b- c* d/")
4
```

Задача 1.82 srt.lisp

Слова в предложении разделены пробелами. Упорядочить слова в соответствии с ростом их длины.

Решение 1.82.1

```
(defun srt (s)
  (string-right-trim
   " "
   (apply
    #'concatenate
    'string
    (mapcar
     #'(lambda (a) (concatenate 'string a " "))
     (mapcar
      #'car
      (sort
       (mapcar
        #'(lambda (a) (list a (length a)))
```

```

(mapcar
 #'string
 (read-from-string
  (concatenate
   'string
    "(" s ")"))))
 #'< :key #'cadr))))))

> (srt "a abc ab")
"A AB ABC"

```

Задача 1.83 *reap-expt.lisp*

Then there is lisp... Blissful ignorance on one side (Lots of Irritating Silly Parenthesis) and Nirvana on the other. Hell lives in between. The code you write in any language will never be the same once you've seen the light in lisp.

Daniel Billotte

Определить функцию, которая находит заданную во входном файле степень числа, в первой строке – основание степени, во второй строке – показатель степени.

Решение 1.83.1

```

(defun reap-expt (path)
  (with-open-file (s path :direction :input)
    (expt (read-from-string (read-line s))
          (read-from-string (read-line s)))))

```

```

d:/test.txt
10
3
> (reap-expt "d:/test.txt")
1000

```

```

d:/test.txt
10
1/3
> (reap-expt "d:/test.txt")
2.1544347

```

```

d:/test.txt
10
1.5
> (reap-expt "d:/test.txt")
31.622776

```


Задача 1.84 maxxx.lisp

You will never understand the most fundamental design decision in software if you don't understand both C and Lisp.

Joshua Bowles

Определить функцию, которая находит наибольшее число, получаемое левыми сдвигами цифр натурального числа.

Решение 1.84.1

```
(defun maxxxx (n &aux (d (shift (digits n))))
  (maxx n d (length d)))

(defun maxx (n d c &aux (m (int d)))
  (cond ((= c 1) (max n m))
        ((< n m) (maxx m (shift d) (1- c)))
        ((maxx n (shift d) (1- c)))))

(defun shift (w) (append (cdr w) (list (car w))))

(defun digits (n &optional acc)
  (if (zerop n) acc
      (digits (truncate n 10) (cons (rem n 10) acc))))

(defun int (w &optional (acc 0))
  (if w (int (cdr w) (+ (car w) (* acc 10))) acc))

> (maxxxx 1730382)
8217303
```

Решение 1.84.2

```
(defun maxxxx (n &aux (d (shift (digits n))))
  (maxx n d (length d)))

(defun maxx (n d c &aux (m (int d)))
  (if (= c 1) (max n m)
      (maxx (if (< n m) m n) (shift d) (1- c))))

(defun shift (w) (append (cdr w) (list (car w))))

(defun digits (n &optional acc)
  (if (zerop n) acc
      (digits (truncate n 10) (cons (rem n 10) acc))))

(defun int (w &optional (acc 0))
  (if w (int (cdr w) (+ (car w) (* acc 10))) acc))

> (maxxxx 1730382)
```

8217303

Задача 1.85 max-divisor.lisp

I like Emacs for the same reason I like the piano. My knowledge and learning will take me far. I used both 30 years ago. I will use both 30 years from now if I am still around. Such reliable tools help you stop thinking about tools, and start creating the art or engineering that is your gift to the rest of us. Don't get distracted by shiny objects. Find good tools, invest your time in them, learn then and use them forever.

Gary Wisniewski

Определить функцию, которая принимает целое положительное число и возвращает наибольший делитель этого числа.

Решение 1.85.1

```
(defun max-divisor (n)
  (loop for a from 2 to (truncate (/ n 2))
        when (zerop (rem n a)) return (/ n a)))

> (max-divisor 600851475143)
8462696833
```

Задача 1.86 card-deck-factorial.lisp

In my engineering field I have experienced quite often that the more difficult a tool is to learn the richer the rewards when you do learn it, emacs is one of those things, I tried, gave up but always would go back, there is just something about that editor, that keeps it on my mind, and oh how I hate to be defeated by anything even the great beast emacs, so I set out again on a venture to come to terms with this monstrously versatile and powerful piece of software, oh how I am enjoying myself.

~

Определить функцию, которая возвращает количество комбинаций карт в колоде.

Решение 1.86.1

```
(defun card-deck-factorial (n &optional (ac 1))
```

```
(if (plussp n) (card-deck-factorial (1- n) (* ac n)) ac)

> (card-deck-factorial 36)
371993326789901217467999448150835200000000
> (card-deck-factorial 52)
80658175170943878571660636856403766975289505440883277824000000000000
```

Задача 1.87 eai-most-frequentp.lisp

Just had this conversation at work and I asked my co-worker about how long he had been using emacs. He settled on about 15 years. I then asked when it was that he became comfortable using emacs and he said "just a few weeks ago". I think that there is some confusion between mastering emacs, and using emacs. You can learn to use emacs in 1/2 an hour. Is that a shockingly long time? Yes. Great design usually makes its uses obvious. But emacs makes up for that initial investment with accelerating returns - where Notepad, or even Eclipse, you stop gaining power and knowledge relatively quickly, emacs is like the universe: no matter how long you look at it, there is always more to learn --- and the best part is the more you learn, the faster you can learn more. So I don't care if I get a PhD before I master emacs. I will have fought, been conquered by, and subsumed within a worthy foe. I will see the world differently for having used emacs. Users of notepad will still be editing text... or maybe not. But it's a fun thought to think, isn't it?

Sam Bleckley

Дана строка. Определить функцию, которая проверяет следующую гипотезу: «Самые часто встречающиеся гласные - это е, а, i».

Решение 1.87.1

(defconstant *s* "Common Lisp (CL) is a dialect of the Lisp programming language, published in ANSI standard document ANSI INCITS 226-1994 (R2004) (formerly X3.226-1994 (R1999)). The Common Lisp HyperSpec, a hyperlinked HTML version, has been derived from the ANSI Common Lisp standard. The Common Lisp language was developed as a standardized and improved successor of MacLisp. By the early 1980s several groups were already at work on diverse successors to MacLisp: Lisp Machine Lisp (aka ZetaLisp), Spice Lisp, NIL and S-1 Lisp. Common Lisp sought to unify, standardise, and extend the features of these MacLisp dialects. Common Lisp is not an implementation, but rather a language

specification. Several implementations of the Common Lisp standard are available, including free and open source software and proprietary products. Common Lisp is a general-purpose, multi-paradigm programming language. It supports a combination of procedural, functional, and object-oriented programming paradigms. As a dynamic programming language, it facilitates evolutionary and incremental software development, with iterative compilation into efficient run-time programs that can be as performant as those written in C. This incremental development is often done interactively without interrupting the running application. It also supports optional type annotation and casting, which can be added as necessary at the later profiling and optimization stages, to permit the compiler to generate more efficient code. For instance, `fixnum` can hold an unboxed integer in a range supported by the hardware and implementation, permitting more efficient arithmetic than on big integers or arbitrary precision types. Similarly, the compiler can be told on a per-module or per-function basis which type safety level is wanted, using `optimize` declarations. Common Lisp includes CLOS, an object system that supports multimethods and method combinations. It is often implemented with a Metaobject Protocol. Common Lisp is extensible through standard features such as Lisp macros (code transformations) and reader macros (input parsers for characters). Common Lisp provides some backwards compatibility to MacLisp and to John McCarthy's original Lisp. This allows older Lisp software to be ported to Common Lisp")

```
(defun eai-most-frequentp (s)
  (not (set-difference
        '(#\e #\a #\i)
        (subseq (mapcar #'car
                        (sort (aeiouy-frequency s)
                             #'> :key #'cadr))
                  0 3)
        :test #'char-equal))))

(defun aeiouy-frequency (s)
  (loop for a in '(#\a #\e #\i #\o #\u #\y)
        collect (list a (count a s))))

> (aeiouy-frequency *s*)
((#\a 159) (#\e 179) (#\i 151) (#\o 141) (#\u 41) (#\y 22))
> (eai-most-frequentp *s*)
T
```

Задача 1.88 *odd-seqp.lisp*

Определить функцию, проверяющую, что список разностей соседних элементов в последовательности квадратов натуральных чисел представляет собой список последовательных нечётных чисел.

Решение 1.88.1

```
(defun odd-seqp (n)
```

```
(when (evenp n)
  (every
    #' oddp
    (mapcar
      #'-
      (loop for a from 1 to (1- n) by 2
        collect (* a a))
      (loop for a from 2 to n by 2
        collect (* a a))))))
```

```
> (odd-seqp 100)
T
```

Решение 1.88.2

```
(defun odd-seqp (n)
  (when (evenp n)
    (every
      #' oddp
      (loop for a from 1 to (1- n) by 2
        for b from 2 to n by 2
        collect (- (* a a) (* b b))))))
```

```
> (odd-seqp 100)
T
```

Решение 1.88.3

```
(defun odd-seqp (n)
  (when (evenp n)
    (loop for a from 1 to n
      if (oddp a) collect (* a a) into odds
      else collect (* a a) into evens
      finally (return
        (every #'oddp
          (mapcar #'- odds evens))))))
```

```
> (odd-seqp 100)
T
```

Решение 1.88.4

```
(defun odd-seqp (n) (when (evenp n) (odd-seq n)))

(defun odd-seq (n)
  (cond ((zerop n)
    ((evenp (- (expt (1- n) 2) (expt n 2))) nil)
    ((odd-seq (- n 2)))))
```

```
> (odd-seqp 100)
T
```

Решение 1.88.5


```

(loop for a across s
      for b from 1 to n
      if (> b (/ n 2)) sum (digit-char-p a) into p2
      else sum (digit-char-p a) into p1
      finally (return (= p1 p2))))

(defun add-zero (s n)
  (if (< (length s) n) (add-zero (concatenate 'string "0" s) n) s))

> (lucky-tktp 000001 6)
NIL
> (lucky-tktp 010001 6)
T

```

Решение 1.90.2

```

(defun lucky-tktp (n) (check-balance (add-zero (int-digit n) 6)))

(defun add-zero (w n)
  (if (< (length w) n) (add-zero (cons 0 w) n) w))

(defun int-digit (n &optional ac)
  (if (zerop n) ac
      (int-digit (truncate n 10) (cons (rem n 10) ac))))

(defun check-balance (w)
  (= (apply #'(subseq w 0 3)) (apply #'(nthcdr 3 w)))))

> (lucky-tktp 000001)
NIL
> (lucky-tktp 001001)
T

```

Задача 1.91 is-power-of-2.lisp

Определить функцию, проверяющую, является ли число степенью двойки.

Решение 1.91.1

```

(defun is-power-of-2 (n)
  (cond ((= n 1)
        ((zerop (rem n 2)) (is-power-of-2 (/ n 2)))
        (t nil)))

> (is-power-of-2 8)
T

```

Решение 1.91.2

```

(defun is-power-of-2 (n)
  (cond ((= n 1)
        ((zerop (rem n 2)) (is-power-of-2 (/ n 2)))

```

```
(nil)))
```

```
> (is-power-of-2 8)
T
```

Решение 1.91.3

```
(defun is-power-of-2 (n)
  (if (= n 1) t (when (zerop (rem n 2)) (is-power-of-2 (/ n 2)))))
```

```
> (is-power-of-2 8)
T
```

Решение 1.91.4

```
(defun is-power-of-2 (n)
  (cond ((= n 1)) ((zerop (rem n 2)) (is-power-of-2 (/ n 2)))))
```

```
> (is-power-of-2 8)
T
```

Задача 1.92 mini-max.lisp

С клавиатуры вводятся пять положительных целых чисел, разделенные пробелами. Определить функцию, выводящую в одной строке через пробел минимальное и максимальное значения, которые можно вычислить, суммируя ровно четыре из введенных чисел.

Пример ввода:

```
1 2 3 4 5
```

Образец вывода:

```
10 14
```

Решение 1.92.1

```
(defun read-input (n) (loop repeat n collect (read)))
```

```
(defun mi-ma (n w)
  (loop for m from 1 to n collect
    (reduce #' + (loop for a in w
                      for b from 1 to n
                      unless (= m b) collect a))))
```

```
(defun mini-max (n &aux (v (mi-ma n (read-input n))))
  (princ (reduce #'min v))
  (princ " ")
  (princ (reduce #'max v)))
```

```
> (mini-max 5)
1 2 3 4 5
10 14
```


14

Задача 1.93 convert.lisp

- Бабушка, а кто такие «люди с гуманитарным складом ума»?
 - Даже и не знаю, как тебе сказать, внучек... Раньше их дураками называли.

~

Время вводится с клавиатуры в формате AM/PM. Определите функцию, выводящую время, конвертированное в 24-часовую шкалу.

Пример ввода:
 07:05:45PM

Образец вывода:
 19:05:45

Решение 1.93.1

```
(defun convert (&aux
  (s (read-line))
  (z (subseq s 8))
  (b (subseq s 0 8))
  (c (subseq b 0 2))
  (a (write-to-string
      (+ (parse-integer c) 12))))
  (if (string-equal z "AM")
      (format t "~a" (if (string-equal c "12")
                        (concatenate 'string "00" (subseq b 2))
                        b))
      (format t "~a"
        (if (string-equal c "12") b
            (concatenate 'string a (subseq b 2))))))

> (convert)
07:05:45PM
19:05:45
NIL
```

Задача 1.94 same-length-word-textp.lisp

Дан текстовый файл. Проверить, что все слова одинаковой длины. Словом является любая последовательность символов между разделителями: знаками препинания, пробелами, концами строк.

Например, test.txt:

aaaa, bbbb, cccc.
 dddd, eeee,

ffff.

Решение 1.94.1

```
(defun same-length-word-textp (path &aux w)
  (with-open-file (s path :direction :input)
    (do ((line (read-line s nil :eof) (read-line s nil :eof)))
        ((eql line :eof))
      (setf w (nconc (list-string line) w)))
    (loop for a in w
      always (= (length (string (car w)))
                 (length (string a))))))

(defun list-string (s)
  (read-from-string
   (concatenate
    'string "(" (delete-if-not #'(lambda (x)
                                   (or (alpha-char-p x)
                                       (equal x #\space)
                                       (equal x #\-)))
                                s) ")"))))

> (same-length-word-textp "c:/temp/test.txt")
T
```

Задача 1.95 fish.lisp

In Lisp, everything is a list. In Smalltalk, everything is an object. Both are extremely simple and elegant.

Richard Kenneth Eng

Всероссийская олимпиада школьников. Типовые задания I (школьного) этапа по математике. 8 класс: Голова рыбы весит столько, сколько хвост и половина туловища, туловище – столько, сколько голова и хвост вместе. Хвост весит 1 кг. Сколько весит рыба?

Решение 1.95.1

```
(defun fish (n) (+ (head n) (body n) (tail n)))

(defun body (n) (+ (head n) n))

(defun head (n) (* 3 n))
; сумма половин (голова минус хвост) равна весу туловища:
; (h - n) + (h - n) = h + n -> h = 3n

(defun tail (n) n)

> (fish 1)
8
```

Задача 1.96 0102-string.lisp

...certain areas of interest are best catered to by adding lots of manpower to their solution, areas which will be "popular" in the most obvious sense of the word, while other areas of interest will not attract people in great spades regardless of the monetary rewards, such as those that ask for significant dedication because of such things as very high risks, skill requirements, entry costs, etc. if you choose one of those areas of interest, no manager in his right mind places silly demands on your programming language of choice and he will probably fire you if you choose "popular" languages subject to vendors who care only about the mass market and not about quality, unless his real plan is to fire you, anyway, only to replace you by someone equally uncritical of his tools. ...solution: find areas of interest not invaded by populist opportunists.

Erik Naggum

Дано целое число *n*. Определить функцию, которая возвращает строку "00 01 02 03 04 0*n*".

Решение 1.96.1

```
(defun 0102-string (n &optional (ac ""))
  (if (minusp n) (string-right-trim " " ac)
      (0102-string (1- n) (concatenate 'string "0"
                                         (write-to-string n) " "
                                         ac))))

> (0102-string 7)
"00 01 02 03 04 05 06 07"
```

Решение 1.96.2

```
(defun 0102-string (n &aux (m (1+ (length (write-to-string n)))))
  (string-right-trim " "
    (format nil (concatenate 'string
                              "~{"~"
                              (write-to-string m)
                              ",'0d ~}")
              (loop for a upto n collect a))))

> (0102-string 12)
"000 001 002 003 004 005 006 007 008 009 010 011 012"
```

Задача 1.97 *mirror.lisp*

Compared to Assembly code, COBOL and Fortran, Lisp provided programmers with capabilities that were light-years ahead. Lisp introduced functions-as-data, the tree data structure, automatic memory management, dynamic typing, recursion, and the REPL and many other innovations.

Paul Reiber

Определить функцию, меняющую знак числа на противоположный.

Решение 1.97.1

```
(defun mirror (n) (- n))
```

```
> (mirror -1)
```

```
1
```

Структура 2 (функция АТОМ) > (СПИСОК)

Lisp is worth learning for a different reason — the profound enlightenment experience you will have when you finally get it. That experience will make you a better programmer for the rest of your days, even if you never actually use Lisp itself a lot.

Eric Raymond

I used vi for 7 years. I tried emacs for 7 days and then I never used vi if I didn't have to.

—

I like it better than vi, and I use vi all the time. What else are you comparing emacs to?

—

Задача 2.1 numbers.lisp

Определить функцию, которая возвращает список чисел от 1 до n.

Решение 2.1.1

```
(defun numbers (n)
  (loop for a from 1 to n collect a))

> (numbers 100000)
(1 2 3 4 5 6 7 8 9 10 11 12 13 14... 99997 99998 99999 100000)
> (numbers 1000000)
(1 2 3 4 5 6 7 8 9 10 11 12 13 14... 999997 999998 999999 1000000)
```

Решение 2.1.2

```
(defun numbers (n &optional ac)
  (cond ((zerop n) ac)
        (t (numbers (1- n) (cons n ac)))))

> (numbers 100000)
(1 2 3 4 5 6 7 8 9 10 11 12 13 14... 99997 99998 99999 100000)
> (numbers 1000000)
(1 2 3 4 5 6 7 8 9 10 11 12 13 14... 999997 999998 999999 1000000)
```

Решение 2.1.3

```
(defun numbers (n &optional ac)
  (cond ((zerop n) ac)
        (t (numbers (1- n) (push n ac))))))

> (numbers 100000)
(1 2 3 4 5 6 7 8 9 10 11 12 13 14... 99997 99998 99999 100000)
> (numbers 1000000)
(1 2 3 4 5 6 7 8 9 10 11 12 13 14... 999997 999998 999999 1000000)
```

Решение 2.1.4

```
(defun numbers (n &optional ac)
  (if (plusp n) (numbers (1- n) (cons n ac)) ac))

> (time (numbers 100000))
(1 2 3 4 5 6 7 8 9 10 11 12 13 14... 99997 99998 99999 100000)
> (time (numbers 100000))
Real time: 0.0624001 sec.
Run time: 0.0624004 sec.
Space: 800000 Bytes
GC: 1, GC time: 0.0312002 sec.
> (time (numbers 1000000))
Real time: 0.3588006 sec.
Run time: 0.3588023 sec.
Space: 8000000 Bytes
GC: 4, GC time: 0.1404009 sec.
(1 2 3 4 5 6 7 8 9 10 11 12 13 14... 999997 999998 999999 1000000)
```

Задача 2.2 numbers-in-reverse.lisp

Определить функцию, которая возвращает список из первых n чисел в обратном порядке.

Решение 2.2.1

```
(defun numbers-in-reverse (n)
  (unless (zerop n) (cons n (numbers-in-reverse (1- n)))))

> (numbers-in-reverse 100)
(100 99 98 97 96 95 94 93 92 91 90 89 88 87 86 85 84 83 82 81 80 79 78
77 76 75 74 73 72 71 70 69 68 67 66 65 64 63 62 61 60 59 58 57 56 55
54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8
7 6 5 4 3 2 1)
```

Решение 2.2.2

```
(defun numbers-in-reverse (n)
  (when (> n 0) (cons n (numbers-in-reverse (1- n)))))

> (numbers-in-reverse 100)
(100 99 98 97 96 95 94 93 92 91 90 89 88 87 86 85 84 83 82 81 80 79 78
```

```

77 76 75 74 73 72 71 70 69 68 67 66 65 64 63 62 61 60 59 58 57 56 55
54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8
7 6 5 4 3 2 1)

```

Решение 2.2.3

```

(defun numbers-in-reverse (n)
  (when (plusp n) (cons n (numbers-in-reverse (1- n))))))

> (time (numbers-in-reverse 10000))
Real time: 0.0 sec.
Run time: 0.0 sec.
Space: 80000 Bytes
(10000 9999 9998 9997... 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1)
> (time (numbers-in-reverse 100000))
; Evaluation aborted
Real time: 0.0156 sec.
Run time: 0.0156001 sec.
Space: 0 Bytes

```

Решение 2.2.4

```

(defun numbers-in-reverse (n)
  (loop for a from n downto 1 collect a))

> (time (numbers-in-reverse 100000))
Real time: 0.0312 sec.
Run time: 0.0312002 sec.
Space: 800000 Bytes
GC: 2, GC time: 0.0156001 sec.
(100000 99999 99998 99997... 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1)

```

Решение 2.2.5

```

(defun numbers-in-reverse (n &optional ac)
  (cond ((zerop n) (nreverse ac))
        (t (numbers-in-reverse (1- n) (cons n ac)))))

> (time (numbers-in-reverse 100000))
Real time: 0.0312001 sec.
Run time: 0.0312002 sec.
Space: 800000 Bytes
(100000 99999 99998 99997... 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1)

```

Решение 2.2.6

```

(defun numbers-in-reverse (n &optional ac)
  (if (plusp n)
      (numbers-in-reverse (1- n) (push n ac))
      (nreverse ac)))

> (time (numbers-in-reverse 100000))

```

```
Real time: 0.0312 sec.
Run time: 0.0312002 sec.
Space: 800000 Bytes
(100000 99999 99998 99997... 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1)
```

Задача 2.3 *balanced.lisp*

Lisp helps in solving problems in bottom up fashion when you don't have the clear solution of a problem and it is required to try out different things.

Ruchir Hajela

Решить следующую задачу: вводится массив 4-хзначных чисел, программа выводит числа, у которых сумма первых двух элементов числа равна сумме двух вторых элементов числа. То есть первое число будет 0000 последнее число 9999, а выводит числа такие 0000, 0101, 2525, 2552 и так далее.

Решение 2.3.1

```
(defun xd (n p)
  (parse-integer (format nil "~4,'0d" n) :start p :end (1+ p)))

(defun bd (m)
  (loop for n from 0 to m
        when (= (+ (xd n 0) (xd n 1)) (+ (xd n 2) (xd n 3)))
        collect n))

(defun balanced (m)
  (mapcar #'(lambda (a) (format nil "~4,'0d" a)) (bd m)))

> (balanced 9999)
("0000" "0101" "0110" "0202" ... "9788" "9797" "9889" "9898" "9999")
```

Решение 2.3.2

```
(defun xd (n p)
  (parse-integer (format nil "~4,'0d" n) :start p :end (1+ p)))

(defun bd (m &optional (n 0))
  (cond ((> n m) nil)
        ((/= (+ (xd n 0) (xd n 1)) (+ (xd n 2) (xd n 3)))
         (bd m (1+ n)))
        ((cons n (bd m (1+ n))))))

(defun balanced (m)
  (mapcar #'(lambda (a) (format nil "~4,'0d" a)) (bd m)))

(balanced 9999)
("0000" "0101" "0110" "0202" ... "9788" "9797" "9889" "9898" "9999")
```


Задача 2.4 int>digits.lisp

Определить функцию для построения списка цифр числа.

Решение 2.4.1

```
(defun int>digits (n)
  (loop for c across (write-to-string n) collect (digit-char-p c)))

> (int>digits 123)
(1 2 3)
```

Решение 2.4.2

```
(defun int>digits (n &optional ac)
  (if (zerop n) ac (int>digits (truncate n 10) (cons (rem n 10) ac))))

> (int>digits 123)
(1 2 3)
```

Решение 2.4.3

```
(defun int>digits (n)
  (mapcar #'digit-char-p (coerce (write-to-string n) 'list)))

> (int>digits 123)
(1 2 3)
```

Решение 2.4.4

```
(defun int>digits (n)
  (mapcar #'digit-char-p (concatenate 'list (write-to-string n))))

> (int>digits 123)
(1 2 3)
```

Решение 2.4.5

```
(defun int>digits (n)
  (mapcar #'digit-char-p (map 'list #'identity (write-to-string n))))

> (int>digits 123)
(1 2 3)
```

Решение 2.4.6

```
(defun int>digits (n)
  (mapcar #'parse-integer (map 'list #'string (write-to-string n))))

> (int>digits 123)
(1 2 3)
```

Решение 2.4.7

```
(defun int>digits (n)
  (mapcar #'parse-integer (map 'list #'string (format nil "~D" n))))

> (int>digits 123)
(1 2 3)
```

Задача 2.5 squares.lisp

Lisp will free your mind... Most programmers will not spend enough time with Lisp to ever get it (also the reason why Lisp has few but ardent adherents). But once you do reach this level of understanding, nothing else will suffice.

Anurag Mendhekar

Написать функцию построения списка квадратов чисел от 1 до n.

Решение 2.5.1

```
(defun squares (n &optional (m 1))
  (when (>= n m) (cons (* m m) (squares n (1+ m)))))

> (squares 10)
(1 4 9 16 25 36 49 64 81 100)
```

Решение 2.5.2

```
(defun squares (n) (loop for x from 1 to n collect (* x x)))

> (squares 10)
(1 4 9 16 25 36 49 64 81 100)
```

Решение 2.5.3

```
(defun squares (n)
  (if (> n 0) (nconc (squares (1- n)) (list (* n n)))))

> (squares 10)
(1 4 9 16 25 36 49 64 81 100)
```

Задача 2.6 prin&list.lisp

Определить функцию (f n), которая сначала печатает строку чисел без пробелов n n-1 ... 1, а на второй строке выводит значение в виде списка чисел (n n-1 ... 1).

Решение 2.6.1

```
(defun prin&list (n &optional ac)
  (cond ((zerop n) (reverse ac))
        (t (prin1 n) (prin&list (- n 1) (cons n ac)))))

> (prin&list 6)
654321
(6 5 4 3 2 1)
```

Решение 2.6.2

```
(defun pa (n ac)
  (cond ((zerop n) (reverse ac))
        (t (prin1 n) (pa (- n 1) (cons n ac)))))

(defun prin&list (n)
  (pa n nil))

> (prin&list 6)
654321
(6 5 4 3 2 1)
```

Решение 2.6.3

```
(defun prin&list (n)
  (labels ((pa (n ac)
            (cond ((zerop n) (reverse ac))
                  (t (prin1 n) (pa (- n 1) (cons n ac)))))
    (pa n nil)))

> (prin&list 6)
654321
(6 5 4 3 2 1)
```

Решение 2.6.4

```
(defun prin&list (n)
  (let ((r))
    (dotimes (i n r)
      (prin1 (- n i))
      (setq r (cons (+ i 1) r)))))

> (prin&list 6)
654321
(6 5 4 3 2 1)
```

Задача 2.7 *deci-bin.lisp*

Определить функцию для перевода числа из десятичной системы в двоичную.

Решение 2.7.1

```
(defun deci-bin (n &optional ac)
  (if (zerop n) ac (deci-bin (floor n 2) (cons (mod n 2) ac))))

> (deci-bin 27)
(1 1 0 1 1)
```

Решение 2.7.2

```
(defun deci-bin (n) (format nil "~b" n))

> (deci-bin 27)
"11011"
```

Задача 2.8 *bin-deci.lisp*

Определить функцию для перевода числа (записанного в виде списка) из двоичной системы в десятичную.

Решение 2.8.1

```
(defun bin-deci (w &optional (n 0))
  (if (null w) n (bin-deci (cdr w) (+ (car w) n))))

> (bin-deci '(1 1 0 1 1))
27
```

Задача 2.9 *drop-word-duplicates.lisp*

В файле записаны слова, некоторые из них повторяются. Требуется считать слова из этого файла и формировать из них список, в котором повторов не будет. Результат записать в файл.

Решение 2.9.1

```
(defun list-string (s)
  (read-from-string
   (concatenate
    'string "(" (delete-if-not #'(lambda (x)
                                   (or (alpha-char-p x)
                                       (equal x #\space)
                                       (equal x #\-))))
          s) ")"))))

(defun drop-word-duplicates (path-in path-out)
  (let ((u))
    (with-open-file (s path-in :direction :input)
      (with-open-file
        (r path-out :direction :output :if-exists :supersede)
        (do ((line (read-line s nil :eof) (read-line s nil :eof)))
            ((eql line :eof))
```

```

      (setq u (nconc (list-string line) u)))
      (format r "~a" (remove-duplicates u))))))

> (drop-word-duplicates "file-in.txt" "file-out.txt")
NIL

file-in.txt
aaa bbb aaa bbb

file-out.txt
(AAA BBB)

```

Задача 2.10 *prime2n-1.lisp*

It was common to say that Lisp, Prolog and Smalltalk were the languages to study if you wanted to become a better programmer. And among those Lisp was considered to be most important.

Andrea Ferro

Составить программу, вычисляющую $2n-1=N$ (где n степень числа 2) и проверяющую является ли N простым числом. Число n меняется от 0 до 100. Вывести все n при которых N простое.

Решение 2.10.1

```

(defun primep (n)
  (loop for a from 2 to (isqrt n) never (zerop (mod n a))))

(defun lucas-lehmer (n)
  (loop with m = (1- (expt 2 n))
        for i from 1 to (1- n)
        for j = 4 then (rem (- (expt j 2) 2) m)
        finally (return (zerop j))))

(defun prime2n-1 (m &optional (v '(2)) (n 2))
  (cond ((= m 2) (reverse v))
        ((and (primep n) (lucas-lehmer n)
              (prime2n-1 (1- m) (cons n v) (1+ n)))
         ((prime2n-1 (1- m) v (1+ n)))))

> (prime2n-1 100)
(2 3 5 7 13 17 19 31 61 89)

```

Задача 2.11 *count-letters.lisp*

Дан текст из латинских букв и знаков препинания. Составить программу частотного анализа букв этого текста, т.е. напечатать каждую букву с указанием количества ее вхождений.

Решение 2.11.1

```
(defun count-letters (pathname)
  (with-open-file (s pathname :direction :input)
    (loop for line = (read-line s nil nil)
          while line
          nconc (list-symb (drop-noise line)) into words
          finally (return (sort (list-pair words) #'> :key #'cdr)))))

(defun list-symb (s)
  (read-from-string (concatenate 'string "(" s ")")))

(defun drop-noise (s)
  (delete-if-not
   #'(lambda (x) (or (alpha-char-p x)
                     (equal x #\space)))
   s))

(defun list-pair (words &aux (hash (make-hash-table)) acc)
  (dolist (word words) (incf (gethash word hash 0)))
  (maphash #'(lambda (e n) (push `(,e . ,n) acc)) hash) acc)

> (count-letters "test.txt")
((A . 2) (E . 1) (V . 1) (S . 1))
```

Задача 2.12 *print-linen.lisp*

Определите функцию (f n), печатающую n раз квадрат 2 x 2 звездочками (*) с промежутком 2 знака.

Решение 2.12.1

```
(defun print-linen (n)
  (dotimes (i n) (format t "* * ")
    (format t "~%")
    (dotimes (i n) (format t "* * "))))

> (print-linen 4)
* * * * *
* * * * *
```

Задача 2.13 *odd50.lisp*

Опишите функцию, которая печатает все нечетные числа от 0 до 100 в порядке возрастания, по одному на строке.

Решение 2.13.1

```
(defun odd50 (&optional (n 1))
  (when (< n 100) (format t "~a~%" n) (odd50 (+ n 2)))))
```

```
> (odd50)
1
3
...
99
NIL
```

Задача 2.14 f.lisp

Определить функцию без параметров, конструирующую список (A B ((C D) E)) из одноуровневых списков и атомов, которые вводятся с клавиатуры.

Решение 2.14.1

```
(defun f ()
  `(, (read) , (read) (, (read) , (read))))

> (f)
1
2
(3 4)
5
(1 2 ((3 4) 5))
```

Решение 2.14.2

```
(defun |[1 2 [[3 4] 5]]| ()
  `(, (read) , (read) (, (read) , (read))))

> (|[1 2 [[3 4] 5]]|)
1
2
(3 4)
5
(1 2 ((3 4) 5))
```

Решение 2.14.3

```
(defun [12[[34]5]] ()
  `(, (read) , (read) (, (read) , (read))))

> ([12[[34]5]])
1
2
(3 4)
5
(1 2 ((3 4) 5))
```

Решение 2.14.4

```
(defun [ab[[cd]e]] ()
```

```
`(,(read) ,(read) ,(read) ,(read)))
> ([ab[[cd]e]])
1
2
(3 4)
5
(1 2 ((3 4) 5))
```

Задача 2.15 *magic-ticket-9999.lisp*

Lisp teaches you not only to think outside the box but also about the box.

Vladislav Zorov

Найти номера "счастливых" билетов (сумма первых двух цифр равна сумме последних двух цифр) с номерами от 0000 до 9999 включительно.

Решение 2.15.1

```
(defun magic-ticket-9999 (n)
  (row-ticket (add-nine n) n))

(defun row-ticket (nn n &aux (m (int-digit nn)) (z (add-zero m n)))
  (when (>= nn 0) (if (check-balance z (/ n 2))
    (cons z (row-ticket (1- nn) n))
    (row-ticket (1- nn) n))))

(defun add-nine (n &optional w)
  (if (< (length w) n) (add-nine n (cons 9 w)) (list-num w)))

(defun list-num (w)
  (parse-integer (apply #'concatenate 'string
    (mapcar #'write-to-string w))))

(defun add-zero (w n)
  (if (< (length w) n) (add-zero (cons 0 w) n) w))

(defun int-digit (n &optional ac)
  (if (zerop n) ac (int-digit (truncate n 10) (cons (rem n 10) ac))))

(defun check-balance (w n)
  (= (apply #'+ (subseq w 0 n)) (apply #'+ (nthcdr n w))))

> (time (magic-ticket-9999 4))
Real time: 0.0624001 sec.
Run time: 0.0624004 sec.
Space: 487976 Bytes
GC: 1, GC time: 0.0 sec.
((9 9 9 9) (9 8 9 8) (9 8 8 9)...(0 1 1 0) (0 1 0 1) (0 0 0 0))
```


Решение 2.15.2

```

(defun magic-ticket-9999 (n)
  (row-ticket (add-nine n) n))

(defun row-ticket (nn n &aux (m (int-digit nn)) (z (add-zero m n)))
  (when (>= nn 0) (if (check-balance z (/ n 2))
    (cons z (row-ticket (1- nn) n))
    (row-ticket (1- nn) n))))

(defun add-nine (n &optional w)
  (if (< (length w) n) (add-nine n (cons 9 w)) (list-num w)))

(defun list-num (w)
  (parse-integer (apply #'concatenate 'string
    (mapcar #'write-to-string w))))

(defun add-zero (w n)
  (if (< (length w) n) (add-zero (cons 0 w) n) w))

(defun int-digit (n)
  (loop with r do (setf (values n r) (truncate n 10))
    collect r into ac until (zerop n)
    finally (return (nreverse ac))))

(defun check-balance (w n)
  (= (apply #' + (subseq w 0 n)) (apply #' + (nthcdr n w))))

> (time (magic-ticket-9999 4))
Real time: 0.0740007 sec.
Run time: 0.078125 sec.
Space: 487976 Bytes
GC: 1, GC time: 0.0 sec.
((9 9 9 9) (9 8 9 8) (9 8 8 9)...(0 1 1 0) (0 1 0 1) (0 0 0 0))

```

Задача 2.16 magic-ticket-222222.lisp

Lisp enables untyped (or rather, untyped) functional programming, without enforcing it.

Andrea Ferro

Найти число "счастливых" билетов (сумма первых трех цифр равна сумме последних трех цифр) с номерами от 000000 до 222222 включительно.

Решение 2.16.1

```

(defun magic-ticket-222222 (n)
  (row-ticket (add-two n) n))

(defun row-ticket (nn n &aux (m (int-digit nn)) (z (add-zero m n)))

```

```

    (when (>= nn 0) (if (check-balance z (/ n 2))
                        (cons z (row-ticket (1- nn) n))
                        (row-ticket (1- nn) n))))

(defun add-two (n &optional w)
  (if (< (length w) n) (add-two n (cons 2 w)) (list-num w)))

(defun list-num (w)
  (parse-integer (apply #'concatenate 'string
                       (mapcar #'write-to-string w))))

(defun add-zero (w n)
  (if (< (length w) n) (add-zero (cons 0 w) n) w))

(defun int-digit (n &optional ac)
  (if (zerop n) ac (int-digit (truncate n 10) (cons (rem n 10) ac))))

(defun check-balance (w n)
  (= (apply #'+ (subseq w 0 n)) (apply #'+ (nthcdr n w))))

> (magic-ticket-222222 6)
((2 2 2 2 2 2) (2 2 2 2 1 3) (2 2 2 2 0 4)... (0 0 0 0 0 0))

```

Задача 2.17 sum-20.lisp

Найти все пары чисел в диапазоне от 1 до 20, сумма которых равна 20.

Решение 2.17.1

```

(defun sum-20 ()
  (loop for a from 1 to 19
        for b from 19 downto 1
        collect `(,a ,b)))

> (sum-20)
((1 19) (2 18) (3 17) (4 16) (5 15) (6 14) (7 13) (8 12) (9 11) (10
10) (11 9) (12 8) (13 7) (14 6) (15 5) (16 4) (17 3) (18 2) (19 1))

```

Задача 2.18 sum-n.lisp

Найти все пары чисел в диапазоне от 1 до n, сумма которых равна n.

Решение 2.18.1

```

(defun sum-n (n)
  (loop for a from 1 to (1- n)
        for b from (1- n) downto 1
        collect `(,a ,b)))

> (sum-n 20)
((1 19) (2 18) (3 17) (4 16) (5 15) (6 14) (7 13) (8 12) (9 11) (10

```

```
10) (11 9) (12 8) (13 7) (14 6) (15 5) (16 4) (17 3) (18 2) (19 1))
```

Задача 2.19 onion-n.lisp

Определите функцию, которая создает на n - уровне вложенный список, элементом которого на самом нижнем уровне является n .

Решение 2.19.1

```
(defun onion-n (n &optional (m n))
  (if (zerop n)
      m
      (list (onion-n (1- n) m))))

> (onion-n 7)
(((((((7)))))))
```

Решение 2.19.2

```
(defun onion-n (n &optional (m n))
  (if (zerop n) m `((, (onion-n (1- n) m))))

> (onion-n 7)
(((((((7)))))))
```

Решение 2.19.3

```
(defun onion-n (n &optional (m n))
  (if (zerop n)
      m
      (cons (onion-n (1- n) m) nil)))

> (onion-n 7)
(((((((7)))))))
```

Задача 2.20 factorials.lisp

Построить список, элементами которого являются значения факториала от 1 до n .

Решение 2.20.1

```
(defun factorial (n)
  (if (= n 0) 1 (* n (factorial (- n 1)))))

(defun factorials (n)
  (loop for a from 1 to n collect (factorial a)))

> (factorials 5)
(1 2 6 24 120)
```

Решение 2.20.2

```
(defun factorial (n)
  (if (= n 0) 1 (* n (factorial (- n 1)))))

(defun factorials (n)
  (when (> n 0) (cons (factorial n) (factorials (1- n)))))

> (factorials 5)
(120 24 6 2 1)
```

Решение 2.20.3

```
(defun factorial (n)
  (if (= n 0) 1 (* n (factorial (- n 1)))))

(defun factorials (n &optional ac)
  (if (> n 0)
      (factorials (1- n) (cons (factorial n) ac))
      ac))

> (factorials 5)
(1 2 6 24 120)
```

Решение 2.20.4

```
(defun factorial (n)
  (if (= n 0) 1 (* n (factorial (- n 1)))))

(defun factorials (n &optional ac)
  (cond ((zerop n) ac)
        ((factorials (1- n) (cons (factorial n) ac)))))

> (factorials 5)
(1 2 6 24 120)
```

Задача 2.21 *dice-game.lisp*

Написать простой вариант игры в кости, в котором бросаются две правильные кости. Если сумма выпавших очков равна 7 или 11 – выигрыш. Если выпало (1,1) или (6,6) – игрок получает право снова бросить кости, во всех остальных случаях ход переходит ко второму игроку, но запоминается сумма выпавших очков. Если второй игрок не выигрывает абсолютно, то выигрывает тот игрок, у которого больше очков. Результат игры и значения выпавших костей выводить на экран.

Решение 2.21.1

```
(defun dice (&optional (d 0))
  (aux (a (random 6)) (b (random 6)) (s (+ a b)))
  (cond ((or (= a b 1) (= a b 6))
        (print (list a '+' b '=' s)) (dice (+ d s)))))
```

```

((or (= s 7) (= s 11)) (print (list a ' + b '= s)) s)
(t (print (list a ' + b '= (if (zerop d) s (+ d s))))
  (if (zerop d) s (+ d s))))

(defun dice-game (&aux (n (dice)) (m (dice)))
  (cond ((or (= n 7) (= n 11) (> n m)) 'you-win)
        ((or (= m 7) (= m 11) (< n m)) 'he-wins)
        (t 'even)))

> (dice-game)
(5 + 2 = 7)
(5 + 4 = 9)
YOU-WIN

```

Решение 2.21.2

```

(defun prnt (a b s)
  (print (list a ' + b '= s)))

(defun dice (&optional (d 0))
  &aux (a (random 6)) (b (random 6)) (s (+ a b)))
  (cond ((or (= a b 1) (= a b 6)) (prnt a b s) (dice (+ d s)))
        ((or (= s 7) (= s 11)) (prnt a b s) s)
        (t (let ((m (if (zerop d) s (+ d s)))) (prnt a b m) m))))

(defun dice-game (&aux (n (dice)) (m (dice)))
  (cond ((or (= n 7) (= n 11) (> n m)) 'you-win)
        ((or (= m 7) (= m 11) (< n m)) 'he-wins)
        (t 'even)))

> (dice-game)
(1 + 1 = 2)
(5 + 1 = 8)
(2 + 2 = 4)
YOU-WIN

```

Задача 2.22 *rand.lisp*

Реализовать функцию которая порождает по заданному N список, состоящий из N случайных натуральных чисел

Решение 2.22.1

```

(defun rand (n)
  (unless (zerop n) (cons (random 101) (rand (1- n)))))

> (rand 10)
(39 86 7 68 81 6 34 34 44 97)

```

Решение 2.22.2

```

(defun rand (n)

```

```
(when (plussp n) (cons (random 101) (rand (1- n)))))
```

```
> (rand 10)
(27 70 66 92 17 45 48 81 76 4)
```

Решение 2.22.3

```
(defun rand (n)
  (loop for i from 1 to n collect (random 101)))
```

```
> (rand 10)
(38 77 22 27 74 90 56 5 26 17)
```

Решение 2.22.4

```
(defun rand (n)
  (loop repeat n collect (random 101)))
```

```
> (rand 10)
(67 90 76 82 96 4 94 27 73 44)
```

Задача 2.23 *cumulate.lisp*

Написать программу, вводящую данные из файла, разделив содержание нечетных и четных строк.

```
d:/test.txt:
1 1
a a
2 2
b b
```

Решение 2.23.1

```
(defun cumulate (path &aux (n 1) a b)
  (with-open-file (s path :direction :input)
    (do ((line (read-line s nil :eof) (read-line s nil :eof)))
        ((eql line :eof))
        (if (evenp n)
            (progn (setf n (1+ n)) (setf a (nconc a (sw line))))
            (progn (setf n (1+ n)) (setf b (nconc b (sw line)))))))
  (values a b))
```

```
(defun sw (s)
  (read-from-string (concatenate 'string "(" s ")")))
```

```
> (cumulate "d:/test.txt")
(1 1 2 2)
(A A B B)
```

Решение 2.23.2

```

(defun cumulate (path &aux (n 1) a b)
  (with-open-file (s path :direction :input)
    (do ((line (read-line s nil :eof) (read-line s nil :eof)))
        ((eql line :eof))
      (cond ((evenp n) (setf n (1+ n)) (setf a (nconc a (sw line))))
            ((setf n (1+ n)) (setf b (nconc b (sw line)))))))
  (values a b))

(defun sw (s)
  (read-from-string (concatenate 'string "(" s ")")))

> (cumulate "d:/test.txt")
(1 1 2 2)
(A A B B)

```

Решение 2.23.3

```

(defun cumulate (path &aux (n 1) a b)
  (with-open-file (s path :direction :input)
    (do ((line (read-line s nil :eof) (read-line s nil :eof)))
        ((eql line :eof))
      (setf n (1+ n))
      (setf (if (evenp n) a b)
            (nconc (if (evenp n) a b) (sw line)))))
  (values a b))

(defun sw (s)
  (read-from-string (concatenate 'string "(" s ")")))

> (cumulate "d:/test.txt")
(1 1 2 2)
(A A B B)

```

Решение 2.23.4

```

(defun cumulate (path &aux (n 1) a b)
  (with-open-file (s path :direction :input)
    (do ((line (read-line s nil :eof) (read-line s nil :eof)))
        ((eql line :eof))
      (if (oddp n)
          (setf a (nconc a (sw line)))
          (setf b (nconc b (sw line))))
      (setf n (1+ n))))
  (values a b))

(defun sw (s)
  (read-from-string (concatenate 'string "(" s ")")))

> (cumulate "d:/test.txt")
(1 1 2 2)
(A A B B)

```

Задача 2.24 *divisors.lisp*

Определить функцию, которая принимает целое положительное число и возвращает список делителей этого числа.

Решение 2.24.1

```
(defun divisors (n)
  (loop for a from 1 to (truncate (/ n 2))
        when (zerop (rem n a))
        collect a))

> (divisors 100)
(1 2 4 5 10 20 25 50)
> (divisors 99)
(1 3 9 11 33)
```

Решение 2.24.2

```
(defun divisors (n &optional (m (truncate (/ n 2))))
  (cond ((zerop m) nil)
        ((zerop (rem n m)) (cons m (divisors n (1- m))))
        ((divisors n (1- m)))))

> (divisors 100)
(50 25 20 10 5 4 2 1)
> (divisors 99)
(33 11 9 3 1)
```

Решение 2.24.3

```
(defun divisors (n &optional (m (truncate (/ n 2))))
  (when (plusp m) (if (zerop (rem n m))
                      (cons m (divisors n (1- m)))
                      (divisors n (1- m)))))

> (divisors 100)
(50 25 20 10 5 4 2 1)
> (divisors 99)
(33 11 9 3 1)
```

Задача 2.25 *suffixes.lisp*

Определить функцию, которая строит список суффиксов строки.

Решение 2.25.1

```
(defun suffixes (s)
  (mapcar #'(lambda (e) (coerce e 'string))
    (cdr (loop for d on
               (loop for a across s
                     collect a))
```



```
by #'cdr collect d))))
```

```
> (suffixes "abc")
("bc" "c")
```

Задача 2.26 bin-deci.lisp

Определить функцию для перевода числа из двоичной системы в десятичную.

Решение 2.26.1

```
(defun bin-deci (n)
  (parse-integer (write-to-string n) :radix 2))

> (bin-deci 1010)
10
4
```

Задача 2.27 fn>*123.lisp

Определить функцию, результатом которой будет выражение, являющееся факториалом числа, и в котором числа (сомножители) упорядочены в порядке возрастания.

Решение 2.27.1

```
(defun fn>*123 (n &optional ac)
  (if (zerop n) (cons '* ac) (fn>*123 (1- n) (cons n ac))))

> (fn>*123 10)
(* 1 2 3 4 5 6 7 8 9 10)
> (eval (fn>*123 10))
3628800
```

Задача 2.28 res.lisp

Определить функцию, формирующую список студентов - имя, год рождения, адрес, средний балл.

Решение 4.28.1

```
(defun res (n)
  (when (> n 0)
    (cons (list (read) (read) (read) (read))
          (res (1- n)))))

> (res 3)
ivanov
1992
pushkina-10
```

```

4
petrov
1992
gogolya-20
5
semenov
1992
esenina-10
4
((IVANOV 1992 PUSHKINA-10 4) (PETROV 1992 GOGOLYA-20 5)
 (SEMENOV 1992 ESENINA-10 4))

```

Задача 2.29 amicable-nums.lisp

Два натуральных числа называются дружественными, если сумма всех делителей числа первого равна второму, а сумма всех делителей второго числа равна первому числу. Найдите пары дружественных чисел меньших 1000.

Решение 2.29.1

```

(defun sum-divisors (n)
  (loop for a from 1 to (truncate (/ n 2))
        when (zerop (rem n a))
        sum a))

(defun amicable-nums (m)
  (loop for n from 2 to m when (= (sum-divisors (sum-divisors n)) n)
        collect (list n (sum-divisors n))))

> (amicable-nums 15000)
((6 6) (28 28) (220 284) (284 220) (496 496) (1184 1210) (1210 1184)
 (2620 2924) (2924 2620) (5020 5564) (5564 5020) (6232 6368) (6368
 6232) (8128 8128) (10744 10856) (10856 10744) (12285 14595) (14595
 12285))

```

Решение 2.29.2

```

(defun sum-divisors (n)
  (loop for a from 1 to (truncate (/ n 2))
        when (zerop (rem n a))
        sum a))

(defun amicable-nums (m)
  (clear-amicable
   (loop for n from 2 to m when (= (sum-divisors (sum-divisors n)) n)
        collect (list n (sum-divisors n)))))

(defun clear-amicable (w &optional ac &aux (a (sort (car w) #'<)))
  (cond ((null w) (nreverse ac))
        ((or (= (car a) (cadr a)) (member a ac :test #'equalp))
         (clear-amicable (cdr w) ac))
        ((clear-amicable (cdr w) (cons a ac)))))

```

```
> (amicable-nums 15000)
((220 284) (1184 1210) (2620 2924) (5020 5564) (6232 6368) (10744
10856) (12285 14595))
```

Решение 2.29.3

```
(defun sum-divisors (n)
  (loop for a from 1 to (truncate (/ n 2))
        when (zerop (rem n a))
        sum a))

(defun amicable-nums (m)
  (clear-amicable
   (loop for n from 2 to m when (= (sum-divisors (sum-divisors n)) n)
         collect (list n (sum-divisors n)))))

(defun clear-amicable (w &optional ac &aux (a (sort (car w) #'<)))
  (cond ((null w) (nreverse ac))
        ((= (car a) (cadr a)) (clear-amicable (cdr w) ac))
        ((clear-amicable (cdr w) (pushnew a ac :test #'equalp)))))

> (amicable-nums 15000)
((220 284) (1184 1210) (2620 2924) (5020 5564) (6232 6368) (10744
10856) (12285 14595))
```

Решение 2.29.4

```
(defun sum-divisors (n)
  (loop for a from 1 to (truncate (/ n 2))
        when (zerop (rem n a))
        sum a))

(defun amicable-nums (m)
  (remove-duplicates
   (mapcar #'(lambda (a) (sort a #'<))
           (remove-if #'(lambda (e) (= (car e) (cadr e)))
                     (loop for n from 2 to m
                           when (= (sum-divisors (sum-divisors n)) n)
                           collect (list n (sum-divisors n))))))
   :test #'equalp))

> (amicable-nums 15000)
((220 284) (1184 1210) (2620 2924) (5020 5564) (6232 6368) (10744
10856) (12285 14595))
```

* Пары дружественных чисел, меньших 100 000:

220 и 284 (Пифагор, около 500 до н. э.)

1184 и 1210 (Паганини, 1866)

2620 и 2924 (Эйлер, 1747)

5020 и 5564 (Эйлер, 1747)

6232 и 6368 (Эйлер, 1750)

10744 и 10856 (Эйлер, 1747)

12285 и 14595 (Браун, 1939)
 17296 и 18416 (Ибн ал-Банна, около 1300, Фариси, около 1300, Ферма, Пьер, 1636)
 63020 и 76084 (Эйлер, 1747)
 66928 и 66992 (Эйлер, 1750)
 67095 и 71145 (Эйлер, 1747)
 69615 и 87633 (Эйлер, 1747)
 79750 и 88730 (Рольф, 1964)

Задача 2.30 n-234.lisp

Определить функцию, которая для произвольного целого n создает список из трех элементов, а именно: квадрата, куба и четвертой степени числа n .

Решение 2.30.1

```
(defun n-234 (n)
  (list (* n n) (* n n n) (* n n n n)))
```

```
> (n-234 2)
(4 8 16)
```

Решение 2.30.2

```
(defun n-234 (n)
  (list (expt n 2) (expt n 3) (expt n 4)))
```

```
> (n-234 2)
(4 8 16)
```

Решение 2.30.3

```
(defun n-234 (n)
  (mapcar #'(lambda (a) (expt n a)) '(2 3 4)))
```

```
> (n-234 2)
(4 8 16)
```

Решение 2.30.4

```
(defun n-234 (n &rest w)
  (mapcar #'(lambda (a) (expt n a)) w))
```

```
> (n-234 2 2 3 4)
(4 8 16)
```

Решение 2.30.5

```
(defun n-234 (n)
  (loop for a in '(2 3 4) collect (expt n a)))
```

```
> (n-234 2)
(4 8 16)
```

Решение 2.30.6

```
(defun n-234 (n &rest w)
  (loop for a in w collect (expt n a)))
```

```
> (n-234 2 2 3 4)
(4 8 16)
```

Решение 2.30.7

```
(defun n-234 (n m)
  (when (> m 1) (nconc (n-234 n (1- m)) (list (expt n m))))))
```

```
> (n-234 2 4)
(4 8 16)
```

Решение 2.30.8

```
(defun n-234 (n m)
  (when (> m 1) (cons (expt n m) (n-234 n (1- m))))))
```

```
> (n-234 2 4)
(16 8 4)
```

Задача 2.31 *lets-digs.lisp*

Дана строка знаков. Разделить строку на две строки, в первую из которых занести только буквы, а во вторую – только цифры.

Решение 2.31.1

```
(defun lets-digs (s)
  (loop for a across s
    if (alpha-char-p a) collect a into lets
    else if (digit-char-p a) collect a into digs
    finally (return (values (coerce lets 'string)
                           (coerce digs 'string)))))
```

```
> (lets-digs "ywugxas?x1212bjj1130")
"ywugxasxbjj"
"12121130"
```

Задача 2.32 *found-more-than-once.lisp*

Дан простой список атомов. Определить функцию, возвращающую элементы списка, встречающиеся более одного раза.

Решение 2.32.1

```

(defun found-more-than-once (w)
  (found (remove-duplicates w) w))

(defun found (v w)
  (cond ((null v) nil)
        ((= (count (car v) w) 1) (found (cdr v) w))
        (t (cons (car v) (found (cdr v) w)))))

> (found-more-than-once '(a b b c c c))
(B C)

```

Решение 2.32.2

```

(defun found-more-than-once (w)
  (found (remove-duplicates w) w))

(defun found (v w)
  (when v (if (= (count (car v) w) 1)
              (found (cdr v) w)
              (cons (car v) (found (cdr v) w)))))

> (found-more-than-once '(a b b c c c))
(B C)

```

Решение 2.32.3

```

(defun found-more-than-once (w)
  (found (remove-duplicates w) w))

(defun found (v w)
  (remove-if #'(lambda (a) (= (count a w) 1)) v))

> (found-more-than-once '(a b b c c c))
(B C)

```

Решение 2.32.4

```

(defun found-more-than-once (w)
  (found (remove-duplicates w) w))

(defun found (v w)
  (loop for a in v
        when (> (count a w) 1)
        collect a))

> (found-more-than-once '(a b b c c c))
(B C)

```

Решение 2.32.5

```

(defun found-more-than-once (w)
  (found (remove-duplicates w) w))

```

```
(defun found (v w &optional z)
  (cond ((null v) (reverse z))
        ((= (count (car v) w) 1) (found (cdr v) w z))
        ((found (cdr v) w (cons (car v) z)))))

> (found-more-than-once '(a b b c c c))
(B C)
```

Решение 2.32.6

```
(defun found-more-than-once (w)
  (found (remove-duplicates w) w))

(defun found (v w &optional z)
  (if v
      (found (cdr v) w (if (= (count (car v) w) 1)
                           z
                           (cons (car v) z)))
      (reverse z)))

> (found-more-than-once '(a b b c c c))
(B C)
```

Задача 2.33 *same-let-words.lisp*

Определить функцию, которая возвращает все слова из заданного предложения, состоящие из тех же букв, что и первое слово предложения.

Решение 2.33.1

```
(defun same-let (s)
  (mapcar
   #'(lambda (a)
       (list a
             (sort
              (remove-duplicates
               (map 'list #'char-code a))
              #'<)))
   (mapcar
    #'string
    (read-from-string
     (concatenate 'string "(" s ")")))))

(defun word-code (w &aux (a (car w)))
  (mapcar
   #'car
   (remove-if-not
    #'(lambda (e) (equal (cadr e) (cadr a)))
    w)))

(defun same-let-words (s)
```

```
(word-code (same-let s)))

> (same-let-words "aba bab aac aab ab abab")
("ABA" "BAB" "AAB" "AB" "ABAB")
```

Задача 2.34 word-length.lisp

Определить функцию, которая возвращает длину каждого слова в заданной строке.

Решение 2.34.1

```
(defun word-length (s)
  (mapcar
   #'length
   (mapcar
    #'string
    (read-from-string
     (concatenate 'string "(" s ")")))))

> (word-length "a aa aaa")
(1 2 3)
```

Задача 2.35 reaper.lisp

Определить функцию, которая возвращает многоуровневый список из текстового файла, а также список чисел этого списка, количество списков на нулевом уровне, максимальную глубину вложенности и максимальную длину списка на этой глубине.

Решение 2.35.1

```
d:/test.txt
(1 (2 ((3)))
 ((a b c d)
 4 e) 5)

(defun max-depth (w)
  (cond ((null w) 0)
        ((atom (car w)) (max-depth (cdr w)))
        ((max (1+ (max-depth (car w))) (max-depth (cdr w))))))

(defun flat (w &optional acc)
  (cond ((null w) acc)
        ((atom w) (cons w acc))
        ((flat (car w) (flat (cdr w) acc)))))

(defun dive (n w &aux (a (car w)) (d (cdr w)))
  (cond ((null w) nil)
        ((= n 0) (list w))
        ((atom a) (dive n d))))
```



```

      ((nconc (dive (1- n) a) (dive n d))))))
(defun reaper (path &aux u)
  (with-open-file (s path :direction :input)
    (do ((line (read-line s nil :eof) (read-line s nil :eof)))
      ((eql line :eof))
      (setf u (concatenate 'string u line)))
    (setf u (read-from-string u))
    (values u
      (remove-if-not #'numberp (flat u))
      (count-if #'consp u)
      (max-depth u)
      (reduce #'max (mapcar #'length (dive (max-depth u) u)))))))
> (reaper "d:/test.txt")
(1 (2 ((3))) (((A B C D)) 4 E) 5)
(1 2 3 4 5)
2
3
4

```

Решение 2.35.2

```

d:/test.txt
(1 (2 ((3)))
  ((a b c d)
  4 e) 5)

(defun max-depth (w)
  (cond ((null w) 0)
        ((atom (car w)) (max-depth (cdr w)))
        ((max (1+ (max-depth (car w))) (max-depth (cdr w))))))

(defun flat (w)
  (loop for a in w
        if (listp a) append (flat a)
        else collect a))

(defun dive (n w &aux (a (car w)) (d (cdr w)))
  (cond ((null w) nil)
        ((= n 0) (list w))
        ((atom a) (dive n d))
        ((nconc (dive (1- n) a) (dive n d))))))

(defun reaper (path &aux u)
  (with-open-file (s path :direction :input)
    (do ((line (read-line s nil :eof) (read-line s nil :eof)))
      ((eql line :eof))
      (setf u (concatenate 'string u line)))
    (setf u (read-from-string u))
    (values u
      (remove-if-not #'numberp (flat u))
      (count-if #'consp u)

```

```

      (max-depth u)
      (reduce #'max (mapcar #'length (dive (max-depth u) u))))))

> (reaper "d:/test.txt")
(1 (2 ((3))) ((A B C D)) 4 E) 5)
(1 2 3 4 5)
2
3
4

```

Решение 2.35.3

```

d:/test.txt
(1 (2 ((3)))
  ((a b c d)
   4 e) 5)

(defun max-depth (w)
  (cond ((null w) 0)
        ((atom (car w)) (max-depth (cdr w)))
        ((max (1+ (max-depth (car w))) (max-depth (cdr w))))))

(defun flat (w)
  (mapcan #'(lambda (a) (if (listp a) (flat a) `(,a))) w))

(defun dive (n w &aux (a (car w)) (d (cdr w)))
  (cond ((null w) nil)
        ((= n 0) (list w))
        ((atom a) (dive n d))
        ((nconc (dive (1- n) a) (dive n d)))))

(defun reaper (path &aux u)
  (with-open-file (s path :direction :input)
    (do ((line (read-line s nil :eof) (read-line s nil :eof)))
        ((eql line :eof))
        (setf u (concatenate 'string u line)))
    (setf u (read-from-string u))
    (values u
            (remove-if-not #'numberp (flat u))
            (count-if #'consp u)
            (max-depth u)
            (reduce #'max (mapcar #'length (dive (max-depth u) u))))))

> (reaper "d:/test.txt")
(1 (2 ((3))) ((A B C D)) 4 E) 5)
(1 2 3 4 5)
2
3
4

```

Решение 2.35.4

```
d:/test.txt
```

```

(1 (2 ((3)))
((a b c d))
4 e) 5)

(defun max-depth (w)
  (cond ((null w) 0)
        ((atom (car w)) (max-depth (cdr w)))
        ((max (1+ (max-depth (car w))) (max-depth (cdr w))))))

(defun flat (w)
  (typecase w
    (cons (mapcan #'flat w))
    (t `(:,w))))

(defun dive (n w &aux (a (car w)) (d (cdr w)))
  (cond ((null w) nil)
        ((= n 0) (list w))
        ((atom a) (dive n d))
        ((nconc (dive (1- n) a) (dive n d)))))

(defun reaper (path &aux u)
  (with-open-file (s path :direction :input)
    (do ((line (read-line s nil :eof) (read-line s nil :eof)))
        ((eql line :eof))
        (setf u (concatenate 'string u line)))
    (setf u (read-from-string u))
    (values u
            (remove-if-not #'numberp (flat u))
            (count-if #'consp u)
            (max-depth u)
            (reduce #'max (mapcar #'length (dive (max-depth u) u))))))

> (reaper "d:/test.txt")
(1 (2 ((3))) ((A B C D)) 4 E) 5)
2
3
4

```

Решение 2.35.5

```

d:/test.txt
(1 (2 ((3)))
((a b c d))
4 e) 5)

(defun max-depth (w)
  (cond ((null w) 0)
        ((atom (car w)) (max-depth (cdr w)))
        ((max (1+ (max-depth (car w))) (max-depth (cdr w))))))

(defun flat (w)
  (if (atom w) `(:,w) (mapcan #'flat w)))

```

```

(defun dive (n w &aux (a (car w)) (d (cdr w)))
  (cond ((null w) nil)
        ((= n 0) (list w))
        ((atom a) (dive n d))
        ((nconc (dive (1- n) a) (dive n d)))))

(defun reaper (path &aux u)
  (with-open-file (s path :direction :input)
    (do ((line (read-line s nil :eof) (read-line s nil :eof)))
        ((eql line :eof))
      (setf u (concatenate 'string u line)))
    (setf u (read-from-string u))
    (values u
            (remove-if-not #'numberp (flat u))
            (count-if #'consp u)
            (max-depth u)
            (reduce #'max (mapcar #'length (dive (max-depth u) u))))))

> (reaper "d:/test.txt")
(1 (2 ((3))) ((A B C D)) 4 E) 5)
(1 2 3 4 5)
2
3
4

```

Решение 2.35.6

```

d:/test.txt
(1 (2 ((3)))
  ((a b c d)
  4 e) 5)

(defun max-depth (w)
  (cond ((null w) 0)
        ((atom (car w)) (max-depth (cdr w)))
        ((max (1+ (max-depth (car w))) (max-depth (cdr w))))))

(defun numbers (w)
  (loop for a in w
        if (consp a) nconc (numbers a)
        else when (numberp a) collect a))

(defun dive (n w &aux (a (car w)) (d (cdr w)))
  (cond ((null w) nil)
        ((zerop n) (list w))
        ((atom a) (dive n d))
        ((nconc (dive (1- n) a) (dive n d)))))

(defun reaper (path &aux u)
  (with-open-file (s path :direction :input)
    (do ((line (read-line s nil :eof) (read-line s nil :eof)))
        ((eql line :eof))

```

```

      (setf u (concatenate 'string u line)))
    (setf u (read-from-string u))
    (values u
      (numbers u)
      (count-if #'consp u)
      (max-depth u)
      (loop for a in (dive (max-depth u) u)
        maximize (length a))))

> (reaper "d:/test.txt")
(1 (2 ((3))) (((A B C D)) 4 E) 5)
(1 2 3 4 5)
2
3
4

```

Задача 2.36 *prime-divisors.lisp*

There is seldom counter-intuitives in lisp programming, instead, you can focus on the abstraction of your program, which maybe one of the most important part of programming.

Yushi Wang

Определить функцию, которая возвращает все простые делители числа n .

Решение 2.36.1

```

(defun prime-divisors (n)
  (loop for a from 2 to n
    when (and (zerop (rem n a)) (primep a))
    collect a))

(defun primep (n)
  (loop for i from 2 to (isqrt n) never (zerop (rem n i))))

> (prime-divisors 1011234)
(2 3 7 24077)

```

Задача 2.37 *reversible-numbers.lisp*

Некоторые положительные целые числа n обладают следующим свойством: сумма (n + перевернутое n) состоит исключительно из нечетных цифр. К примеру, $36 + 63 = 99$ и $409 + 904 = 1313$. Такие числа будем называть переворачиваемыми. Таким образом, 36, 63, 409, и 904 являются переворачиваемыми. Ни n , ни перевернутое(n) не могут начинаться с нулей. Вывести список переворачиваемых чисел, существующих в пределах одной тысячи.

Решение 2.37.1

```

(defun reversible-numbers (n)
  (loop for a from 1 to n
        unless
          (or
            (zerop (rem a 10))
            (some
              #'evenp
              (int-digit
                (+ (reverse-digits a)
                  a))))
          collect a))

(defun int-digit (n &optional ac)
  (if (zerop n)
      ac
      (int-digit (truncate n 10) (cons (rem n 10) ac))))

(defun reverse-digits (n &optional (m 0))
  (cond ((zerop n) m)
        (t (multiple-value-bind (q r) (truncate n 10)
              (reverse-digits q (+ (* m 10) r))))))

> (reversible-numbers 1000)
(12 14 16 18 21 23 25 27 32 34 36 41... 932 934 936 938 942 944 946 948)

```

Задача 2.38 *symb-freq-percent.lisp*

Lisp is a language that's best learned by people who've never programmed before.

Mark Miller

Дан текстовый файл. Определить функцию, возвращающую для каждого символа количество его появлений в тексте и процент от общего числа символов.

Решение 2.38.1

```

d:/test.txt
ab123 cd456
abcd1 e4567
aaad8 aa999

(defun symb-freq-percent (path &aux u)
  (with-open-file (s path :direction :input)
    (do ((line (read-line s nil :eof) (read-line s nil :eof)))
        ((eql line :eof))
        (setf u (concatenate 'string u line)))
    (count-symb u)))

(defun count-symb (s

```

```

&aux
(w (loop for c across s collect (string c)))
(v (remove-duplicates w :test #'equal))
(m (length w))
(z (loop for a in v
        collect (count a w :test #'equal))))
(loop for a in v
      for b in z
      collect (list a b (percent b m)))

(defun percent (n m)
  (float (/ (* n 100) m)))

> (symb-freq-percent "d:/test.txt")
(("2" 1 3.030303) ("3" 1 3.030303) ("b" 2 6.060606) ("c" 2 6.060606)
 ("1" 2 6.060606) ("e" 1 3.030303) ("4" 2 6.060606) ("5" 2 6.060606)
 ("6" 2 6.060606) ("7" 1 3.030303) ("d" 3 9.090909) ("8" 1 3.030303)
 (" " 3 9.090909) ("a" 7 21.212122) ("9" 3 9.090909))

```

Решение 2.38.2

```

d:/test.txt
ab123 cd456
abcd1 e4567
aad8 aa999

(defun symb-freq-percent (path &aux u)
  (with-open-file (s path :direction :input)
    (do ((line (read-line s nil :eof) (read-line s nil :eof)))
        ((eql line :eof))
        (setf u (concatenate 'string u line)))
    (count-symb u)))

(defun count-symb (s
  &aux
    (w (loop for c across s collect (string c)))
    (v (remove-duplicates w :test #'equal))
    (m (length w))
    (z (loop for a in v
            collect (count a w :test #'equal))))
  (loop for a in v
        for b in z
        collect (list (char a 0) b (/ (* b 100.0) m))))

> (symb-freq-percent "d:/test.txt")
((#\2 1 3.030303) (#\3 1 3.030303) (#\b 2 6.060606) (#\c 2 6.060606)
 (#\1 2 6.060606) (#\e 1 3.030303) (#\4 2 6.060606) (#\5 2 6.060606)
 (#\6 2 6.060606) (#\7 1 3.030303) (#\d 3 9.090909) (#\8 1 3.030303)
 (#\Space 3 9.090909) (#\a 7 21.212122) (#\9 3 9.090909))

```

Решение 2.38.3

```

d:/test.txt

```

```
abl23 cd456
abcd1 e4567
aaad8 aa999
```

```
(defun symb-freq-percent (path &aux ac)
  (with-open-file (s path)
    (loop with u = (make-hash-table)
      for c = (read-char s nil nil)
      for n upfrom 0 while c do
        (incf (gethash c u 0))
      finally
        (maphash #'(lambda (a b)
                      (push (list a b (percent b n)) ac))
                  u)))
  ac)

(defun percent (n m)
  (float (/ (* n 100) m)))
```

```
> (symb-freq-percent "d:/test.txt")
((#\a 7 20.0) (#\b 2 5.714286) (#\1 2 5.714286) (#\2 1 2.857143) (#\3
1 2.857143) (#\Space 3 8.571428) (#\c 2 5.714286) (#\d 3 8.571428)
(#\4 2 5.714286) (#\5 2 5.714286) (#\6 2 5.714286) (#\Newline 2
5.714286) (#\e 1 2.857143) (#\7 1 2.857143) (#\8 1 2.857143) (#\9 3
8.571428))
```

Решение 2.38.4

```
d:/test.txt
abl23 cd456
abcd1 e4567
aaad8 aa999
```

```
(defun symb-freq-percent (path &aux ac)
  (with-open-file (s path)
    (loop with u = (make-hash-table)
      for c = (read-char s nil nil)
      for n upfrom 0 while c do
        (incf (gethash c u 0))
      finally
        (maphash #'(lambda (a b)
                      (push (list a b (/ (* b 100.0) n))
                          ac))
                  u)))
  ac)
```

```
> (symb-freq-percent "d:/test.txt")
((#\a 7 20.0) (#\b 2 5.714286) (#\1 2 5.714286) (#\2 1 2.857143) (#\3
1 2.857143) (#\Space 3 8.571428) (#\c 2 5.714286) (#\d 3 8.571428)
(#\4 2 5.714286) (#\5 2 5.714286) (#\6 2 5.714286) (#\Newline 2
5.714286) (#\e 1 2.857143) (#\7 1 2.857143) (#\8 1 2.857143) (#\9 3
8.571428))
```


Задача 2.39 *explode.lisp*

Дан символ. Превратить его в список составляющих его цифр и символов .

Решение 2.39.1

```
(defun explode (a)
  (mapcar #'read-from-string
    (map 'list #'string (write-to-string a))))
```

```
> (explode 'abc12)
(A B C 1 2)
```

Решение 2.39.2

```
(defun explode (a)
  (loop for c across (write-to-string a)
    collect (read-from-string (string c))))
```

```
> (explode 'abc12)
(A B C 1 2)
```

Задача 2.40 *fibonacci.lisp*

Определить функцию, возвращающую ряд чисел Фибоначчи от 1 до n.

Решение 2.40.1

```
(defun fibonacci (z &optional (n 0) (m 1))
  (if (>= m z) nil (cons m (fibonacci z m (+ n m)))))
```

```
> (fibonacci 1000)
(1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987)
```

Решение 2.40.2

```
(defun fibonacci (z &optional (n 0) (m 1))
  (when (< m z) (cons m (fibonacci z m (+ n m)))))
```

```
> (fibonacci 1000)
(1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987)
```

Задача 2.41 *count-letters.lisp*

Определить функцию, которая по заданной строке строит список пар: (<буква> <частота встречаемости в тексте>).

Решение 2.41.1

```
(defun count-letters (s &aux (z (remove-duplicates s)))
  (loop for a across z collect (list a (count a s))))
```

```
> (count-letters "aaabbc")
((#\a 3) (#\b 2) (#\c 1))
```

Решение 2.41.2

```
(defun count-letters (s &aux (z (remove-duplicates s)))
  (loop for a across z
    collect `(,a ,(count a s))))

> (count-letters "aaabbc")
((#\a 3) (#\b 2) (#\c 1))
```

Задача 2.42 100m.lisp

Дан результат ямайского легкоатлета Усейна Болта, обладателя мирового рекорда в беге на 100 метров – 9,58 с. Вернуть список из мирового рекорда и скорости бега.

Решение 2.42.1

```
(defun 100m (s) (list s (float (/ 100 s))))

> (100m 9.58)
(9.58 10.438414)
```

Задача 2.43 200m.lisp

Дан результат ямайского легкоатлета Усейна Болта, обладателя мирового рекорда в беге на 200 метров – 19,19 с. Вернуть список из мирового рекорда и скорости бега.

Решение 2.43.1

```
(defun 200m (s) (list s (float (/ 200 s))))

> (200m 19.19)
(19.19 10.422094)
```

Задача 2.44 400m.lisp

Дан результат американского легкоатлета Майкла Джонсона, обладателя мирового рекорда в беге на 400 метров – 43,18 с. Вернуть список из мирового рекорда и скорости бега.

Решение 2.44.1

```
(defun 400m (s)
  (list s (float (/ 400 s))))

> (400m 43.18)
```

```
(43.18 9.263548)
```

Задача 2.45 square-digits.lisp

Среди двузначных чисел найдите те, сумма квадратов цифр которых делится на n .

Решение 2.45.1

```
(defun square-digits (n)
  (loop for a from 1 to 9
        nconc (loop for b from 0 to 9
                     when (zerop (rem (+ (* a a) (* b b)) n))
                     collect (+ (* a 10) b))))

> (square-digits 13)
(15 18 23 32 46 47 51 64 69 74 79 81 96 97)
```

Решение 2.45.2

```
(defun square-digits (n)
  (loop for a from 10 to 99
        when (zerop (rem (+ (expt (floor a 10) 2)
                             (expt (mod a 10) 2)))
              n))
        collect a))

> (square-digits 13)
(15 18 23 32 46 47 51 64 69 74 79 81 96 97)
```

Задача 2.46 digi-div-n.lisp

Найти все натуральные числа до n включительно, которые делятся без остатка на каждую из цифр числа n .

Решение 2.46.1

```
(defun int-digits (n)
  (loop for c across (write-to-string n)
        collect (digit-char-p c)))

(defun digi-div-n (n &aux (w (int-digits n)))
  (loop for a from 1 to n
        when (and (notany #'zerop w)
                  (loop for e in w
                        always (zerop (mod a e))))
        collect a))

> (digi-div-n 129)
(18 36 54 72 90 108 126)
> (digi-div-n 130)
```

NIL

Решение 2.46.2

```

(defun int-digits (n &optional ac)
  (if (zerop n)
      ac
      (int-digits (truncate n 10) (cons (rem n 10) ac))))

(defun check (z n &aux (w (int-digits n)))
  (and (notany #'zerop w)
       (loop for e in w
             always (zerop (mod z e)))))

(defun digi-div-n (n &optional (z 1))
  (cond ((> z n) nil)
        ((check z n) (cons z (digi-div-n n (1+ z))))
        ((digi-div-n n (1+ z)))))

> (digi-div-n 129)
(18 36 54 72 90 108 126)
> (digi-div-n 130)
NIL

```

Решение 2.46.3

```

(defun int-digits (n)
  (loop with r do (setf (values n r) (truncate n 10))
        collect r into ac until (zerop n)
        finally (return (nreverse ac))))

(defun check (z n &aux (w (int-digits n)))
  (and (notany #'zerop w)
       (loop for e in w
             always (zerop (mod z e)))))

(defun digi-div-n (n &optional (z 1))
  (cond ((> z n) nil)
        ((check z n) (cons z (digi-div-n n (1+ z))))
        ((digi-div-n n (1+ z)))))

> (digi-div-n 129)
(18 36 54 72 90 108 126)
> (digi-div-n 130)
NIL

```

Задача 2.47 digi-div.lisp

Найти все натуральные числа до n включительно, которые делятся без остатка на каждую из своих цифр.

Решение 2.47.1

```
(defun int-digits (n)
  (loop for c across (write-to-string n) collect (digit-char-p c)))

(defun digi-div (n)
  (loop for a from 1 to n
        when (and (notany #'zerop (int-digits a))
                  (loop for e in (int-digits a)
                        always (zerop (mod a e))))
        collect a))

> (digi-div 129)
(1 2 3 4 5 6 7 8 9 11 12 15 22 24 33 36 44 48 55 66 77 88 99 111 112
115 122 124 126 128)
```

Решение 2.47.2

```
(defun int-digits (n)
  (loop for c across (write-to-string n) collect (digit-char-p c)))

(defun digi-div (n)
  (loop for a from 1 to n for w = (int-digits a)
        when (and (notany #'zerop w)
                  (loop for e in w
                        always (zerop (mod a e))))
        collect a))

> (digi-div 129)
(1 2 3 4 5 6 7 8 9 11 12 15 22 24 33 36 44 48 55 66 77 88 99 111 112
115 122 124 126 128)
```

Решение 2.47.3

```
(defun int-digits (n)
  (loop for c across (write-to-string n) collect (digit-char-p c)))

(defun digi-div (n)
  (remove-if-not
   #'(lambda (a)
       (and (notany #'zerop (int-digits a))
            (loop for e in (int-digits a)
                  always (zerop (mod a e)))))
   (loop for a from 1 to n collect a)))

> (digi-div 129)
(1 2 3 4 5 6 7 8 9 11 12 15 22 24 33 36 44 48 55 66 77 88 99 111 112
115 122 124 126 128)
```

Решение 2.47.4

```
(defun int-digits (n)
  (loop for c across (write-to-string n) collect (digit-char-p c)))
```

```
(defun digi-div (n)
  (remove-if-not
    #'(lambda (a &aux (w (int-digits a)))
      (and (notany #'zerop w)
        (loop for e in w
          always (zerop (mod a e))))))
    (loop for a from 1 to n collect a)))

> (digi-div 129)
(1 2 3 4 5 6 7 8 9 11 12 15 22 24 33 36 44 48 55 66 77 88 99 111 112
115 122 124 126 128)
```

Решение 2.47.5

```
(defun int-digits (n)
  (loop with r do (setf (values n r) (truncate n 10))
    collect r into ac until (zerop n)
    finally (return (nreverse ac))))

(defun check (n &aux (w (int-digits n)))
  (and (notany #'zerop w)
    (loop for e in w
      always (zerop (mod n e)))))

(defun digi-div (n)
  (remove-if-not
    #'(lambda (a) (check a))
    (loop for a from 1 to n collect a)))

> (digi-div 129)
(1 2 3 4 5 6 7 8 9 11 12 15 22 24 33 36 44 48 55 66 77 88 99 111 112
115 122 124 126 128)
```

Решение 2.47.6

```
(defun int-digits (n &optional ac)
  (if (zerop n)
    ac
    (int-digits (truncate n 10) (cons (rem n 10) ac))))

(defun check (n &aux (w (int-digits n)))
  (and (notany #'zerop w)
    (loop for e in w
      always (zerop (mod n e)))))

(defun digi-div (n &optional (z 1))
  (cond ((> z n) nil)
    ((check z) (cons z (digi-div n (1+ z))))
    ((digi-div n (1+ z)))))

> (digi-div 129)
(1 2 3 4 5 6 7 8 9 11 12 15 22 24 33 36 44 48 55 66 77 88 99 111 112
115 122 124 126 128)
```

Задача 2.48 fibonacci-sequence.lisp

Определить функцию, возвращающую n чисел Фибоначчи.

Решение 2.48.1

```
(defun fibonacci-sequence (z
                           &optional
                           (n 0)
                           (m 1)
                           (k 0))
  (if (= k z)
      nil
      (cons m
             (fibonacci-sequence z
                                  m
                                  (+ n m)
                                  (1+ k))))))

> (fibonacci-sequence 20)
(1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765)
```

Решение 2.48.2

```
(defun fibonacci-sequence (z &optional (n 0) (m 1) (k 0))
  (when (< k z)
    (cons m (fibonacci-sequence z m (+ n m) (1+ k)))))

> (fibonacci-sequence 20)
(1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765)
```

Решение 2.48.3

```
(defun fibonacci-sequence (z &optional (n 0) (m 1))
  (when (plusp z)
    (cons m (fibonacci-sequence (1- z) m (+ n m)))))

> (fibonacci-sequence 20)
(1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765)
```

Решение 2.48.4

```
(defun fibonacci-sequence (n)
  (loop repeat n
        for a = 0 then b
        and for b = 1 then (+ a b)
        collect b))

> (fibonacci-sequence 20)
(1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765)
```

Задача 2.49 doors.lisp

There is no single "most powerful" programming language - but Lisp's claim to that title is its homoiconic feature, its code is data.

Miguel Paraz

You have n doors in a row that are all initially closed. You make n passes by the doors. The first time through, you visit every door and toggle the door (if the door is closed, you open it; if it is open, you close it). The second time you only visit every 2nd door (door #2, #4, #6, ...). The third time, every 3rd door (door #3, #6, #9, ...), etc, until you only visit the n -th door. What state are the doors in after the last pass? Which are open, which are closed?

Имеется n пронумерованных дверей — 1 .. n . Изначально все двери закрыты. 1-й шаг: каждая дверь меняет своё состояние (если открыта — закрывается и наоборот). 2-й шаг: каждая 2-я дверь меняет своё состояние. 3-й шаг: каждая 3-я дверь меняет своё состояние. ... n -й шаг: каждая n -я дверь меняет своё состояние. Вопрос: какие двери будут открыты (закрыты) через n шагов?

Решение 2.49.1

```
(defun toggle (w m z)
  (loop for a in w for n from 1 to z
        collect (if (zerop (mod n m)) (not a) a)))

(defun doors (z &optional (w (make-list z)) (n 1))
  (if (> n z) w (doors z (toggle w n z) (1+ n))))

> (doors 100)
(T NIL NIL T NIL NIL NIL NIL T NIL NIL NIL NIL NIL NIL T NIL NIL NIL
NIL NIL NIL NIL NIL T NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL T NIL
NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL T NIL NIL NIL NIL NIL NIL
NIL NIL NIL NIL NIL NIL NIL NIL T NIL NIL NIL NIL NIL NIL NIL NIL NIL
NIL NIL NIL NIL NIL NIL NIL NIL T NIL NIL NIL NIL NIL NIL NIL NIL NIL
NIL NIL NIL NIL NIL NIL NIL NIL T NIL NIL NIL NIL NIL NIL NIL NIL NIL)
```

Решение 2.49.2

```
(defun doors (n)
  (loop for a from 1 to n collect
        (if (zerop (mod (sqrt a) 1)) t nil)))

> (doors 100)
(T NIL NIL T NIL NIL NIL NIL T NIL NIL NIL NIL NIL T NIL NIL NIL
NIL NIL NIL NIL NIL T NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL T NIL
NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL T NIL NIL NIL NIL NIL NIL
NIL NIL NIL NIL NIL NIL NIL NIL T NIL NIL NIL NIL NIL NIL NIL NIL NIL
NIL NIL NIL NIL NIL NIL NIL NIL T NIL NIL NIL NIL NIL NIL NIL NIL NIL)
```



```
NIL NIL NIL NIL NIL NIL NIL T NIL NIL NIL NIL NIL NIL NIL NIL NIL
NIL NIL NIL NIL NIL NIL NIL NIL NIL T)
```

Задача 2.50 count-word.lisp

Lisp is a language unique to all programming languages - in that it's both powerful (in the sense that it makes your task a lot less sweat-invoking) and freeing (in the sense that it never tries to force you into a specific way). However, this also means your thought process when starting a program is a lot longer than most others - from many more choices comes many more alternatives, thus much more to think about.

Irné Barnard

Написать программу, которая по заданному тексту строит список пар: (<слово> <частота встречаемости в тексте>).

Решение 2.50.1

```
(defun count-word (pathname)
  (let ((u)) (with-open-file (s pathname :direction :input)
    (do ((c (read-line s nil :eof) (read-line s nil :eof)))
      ((eql c :eof))
      (setq u (nconc (en c) u))))
    (qu u)))

(defun en (s)
  (read-from-string
   (concatenate
    'string "(" (delete-if-not #'(lambda (x)
                                   (or (alpha-char-p x)
                                       (equal x #\space)
                                       (equal x #\-)))
                                s) ")"))))

(defun qu (w)
  (when w (let ((a (car w)) (d (cdr w)))
    (cons (cons (string-downcase a) (count a w))
          (qu (delete a d))))))

> (count-word "test.txt")
(("less" . 5) ("de-facto" . 1))
```

Решение 2.50.2

```
(defun count-word (pathname)
  (let ((u)) (with-open-file (s pathname :direction :input)
    (do ((c (read-line s nil :eof) (read-line s nil :eof)))
```

```

        ((eql c :eof))
        (setq u (nconc (en c) u))))
    (symb-str u)))

(defun en (s)
  (read-from-string
   (concatenate
    'string "(" (delete-if-not #'(lambda (x)
                                   (or (alpha-char-p x)
                                       (equal x #\space)
                                       (equal x #\-)))
                                s) ")"))))

(defun symb-str (w)
  (let ((u) (dolist (n w)
                    (push (string-downcase n) u)
                    (count-str (sort u #'string-greaterp)))))

  u))

(defun count-str (w)
  (let ((u) (e (car w)) (n 1))
    (dolist (m w)
      (cond ((equal m e) (incf n))
            ((push (cons e n) u) (setq n 1 e m))))
    u))

> (count-word "test.txt")
(("along" . 3) ("already" . 3))

```

Решение 2.50.3

```

(defun count-word (pathname)
  (let ((u) (with-open-file (s pathname :direction :input)
                (do ((c (read-line s nil :eof) (read-line s nil :eof)))
                    ((eql c :eof))
                    (setq u (nconc (list-symb (drop-noise c)) u))))
        (list-pair (list-str u)))))

(defun list-symb (s)
  (read-from-string (concatenate 'string "(" s ")")))

(defun drop-noise (s)
  (delete-if-not #'(lambda (x) (or
                                (alpha-char-p x)
                                (equal x #\space)
                                (equal x #\-))) s))

(defun list-str (w)
  (sort (mapcar #'(lambda (a) (string-downcase a)) w) #'string-lessp))

(defun list-pair (w &optional (e (car w)) (n 0) u)
  (mapcan #'(lambda (a) (incf n) (when (string-not-equal a e)
                                     (setq u (cons e n) n 0 e a)
                                     (list u))))

```

```

w))

> (count-word "test.txt")
(("balance" . 1) ("based" . 1))

Решение 2.50.4 * mapcan-version

(defun count-word (pathname)
  (let ((u)) (with-open-file (s pathname :direction :input)
    (do ((c (read-line s nil :eof) (read-line s nil :eof)))
      ((eql c :eof))
      (setq u (nconc (list-symb (drop-noise c)) u))))
    (list-pair (sort u #'string<)))))

(defun list-symb (s)
  (read-from-string (concatenate 'string "(" s ")"))))

(defun drop-noise (s)
  (delete-if-not #'(lambda (x) (or
    (alpha-char-p x)
    (equal x #\space)
    (equal x #\-))) s))

(defun list-pair (w &optional (e (car w)) (n 0) u)
  (mapcan #'(lambda (a) (incf n) (unless (eql a e)
    (setq u (cons e n) n 0 e a)
    (list u)))
    w))

> (count-word "test.txt")
(LEGAL . 1) (LEGALLY . 2)

```

Решение 2.50.5 * maphash-version

```

(defun count-word (pathname)
  (with-open-file (s pathname :direction :input)
    (loop for line = (read-line s nil nil)
      while line
      nconc (list-symb (drop-noise line)) into words
      finally (return (sort (list-pair words) #'> :key #'cdr)))))

(defun list-symb (s)
  (read-from-string (concatenate 'string "(" s ")"))))

(defun drop-noise (s)
  (delete-if-not #'(lambda (x) (or
    (alpha-char-p x)
    (equal x #\space)
    (equal x #\-))) s))

(defun list-pair (words &aux (hash (make-hash-table)) acc)
  (dolist (word words) (incf (gethash word hash 0)))
  (maphash #'(lambda (e n) (push `(,e . ,n) acc)) hash acc)

```

```
> (count-word "test.txt")
((ANNEXED . 1))
```

Задача 2.51 run-test.lisp

Есть набранная программа, она записана в отдельном файле. К ней есть тесты. Они записаны в другом файле. Как (что надо в программу добавить или может как-нибудь по другому не меняя саму программу) сделать так, чтоб подгрузились тесты из файла и по нажатию ENTER мы увидели на экране интерпретатора их выполнение? Т.е надо сделать так чтоб тесты не вводились в самом интерпретаторе, а брались из текстового файла и по нажатию ENTER они выполнялись.

Решение 2.51.1

test.txt

```
a a b c
1 1 2 3
```

```
(defun run-test (path)
  (with-open-file (s path :direction :input)
    (do ((line (read-line s nil :eof)(read-line s nil :eof)))
        ((eql line :eof))
        (format t "~a~%" (rm (sm line))))))
```

```
(defun sm (s)
  (read-from-string (concatenate 'string "(" s ")")))
```

```
(defun rm (v)
  (remove (car v) (cdr v)))
```

```
> (run-test "c:/test.txt")
(B C)
(2 3)
```

Решение 2.51.2

test.txt

```
1 2 3 4 kill 1 2
a b c d kill a b
```

```
(defun run-test (path)
  (with-open-file (s path :direction :input)
    (do ((line (read-line s nil :eof)(read-line s nil :eof)))
        ((eql line :eof))
        (format t "~a~%" (rm (sm line))))))
```

```
(defun sm (s)
  (read-from-string (concatenate 'string "(" s ")")))
```

```
(defun rm (w &aux (p (position 'kill w)))
  (nset-difference (subseq w 0 p) (subseq w (1+ p))))

> (run-test "c:/test.txt")
(3 4)
(C D)
```

Задача 2.52 1357...2*n-1.lisp

Определить функцию, которая строит список $(1\ 3\ 5\ 7\ \dots\ 2 * n - 1)$ для заданного n .

Решение 2.52.1

```
(defun 1357...2*n-1 (n &optional ac)
  (if (zerop n) ac (1357...2*n-1 (1- n) (cons (1- (* n 2)) ac))))

> (1357...2*n-1 7)
(1 3 5 7 9 11 13)
```

Решение 2.52.2

```
(defun 1357...2*n-1 (n) (f n nil))

(defun f (n ac) (if (zerop n) ac (f (1- n) (cons (1- (* n 2)) ac))))

> (1357...2*n-1 7)
(1 3 5 7 9 11 13)
```

Решение 2.52.3

```
(defun 1357...2*n-1 (n &optional (m 1))
  (if (< n m) nil (cons (1- (* m 2)) (1357...2*n-1 n (1+ m)))))

> (1357...2*n-1 7)
(1 3 5 7 9 11 13)
```

Решение 2.52.4

```
(defun 1357...2*n-1 (n &optional (m 1))
  (when (>= n m) (cons (1- (* m 2)) (1357...2*n-1 n (1+ m)))))

> (1357...2*n-1 7)
(1 3 5 7 9 11 13)
```

Решение 2.52.5

```
(defun 1357...2*n-1 (n) (f n 1))

(defun f (n m) (when (>= n m) (cons (1- (* m 2)) (f n (1+ m)))))
```

```
> (1357...2*n-1 7)
(1 3 5 7 9 11 13)
```

Решение 2.52.6

```
(defun 1357...2*n-1 (n) (loop for a from 1 to n collect (1- (* a 2))))

> (1357...2*n-1 7)
(1 3 5 7 9 11 13)
```

Задача 2.53 *product-odd-number-odd.lisp*

Определить функцию, создающую массив n случайных чисел от 1 до 100 и вычисляющую произведение нечётных элементов массива их количество.

Решение 2.53.1

```
(defun product-odd-number-odd (n
                                &aux
                                (a (make-array n))
                                (p 1)
                                (m 0))
  (loop for i upto (1- n)
        do (setf (aref a i) (random 100)))
  (dotimes (i n (values p m))
    (if (oddp (aref a i))
        (setf m (1+ m) p (* p (aref a i))))))

> (product-odd-number-odd 15)
474308806687340625
10
```

Задача 2.54 *cubed.lisp*

Представить число в виде суммы кубов трех чисел.

Решение 2.54.1

```
(defun cubed (n &aux (m (floor (expt n 1/3))))
  (loop for a from 1 to m nconc
        (loop for b from a to m nconc
              (loop for c from b to m
                    when (zerop (- n
                                    (expt a 3)
                                    (expt b 3)
                                    (expt c 3)))
                      collect (list a b c)))))

> (cubed 216)
((3 4 5))
```

Задача 2.55 collect-lowercase.lisp

Lisp macros teach you that with a very simple superficial syntax restriction to be applied across a family of syntaxes, you can build whatever syntaxes you want within that family and give them your own meaning. In other words you can easily create a language specific for your problem and then solve the problem in that language. This kind of trick is not possible in all languages. It is possible in homoiconic languages (like Lisp and all languages in the Lisp family) and, to a more limited extent, in a few other. A somewhat famous case is Ruby, where the syntax is flexible enough that it can be reused in non trivial ways by projects like Ruby On Rails. But homoiconicity a-la-Lisp is far more powerful. For complex (and sometimes not so complex, but very common) idioms and problems, the Lisp way is to code the solution of the problem and then also code a special syntax that invokes that code in a way that makes using that solution something very readable and maintainable.

Andrea Ferro

Дан текст из цифр, прописных и строчных букв латинского алфавита, за которым следует точка. Определить функцию, возвращающую список строчных букв текста.

Решение 2.55.1

```
(defun collect-lowercase (s)
  (loop for c across s when (char<= #\a c #\z) collect c))

> (collect-lowercase "55stErLing 55stERLing.")
(#\s #\t #\r #\i #\n #\g #\s #\t #\i #\n #\g)
```

Решение 2.55.2

```
(defun collect-lowercase (s)
  (mapcan #'(lambda (c) (when (char<= #\a c #\z) (list c)))
    (loop for c across s collect c)))

> (collect-lowercase "55stErLing 55stERLing.")
(#\s #\t #\r #\i #\n #\g #\s #\t #\i #\n #\g)
```

Решение 2.55.3

```
(defun collect-lowercase (s)
```

```
(mapcan #'(lambda (c) (and (char<= #\a c #\z) ` (,c)))
  (loop for c across s collect c)))

> (collect-lowercase "55stErLing 55stERLing.")
(#\s #\t #\r #\i #\n #\g #\s #\t #\i #\n #\g)
```

Решение 2.55.4

```
(defun collect-lowercase (s)
  (loop for c across
        (remove-if-not #'(lambda (c) (char<= #\a c #\z)) s)
        collect c))

> (collect-lowercase "55stErLing 55stERLing.")
(#\s #\t #\r #\i #\n #\g #\s #\t #\i #\n #\g)
```

Решение 2.55.5

```
(defun collect-lowercase (s)
  (coerce (remove-if-not #'(lambda (c) (char<= #\a c #\z))
                        s)
          'list))

> (collect-lowercase "55stErLing 55stERLing.")
(#\s #\t #\r #\i #\n #\g #\s #\t #\i #\n #\g)
```

Решение 2.55.6

```
(defun collect-lowercase (s)
  (map 'list #'identity (remove-if-not
                        #'(lambda (c) (char<= #\a c #\z))
                        s)))

> (collect-lowercase "55stErLing 55stERLing.")
(#\s #\t #\r #\i #\n #\g #\s #\t #\i #\n #\g)
```

Решение 2.55.7

```
(defun collect-lowercase (s)
  (concatenate 'list (remove-if-not
                    #'(lambda (c) (char<= #\a c #\z))
                    s)))

> (collect-lowercase "55stErLing 55stERLing.")
(#\s #\t #\r #\i #\n #\g #\s #\t #\i #\n #\g)
```

Решение 2.55.8

```
(defun collect-lowercase (s)
  (loop for c across s when (lower-case-p c) collect c))

> (collect-lowercase "55stErLing 55stERLing.")
(#\s #\t #\r #\i #\n #\g #\s #\t #\i #\n #\g)
```


Решение 2.55.9

```
(defun collect-lowercase (s)
  (mapcan #'(lambda (c) (when (lower-case-p c) (list c)))
    (loop for c across s collect c)))

> (collect-lowercase "55stErLing 55stERLing.")
(#\s #\t #\r #\i #\n #\g #\s #\t #\i #\n #\g)
```

Решение 2.55.10

```
(defun collect-lowercase (s)
  (mapcan #'(lambda (c) (and (lower-case-p c) `(,c)))
    (loop for c across s collect c)))

> (collect-lowercase "55stErLing 55stERLing.")
(#\s #\t #\r #\i #\n #\g #\s #\t #\i #\n #\g)
```

Решение 2.55.11

```
(defun collect-lowercase (s)
  (loop for c across (remove-if-not #'lower-case-p s)
    collect c))

> (collect-lowercase "55stErLing 55stERLing.")
(#\s #\t #\r #\i #\n #\g #\s #\t #\i #\n #\g)
```

Решение 2.55.12

```
(defun collect-lowercase (s)
  (coerce (remove-if-not #'lower-case-p s) 'list))

> (collect-lowercase "55stErLing 55stERLing.")
(#\s #\t #\r #\i #\n #\g #\s #\t #\i #\n #\g)
```

Решение 2.55.13

```
(defun collect-lowercase (s)
  (map 'list #'identity (remove-if-not #'lower-case-p s)))

> (collect-lowercase "55stErLing 55stERLing.")
(#\s #\t #\r #\i #\n #\g #\s #\t #\i #\n #\g)
```

Решение 2.55.14

```
(defun collect-lowercase (s)
  (concatenate 'list (remove-if-not #'lower-case-p s)))

> (collect-lowercase "55stErLing 55stERLing.")
(#\s #\t #\r #\i #\n #\g #\s #\t #\i #\n #\g)
```

Задача 2.56 roman-alpha.lisp

There is a simple and elegant answer to this question: Just learn Common Lisp well first. New languages are exciting to people who know mostly new languages, so learn an old language before you learn new ones and get out of the maelstrom that will drown you in ever new languages that add nothing at all except some miniscule additional feature from another language that someone needed to make a whole new language to implement because he did not know Common Lisp to begin with. A "new" language that differs from the rest of the crop by one or a couple features is proof positive that both what it came from and what it has become are mutations about to die. There are tens if not hundreds of thousands of such "languages" that people have invented over the years, for all sorts of weird purposes where they just could not use whatever language they were already using, could not extend it, and could not fathom how to modify its tools without making a whole new language. They never stopped to think about how horribly wasteful this is, they just went on to create yet another language called Dodo, the Titanic, Edsel, Kyoto-agreement...

Erik Naggum

Создать программу, которая считывает произвольный текст и распечатывает в алфавитном порядке все латинские буквы, входящие в этот текст.

```
d:/test.txt
abracadabra 477
abracadabra 477
```

Решение 2.56.1

```
(defun roman-alpha (path &aux u)
  (with-open-file (s path :direction :input)
    (do ((line (read-line s nil :eof) (read-line s nil :eof)))
        ((eql line :eof))
        (setf u (concatenate 'string u line)))
    (roman-alpha* u)))

(defun roman-alpha* (s)
  (sort (remove-if-not #'alpha-char-p s) #'char<))

> (roman-alpha "d:/test.txt")
```

```
"aaaaaaaaaabbccddrrrr"
```

Решение 2.56.2

```
(defun roman-alpha (path &aux u)
  (with-open-file (s path :direction :input)
    (do ((line (read-line s nil :eof) (read-line s nil :eof)))
        ((eql line :eof))
        (setf u (concatenate 'string u line)))
    (roman-alpha* u)))

(defun roman-alpha* (s)
  (loop for a across
        (sort (remove-if-not #'alpha-char-p s) #'char<)
        do (print a)))
```

```
> (roman-alpha "d:/test.txt")
```

```
#\a
#\a
#\a
#\a
#\a
#\a
#\a
#\a
#\a
#\a
#\b
#\b
#\b
#\b
#\c
#\c
#\d
#\d
#\r
#\r
#\r
#\r
NIL
```

Решение 2.56.3

```
(defun roman-alpha (path &aux u)
  (with-open-file (s path :direction :input)
    (do ((line (read-line s nil :eof) (read-line s nil :eof)))
        ((eql line :eof))
        (setf u (concatenate 'string u line)))
    (roman-alpha* u)))

(defun roman-alpha* (s)
  (loop for a across
        (sort (remove-if-not #'alpha-char-p s) #'char<))
```

```

collect a))

> (roman-alpha "d:/test.txt")
(#\a #\a #\a #\a #\a #\a #\a #\a #\a #\a #\b #\b #\b #\b #\c #\c #\d
#\d #\r #\r #\r #\r)

```

Задача 2.57 sum-number-digit.lisp

Определить функцию, которая считает сумму и число цифр целого n.

Решение 2.57.1

```

(defun sum-number-digit (n)
  (if (zerop n)
      '(0 0)
      (list (+ (rem n 10)
                (car (sum-number-digit (floor n 10)))))
            (+ 1
              (cadr (sum-number-digit (floor n 10)))))))

> (sum-number-digit 12345)
(15 5)

```

Решение 2.57.2

```

(defun sum-number-digit (n)
  (if (zerop n)
      '(0 0)
      `(+ (rem n 10) (car (sum-number-digit (floor n 10)))
          , (1+ (cadr (sum-number-digit (floor n 10)))))))

> (sum-number-digit 12345)
(15 5)

```

Задача 2.58 consonant.lisp

Дана строка из букв латинского алфавита, цифр и знаков препинания. Определить функцию, возвращающую список согласных.

Решение 2.58.1

```

(defparameter w '(#\b #\c #\d #\f #\g #\h #\j #\k #\l #\m #\n #\p
#\q #\r #\s #\t #\v #\x #\z #\w #\y #\B #\C #\D #\F #\G #\H #\J
#\K #\L #\M #\N #\P #\Q #\R #\S #\T #\V #\X #\Z #\W #\Y))

(defun consonant (s)
  (loop for c across s when (member c w) collect c))

> (consonant "ABRA, CA dabra477.")
(#\B #\R #\C #\d #\b #\r)

```

Задача 2.59 armstrong-decimal-number.lisp

I don't think using Emacs could make a better programmer, nor do I think that having a couple of relatively famous middle or old aged people using (or previously using) Emacs could prove that Emacs has some unique power. However, when I started, there were a few Emacs users that I have high respect for, which is one of the reasons that made me continue to play with Emacs. Famous Emacs Users (that are not famous for using Emacs): Marijn Haverbeke - the author of Eloquent JavaScript and CodeMirror; Jeffrey Friedl - the author of Mastering Regular Expressions; Michael Widenius - the author of MySQL and MariaDB; Joe Armstrong - the creator of Erlang; Amelia Andersdotter - Politician; Steven Brust - the author of Vlad Taltos; Vernon Vinge - the author of True Names and A Fire Upon the Deep, Hugo Award winner; Whitfield Diffie - security expert (Diffie-Hellman key exchange); Marc Andreessen - the creator of the Mosaic browser and the founder of Netscape; Richard Gabriel - the author of Worse is Better; Guy Steele - an expert of programming languages, the author of C: A Reference Manual; Peter Norvig - the author of Artificial Intelligence: A Modern Approach; Jamie Zawinski - an important programmer of the Mozilla project team; Daniel Weinreb - cofounder of Symbolics; Julian Assange - the founder of WikiLeaks; Martin Fowler - coauthor of Refactoring: Improving the Design of Existing Code; Mark Zuckerberg - Facebook founder; Yukihiro Matsumoto - the creator of Ruby; Guido van Rossum - the creator of Python; Linus Torvalds - the creator of Linux and Git; Donald Knuth - the creator of TeX and the author of TAOCP, Turing Award winner; Eric Raymond - the author of The Cathedral and the Bazaar and How to become a hacker; Richard Stallman - the creator of GNU Emacs, the founder of GNU.

Ren Wenshan

Определить функцию, возвращающую десятичные числа Армстронга (натуральное число, которое в данной системе счисления равно сумме своих цифр, возведённых в степень, равную количеству его цифр - самовлюблённое число, или совершенный цифровой инвариант - Pluperfect Digital Invariant, PPDI), не превышающие заданного n .

Решение 2.59.1

```
(defun int-digit (n)
  (mapcar #'parse-integer (map 'list #'string (format nil "~D" n))))

(defun narcissisticp (n &aux (w (int-digit n)) (m (length w)))
  (= n (reduce #'+ (mapcar #'(lambda (a) (expt a m))
                           w)))))

(defun armstrong-decimal-number (n)
  (loop for a from 0 to n
        when (narcissisticp a) collect a))

> (time (armstrong-decimal-number 100000))
Real time: 3.2071834 sec.
Run time: 3.2136207 sec.
Space: 81556620 Bytes
GC: 156, GC time: 0.9048058 sec.
(0 1 2 3 4 5 6 7 8 9 153 370 371 407 1634 8208 9474 54748 92727 93084)
```

Решение 2.59.2

```
(defun int-digit (n)
  (mapcar #'parse-integer (map 'list #'string (write-to-string n))))

(defun narcissisticp (n &aux (w (int-digit n)) (m (length w)))
  (= n (reduce #'+ (mapcar #'(lambda (a) (expt a m))
                           w)))))

(defun armstrong-decimal-number (n)
  (loop for a from 0 to n
        when (narcissisticp a) collect a))

> (time (armstrong-decimal-number 100000))
Real time: 2.9761703 sec.
Run time: 2.964019 sec.
Space: 81556620 Bytes
GC: 156, GC time: 0.8892057 sec.
(0 1 2 3 4 5 6 7 8 9 153 370 371 407 1634 8208 9474 54748 92727 93084)
```

Решение 2.59.3

```
(defun int-digit (n)
  (mapcar #'digit-char-p (coerce (write-to-string n) 'list)))

(defun narcissisticp (n &aux (w (int-digit n)) (m (length w)))
  (= n (reduce #'+ (mapcar #'(lambda (a) (expt a m))
                           w)))))

(defun armstrong-decimal-number (n)
  (loop for a from 0 to n
        when (narcissisticp a) collect a))
```

```
> (time (armstrong-decimal-number 100000))
Real time: 2.6831534 sec.
Run time: 2.6832173 sec.
Space: 75689868 Bytes
GC: 145, GC time: 0.780005 sec.
(0 1 2 3 4 5 6 7 8 9 153 370 371 407 1634 8208 9474 54748 92727 93084)
```

Решение 2.59.4

```
(defun int-digit (n)
  (mapcar #'digit-char-p (coerce (write-to-string n) 'list)))

(defun narcissisticp (n &aux (w (int-digit n)) (m (length w)))
  (= n (reduce #'+ (mapcar (lambda (a) (expt a m)) w))))

(defun armstrong-decimal-number (n)
  (loop for a from 0 to n
        when (narcissisticp a) collect a))

> (time (armstrong-decimal-number 100000))
Real time: 2.6981542 sec.
Run time: 2.6988173 sec.
Space: 75689868 Bytes
GC: 145, GC time: 0.8580055 sec.
(0 1 2 3 4 5 6 7 8 9 153 370 371 407 1634 8208 9474 54748 92727 93084)
```

Решение 2.59.5

```
(defun int-digit (n)
  (mapcar #'digit-char-p (map 'list #'identity (write-to-string n))))

(defun narcissisticp (n &aux (w (int-digit n)) (m (length w)))
  (= n (reduce #'+ (mapcar #'(lambda (a) (expt a m))
                             w)))))

(defun armstrong-decimal-number (n)
  (loop for a from 0 to n
        when (narcissisticp a) collect a))

> (time (armstrong-decimal-number 100000))
Real time: 2.6961544 sec.
Run time: 2.6832173 sec.
Space: 75689868 Bytes
GC: 145, GC time: 0.8580055 sec.
(0 1 2 3 4 5 6 7 8 9 153 370 371 407 1634 8208 9474 54748 92727 93084)
```

Решение 2.59.6

```
(defun int-digit (n)
  (mapcar #'digit-char-p (concatenate 'list (write-to-string n))))

(defun narcissisticp (n &aux (w (int-digit n)) (m (length w)))
```

```

(= n (reduce #' + (mapcar #' (lambda (a) (expt a m))
                             w))))

(defun armstrong-decimal-number (n)
  (loop for a from 0 to n
        when (narcissisticp a) collect a))

> (time (armstrong-decimal-number 100000))
Real time: 2.5551462 sec.
Run time: 2.496016 sec.
Space: 75689868 Bytes
GC: 145, GC time: 0.8112052 sec.
(0 1 2 3 4 5 6 7 8 9 153 370 371 407 1634 8208 9474 54748 92727 93084)

```

Решение 2.59.7

```

(defun int-digit (n)
  (loop for c across (write-to-string n)
        collect (digit-char-p c)))

(defun narcissisticp (n &aux (w (int-digit n)) (m (length w)))
  (= n (reduce #' + (mapcar #' (lambda (a) (expt a m))
                             w))))

(defun armstrong-decimal-number (n)
  (loop for a from 0 to n
        when (narcissisticp a) collect a))

> (time (armstrong-decimal-number 100000))
Real time: 2.3461342 sec.
Run time: 2.2932148 sec.
Space: 71778700 Bytes
GC: 137, GC time: 0.7176046 sec.
(0 1 2 3 4 5 6 7 8 9 153 370 371 407 1634 8208 9474 54748 92727 93084)

```

Решение 2.59.8

```

(defun int-digit (n)
  (map 'list #' digit-char-p (prin1-to-string n)))

(defun narcissisticp (n &aux (w (int-digit n)) (m (length w)))
  (= n (reduce #' + (mapcar #' (lambda (a) (expt a m))
                             w))))

(defun armstrong-decimal-number (n)
  (loop for a from 0 to n
        when (narcissisticp a) collect a))

> (time (armstrong-decimal-number 100000))
Real time: 2.499143 sec.
Run time: 2.4492157 sec.
Space: 71778700 Bytes
GC: 137, GC time: 0.8424054 sec.

```



```
(0 1 2 3 4 5 6 7 8 9 153 370 371 407 1634 8208 9474 54748 92727 93084)
```

Решение 2.59.9

```
(defun int-digit (n &optional ac)
  (if (zerop n) ac
      (int-digit (truncate n 10)
                  (cons (rem n 10) ac))))

(defun narcissisticp (n &aux (w (int-digit n)) (m (length w)))
  (= n (reduce #'+ (mapcar #'(lambda (a) (expt a m))
                             w))))

(defun armstrong-decimal-number (n)
  (loop for a from 0 to n
        when (narcissisticp a) collect a))

> (time (armstrong-decimal-number 100000))
Real time: 1.0380594 sec.
Run time: 1.0296066 sec.
Space: 7822480 Bytes
GC: 15, GC time: 0.0780005 sec.
(0 1 2 3 4 5 6 7 8 9 153 370 371 407 1634 8208 9474 54748 92727 93084)
```

Решение 2.59.10

```
(defparameter *v* (load-expt 10 10))

(defun load-expt (n m &aux (v (make-array (list n m))))
  (dotimes (i n)
    (dotimes (j m)
      (setf (aref v i j) (expt i j))))
  v)

(defun int-digit (n)
  (loop with r do (setf (values n r) (truncate n 10))
        collect r into ac until (zerop n)
        finally (return (nreverse ac))))

(defun narcissisticp (n &aux (w (int-digit n)) (m (length w)))
  (= n (reduce #'+ (mapcar #'(lambda (a) (aref *v* a m))
                             w))))

(defun armstrong-decimal-number (n)
  (loop with z for a upto n
        when (narcissisticp a) collect a))

> (time (armstrong-decimal-number 100000))
Real time: 0.8200493 sec.
Run time: 0.8125 sec.
Space: 7822496 Bytes
GC: 15, GC time: 0.078125 sec.
(0 1 2 3 4 5 6 7 8 9 153 370 371 407 1634 8208 9474 54748 92727 93084)
```

Решение 2.59.11

```
(defparameter *v* (load-expt 10 10))

(defun load-expt (n m &aux (v (make-array (list n m))))
  (dotimes (i n)
    (dotimes (j m)
      (setf (aref v i j) (expt i j))))
  v)

(defun int-digit (n &optional ac)
  (if (zerop n) ac
      (int-digit (truncate n 10)
                  (cons (rem n 10) ac))))

(defun narcissisticp (n &aux (w (int-digit n)) (m (length w)))
  (= n (reduce #'(lambda (a) (aref *v* a m))
               w))))

(defun armstrong-decimal-number (n)
  (loop with z for a upto n
        when (narcissisticp a) collect a))

> (time (armstrong-decimal-number 100000))
Real time: 0.8490485 sec.
Run time: 0.8424054 sec.
Space: 7822480 Bytes
GC: 15, GC time: 0.0780005 sec.
(0 1 2 3 4 5 6 7 8 9 153 370 371 407 1634 8208 9474 54748 92727 93084)
```

Решение 2.59.12 (автор - helter, www.cyberforum.ru)

```
(declaim (optimize (speed 3) (safety 0) (debug 0) (space 0)))

(declaim (inline pow))

(defun pow (x k)
  (declare (type fixnum x k))
  (let ((p 1))
    (declare (type fixnum p))
    (loop
      repeat k
      do (setf p (the fixnum (* p x)))
      finally (return p))))

(declaim (inline armstrongp))

(defun armstrongp (n)
  (declare (type fixnum n))
  (let ((p 1))
    (declare (type fixnum p))
    (let ((n n))
```

```

    (declare (type fixnum n))
    (loop
      (when (< n 10) (return))
      (setf n (the fixnum (floor (the unsigned-byte n) 10)))
      (incf p)))
  (let ((b n)
        (r 0)
        (s 0))
    (declare (type fixnum p b r s))
    (loop
      (multiple-value-setq (b r) (floor (the unsigned-byte b) 10))
      (incf s (the fixnum (pow r p)))
      (when (zerop b) (return (= s n))))))

(defun armstrong-decimal-number (n)
  (declare (type fixnum n))
  (let ((res '()))
    (dotimes (i n (nreverse res))
      (declare (fixnum i))
      (when (armstrongp i)
        (push i res)))))

> (time (armstrong-decimal-number 100000))
Real time: 1.363078 sec.
Run time: 1.248008 sec.
Space: 160 Bytes
(0 1 2 3 4 5 6 7 8 9 153 370 371 407 1634 8208 9474 54748 92727 93084)

```

Задача 2.60 longest-roman-prime.lisp

Определить функцию, возвращающую список из не превышающих заданного *n* самых длинных чисел в римской записи вместе с этим же числом в арабской записи и длиной этого числа в римской записи.

Решение 2.60.1

```

(defun longest-roman-prime (m
                            &aux
                            (v (loop for n from 1 to m
                                      when (primep n)
                                      collect n))
                            (z (loop for n in v
                                      maximize
                                      (length
                                       (format nil "~@r" n))))))
  (loop for n in v
        when (= (length (format nil "~@r" n)) z)
        collect (list (format nil "~@r" n) n z)))

(defun primep (n)
  (loop for i from 2 to (isqrt n)
        never (zerop (rem n i))))

```

```
> (longest-roman-prime 3999)
(("MMDCCCLXXXVII" 2887 13) ("MMMDCCLXXXIII" 3833 13) ("MMMDCCLXXXVII"
3877 13) ("MMMDCCLXXXIX" 3889 13))
```

Решение 2.60.2

```
(defun longest-roman-prime (m
  &aux
  (v (loop for n from 1 to m
    when (primep n)
    collect n))
  (z (loop for n in v
    maximize
    (length
    (format nil "~:@r" n)))))
  (loop for n in v
    when (= (length (format nil "~:@r" n)) z)
    collect (list (format nil "~:@r" n) n z)))

(defun primep (n)
  (loop for i from 2 to (isqrt n)
    never (zerop (rem n i))))

> (longest-roman-prime 3999)
(("MMDCCCLXXXVIII" 2999 17) ("MMMCCCLXXXVIII" 3499 17)
 ("MMMDCCLXXXVIII" 3989 17))
> (longest-roman-prime 4999)
(("MMMDCCLXXXVIII" 4999 19))
```

Решение 2.60.3

```
(defun longest-roman-prime (m
  &aux
  (v (loop for n from 1 to m
    when (primep n)
    collect
    (list (format nil "~:@r" n) n)))
  (z (reduce #'max
    (mapcar
    #'(lambda (a) (length (car a)))
    v))))
  (loop for a in v
    for b in (mapcar #'cadr v)
    when (= (length (car a)) z)
    collect (list a b z)))

(defun primep (n)
  (loop for i from 2 to (isqrt n)
    never (zerop (rem n i))))

> (longest-roman-prime 3999)
(("MMDCCCLXXXVII" 2887 13) ("MMMDCCLXXXIII" 3833 13) ("MMMDCCLXXXVII"
```

```
3877 13) ("MMMDCCLXXXIX" 3889 13))
```

Решение 2.60.4

```
(defun longest-roman-prime (m
  &aux
    (v (loop for n from 1 to m
      when (primep n)
      collect
        (list (format nil "~:@r" n) n)))
    (z (reduce #'max
      (mapcar
        #'(lambda (a) (length (car a)))
        v))))
  (loop for a in v
    for b in (mapcar #'cadr v)
    when (= (length (car a)) z)
    collect (list a b z)))

(defun primep (n)
  (loop for i from 2 to (isqrt n)
    never (zerop (rem n i))))

> (longest-roman-prime 3999)
(("MMDCCCCLXXXVIII" 2999 17) ("MMMCCCCLXXXVIII" 3499 17)
 ("MMDCCCCLXXXVIII" 3989 17))
> (longest-roman-prime 4999)
(("MMMMDCCCCLXXXVIII" 4999 19))
```

Задача 2.61 split.lisp

Определить функцию, возвращающую список из не превышающих заданного *n* самых длинных чисел в римской записи вместе с этим же числом в арабской записи и длиной этого числа в римской записи.

Решение 2.61.1 (автор – Zach Beane, <http://lisptips.com>)

```
(defun split (string &optional (split-character #\Space))
  (let ((result '()))
    (stream (make-string-output-stream)))
    (loop for char across string
      if (char= char split-character)
      do (push (get-output-stream-string stream) result)
      else
      do (write-char char stream))
    (push (get-output-stream-string stream) result)
    (nreverse result)))

> (split "1 2 3 466 55")
("1" "2" "3" "466" "55")

Решение 2.61.2
```

```
(defun split (s &optional (c #\Space)
              &aux ac (m (make-string-output-stream)))
  (loop for a across s
        if (char= a c) do
          (push (get-output-stream-string m) ac)
        else do (write-char a m))
  (push (get-output-stream-string m) ac)
  (nreverse ac))

> (split "1 2 3 466 55")
("1" "2" "3" "466" "55")
```

Задача 2.62 *max-length-number.lisp*

Определить функцию, которая принимает на вход текст и возвращает два значения: длину самого длинного слова, состоящего только из цифр, и само это слово. Если таких слов несколько, функция возвращает последнее из них.

Решение 2.62.1

```
(defun split (s &aux ac (m (make-string-output-stream)))
  (loop for a across s
        if (or (char= a #\Space)
                (char= a #\Tab)
                (char= a #\Newline)) do
          (push (get-output-stream-string m) ac)
        else do (write-char a m))
  (push (get-output-stream-string m) ac)
  (nreverse ac))

(defun max-length (w &aux (m (loop for a in w
                                   when (all-digit-p a)
                                   maximize (length a))))
  (when (and m (plusp m))
    (list m (find-if
              #'(lambda (a)
                  (and (all-digit-p a)
                      (= (length a)
                         m)))
              w :from-end t)))))

(defun all-digit-p (s)
  (loop for c across s always (digit-char-p c)))

(defun max-length-number (s) (max-length (split s)))

> (max-length-number "aa 333 bb 555 77")
(3 "555")
> (max-length-number "77")
(2 "77")
```

```
> (max-length-number "aa")
NIL
> (max-length-number "")
NIL
```

Задача 2.63 *perfect.lisp*

If nothing else, it teaches you an entirely different way of programming. It stretches your brain.

Michael Kosak

Определить функцию, возвращающую все натуральные числа, которые меньше либо равны n и делятся без остатка на цифры числа n .

Решение 2.63.1

```
(defun int-digits (n &optional ac)
  (if (zerop n) ac
      (int-digits (truncate n 10)
                   (cons (rem n 10) ac))))

(defun perfect (n)
  (loop for a from 1 to n
        when (loop for b in (remove 0 (int-digits n))
                    always (zerop (rem a b)))
        collect a))

> (perfect 23479)
(252 504 756 1008 1260 1512 1764 2016 2268 2520 2772 3024 3276 3528
 3780 4032 4284 4536 4788 5040 5292 5544 5796 6048 6300 6552 6804 7056
 7308 7560 7812 8064 8316 8568 8820 9072 9324 9576 9828 10080 10332
 10584 10836 11088 11340 11592 11844 12096 12348 12600 12852 13104
 13356 13608 13860 14112 14364 14616 14868 15120 15372 15624 15876
 16128 16380 16632 16884 17136 17388 17640 17892 18144 18396 18648
 18900 19152 19404 19656 19908 20160 20412 20664 20916 21168 21420
 21672 21924 22176 22428 22680 22932 23184 23436)
```

Решение 2.63.2

```
(defun int-digits (n &optional ac)
  (if (zerop n) ac
      (int-digits (truncate n 10)
                   (cons (rem n 10) ac))))

(defun perfect (n)
  (loop for a from 1 to n
        when (loop for b in (remove 0 (int-digits n))
                    never (plusp (rem a b)))
        collect a))
```

```
> (perfect 23479)
(252 504 756 1008 1260 1512 1764 2016 2268 2520 2772 3024 3276 3528
3780 4032 4284 4536 4788 5040 5292 5544 5796 6048 6300 6552 6804 7056
7308 7560 7812 8064 8316 8568 8820 9072 9324 9576 9828 10080 10332
10584 10836 11088 11340 11592 11844 12096 12348 12600 12852 13104
13356 13608 13860 14112 14364 14616 14868 15120 15372 15624 15876
16128 16380 16632 16884 17136 17388 17640 17892 18144 18396 18648
18900 19152 19404 19656 19908 20160 20412 20664 20916 21168 21420
21672 21924 22176 22428 22680 22932 23184 23436)
```

Решение 2.63.3

```
(defun int-digits-zero- (n &optional ac &aux (r (rem n 10)))
  (if (zerop n) ac
      (int-digits (truncate n 10)
                   (if (zerop r) ac (cons r ac)))))

(defun perfect (n)
  (loop for a from 1 to n
        when (loop for b in (int-digits-zero- n)
                     always (zerop (rem a b)))
        collect a))
```

```
> (perfect 23479)
(252 504 756 1008 1260 1512 1764 2016 2268 2520 2772 3024 3276 3528
3780 4032 4284 4536 4788 5040 5292 5544 5796 6048 6300 6552 6804 7056
7308 7560 7812 8064 8316 8568 8820 9072 9324 9576 9828 10080 10332
10584 10836 11088 11340 11592 11844 12096 12348 12600 12852 13104
13356 13608 13860 14112 14364 14616 14868 15120 15372 15624 15876
16128 16380 16632 16884 17136 17388 17640 17892 18144 18396 18648
18900 19152 19404 19656 19908 20160 20412 20664 20916 21168 21420
21672 21924 22176 22428 22680 22932 23184 23436)
```

Задача 2.64 invert-deci-bin.lisp

Прочитать беззнаковое целое число и в каждом байте числа инвертировать значения двух старших битов каждой тетрады, если младший бит тетрады единичный. Примеры преобразования:

```
65535 (11111111 11111111) => 13107 (00110011 00110011)
333444 (00000101 00010110 10000100) => 644740 (00001001 11010110
10000100)
```

Решение 2.64.1

```
(defun invert-deci-bin (n) (invert (deci-bin n)))

(defun deci-bin (n &optional ac)
  (if (zerop n) ac (deci-bin (floor n 2) (cons (mod n 2) ac))))

(defun invert (w &aux (f (caddr w)))
  (cond ((null f) nil)
        ((= f 1) (cons (inv (car w))
```



```

                                (equal x #\-)))
s) " )"))))

> (abundant-word "aaa aaa bbb bbb ccc")
(AAA BBB)
> (abundant-word "aaa aaa bbb")
(AAA)
> (abundant-word "")
NIL

```

Задача 2.66 *s-t-r-i-n-g.lisp*

Дана строка. Определить функцию, возвращающую список букв исходной строки.

Решение 2.66.1

```

(defun s-t-r-i-n-g (s) (loop for c across s collect c))

> (s-t-r-i-n-g "brett")
(#\b #\r #\e #\t #\t)

```

Решение 2.66.2

```

(defun s-t-r-i-n-g (s) (map 'list #'identity s))

> (s-t-r-i-n-g "brett")
(#\b #\r #\e #\t #\t)

```

Решение 2.66.3

```

(defun s-t-r-i-n-g (s) (concatenate 'list s))

> (s-t-r-i-n-g "brett")
(#\b #\r #\e #\t #\t)

```

Решение 2.66.4

```

(defun s-t-r-i-n-g (s) (coerce s 'list))

> (s-t-r-i-n-g "brett")
(#\b #\r #\e #\t #\t)

```

Задача 2.67 *sierpinski-carpet.lisp*

Определить функцию, которая выводит ковёр Серпинского (квадрат Серпинского) — фрактал, один из двумерных аналогов множества Кантора, предложенный польским математиком Вацлавом Серпинским.

Решение 2.67.1 (автор ~, <https://rosettacode.org>)

```
(defun sierpinski-carpet (order)
  (let ((size (expt 3 order)))
    (flet ((trinary (x) (format nil "~3,vR" order x))
      (ones (a b) (and (eql a #\1) (eql b #\1))))
      (loop for i below size do
        (fresh-line)
        (loop for j below size do
          (princ (if (some #'ones (trinary i) (trinary j))
                     " "
                     "#"))))))))
```

```
> (sierpinski-carpet 3)
#####
# # # # # # # # # # #
#####
###   #####   #####   ###
# #   # # #   # # #   # #
###   #####   #####   ###
#####
# # # # # # # # # # #
#####
#####           #####
# # # # #       # # # # #
#####           #####
###   ###       ###   ###
# #   # #       # #   # #
###   ###       ###   ###
#####           #####
# # # # #       # # # # #
#####           #####
#####
# # # # # # # # # # #
#####
###   #####   #####   ###
# #   # # #   # # #   # #
###   #####   #####   ###
#####
# # # # # # # # # # #
#####
```

NIL

Задача 2.68 sd.lisp

Определить функцию, которая по заданному вещественному числу формирует список двух элементов. Первый элемент списка — знак числа, второй — принадлежность числа диапазону $(-5\ 5)$.

Решение 2.68.1

```
(defun sd (n)
  (list (cond ((plusp n) 1)
        ((zerop n) 0)
```

```
      (-1))
    (<= -5 n 5)))
```

```
> (sd -3)
(-1 T)
> (sd 3)
(1 T)
```

Решение 2.68.2

```
(defun sd (n) `(,(if (zerop n) n (/ n (abs n))) ,(<= -5 n 5)))

> (sd -3)
(-1 T)
> (sd 3)
(1 T)
```

Решение 2.68.3

```
(defun sd (n) `(,(signum n) ,(<= -5 n 5)))

> (sd -3)
(-1 T)
> (sd 3)
(1 T)
```

Задача 2.69 *queens.lisp*

Sometimes, the elegant implementation is just a function. Not a method. Not a class. Not a framework. Just a function.

John Carmack

Programming isn't about what you know; it's about what you can figure out.

Chris Pine

Only lispers have a true definition of fun.

Mike Pechkin

Определить функцию, расставляющую на шахматной доске 8 ферзей в соответствии со стандартными правилами, так, чтобы никакие два ферзя не били друг друга (ферзь бьёт любую фигуру, находящуюся с ним на одной вертикали, горизонтали и диагонали).

Решение 2.69.1

```
(defun queens (n &optional (m n))
  (if (zerop n) `(,nil)
```

```

(loop for s in (queens (1- n) m)
  nconc (loop for nc from 1 to m
    when (loop for r from 1 to n
      for c in s
        always (/= nc c (+ c r) (- c r)))
      collect (cons nc s))))

> (queens 8)
((4 2 7 3 6 8 5 1) (5 2 4 7 3 8 6 1) (3 5 2 8 6 4 7 1) (3 6 4 2 8 5 7
1) (5 7 1 3 8 6 4 2) (4 6 8 3 1 7 5 2) (3 6 8 1 4 7 5 2) (5 3 8 4 7 1
6 2) (5 7 4 1 3 8 6 2) (4 1 5 8 6 3 7 2) (3 6 4 1 8 5 7 2) (4 7 5 3 1
6 8 2) (6 4 2 8 5 7 1 3) (6 4 7 1 8 2 5 3) (1 7 4 6 8 2 5 3) (6 8 2 4
1 7 5 3) (6 2 7 1 4 8 5 3) (4 7 1 8 5 2 6 3) (5 8 4 1 7 2 6 3) (4 8 1
5 7 2 6 3) (2 7 5 8 1 4 6 3) (1 7 5 8 2 4 6 3) (2 5 7 4 1 8 6 3) (4 2
7 5 1 8 6 3) (5 7 1 4 2 8 6 3) (6 4 1 5 8 2 7 3) (5 1 4 6 8 2 7 3) (5
2 6 1 7 4 8 3) (6 3 7 2 8 5 1 4) (2 7 3 6 8 5 1 4) (7 3 1 6 8 5 2 4)
(5 1 8 6 3 7 2 4) (1 5 8 6 3 7 2 4) (3 6 8 1 5 7 2 4) (6 3 1 7 5 8 2
4) (7 5 3 1 6 8 2 4) (7 3 8 2 5 1 6 4) (5 3 1 7 2 8 6 4) (2 5 7 1 3 8
6 4) (3 6 2 5 8 1 7 4) (6 1 5 2 8 3 7 4) (8 3 1 6 2 5 7 4) (2 8 6 1 3
5 7 4) (5 7 2 6 3 1 8 4) (3 6 2 7 5 1 8 4) (6 2 7 1 3 5 8 4) (3 7 2 8
6 4 1 5) (6 3 7 2 4 8 1 5) (4 2 7 3 6 8 1 5) (7 1 3 8 6 4 2 5) (1 6 8
3 7 4 2 5) (3 8 4 7 1 6 2 5) (6 3 7 4 1 8 2 5) (7 4 2 8 6 1 3 5) (4 6
8 2 7 1 3 5) (2 6 1 7 4 8 3 5) (2 4 6 8 3 1 7 5) (3 6 8 2 4 1 7 5) (6
3 1 8 4 2 7 5) (8 4 1 3 6 2 7 5) (4 8 1 3 6 2 7 5) (2 6 8 3 1 4 7 5)
(7 2 6 3 1 4 8 5) (3 6 2 7 1 4 8 5) (4 7 3 8 2 5 1 6) (4 8 5 3 1 7 2
6) (3 5 8 4 1 7 2 6) (4 2 8 5 7 1 3 6) (5 7 2 4 8 1 3 6) (7 4 2 5 8 1
3 6) (8 2 4 1 7 5 3 6) (7 2 4 1 8 5 3 6) (5 1 8 4 2 7 3 6) (4 1 5 8 2
7 3 6) (5 2 8 1 4 7 3 6) (3 7 2 8 5 1 4 6) (3 1 7 5 8 2 4 6) (8 2 5 3
1 7 4 6) (3 5 2 8 1 7 4 6) (3 5 7 1 4 2 8 6) (5 2 4 6 8 3 1 7) (6 3 5
8 1 4 2 7) (5 8 4 1 3 6 2 7) (4 2 5 8 6 1 3 7) (4 6 1 5 2 8 3 7) (6 3
1 8 5 2 4 7) (5 3 1 6 8 2 4 7) (4 2 8 6 1 3 5 7) (6 3 5 7 1 4 2 8) (6
4 7 1 3 5 2 8) (4 7 5 2 6 1 3 8) (5 7 2 6 3 1 4 8))

```

Задача 2.70 *cleave.lisp*

Дана строка. Определить функцию, которая формирует список подстрок по разделителю двоеточие.

Решение 2.70.1

```

(defun cleave (s)
  (let ((p (position #\: s)))
    (if p (cons (subseq s 0 p) (cleave (subseq s (1+ p)))) `(),s)))

> (cleave "aa:bb:cc")
("aa" "bb" "cc")

```

Решение 2.70.2

```

(defun cleave (s &optional ac &aux (p (position #\: s)))
  (if p (cleave (subseq s (1+ p)) (cons (subseq s 0 p) ac))
    (reverse (cons s ac))))

```

```
> (cleave "aa:bb:cc")
("aa" "bb" "cc")
```

Задача 2.71 *snail.lisp*

While many people seems to be annoyed at the parenthesis in Lisp, Lisp none-the-less remains the language with the simplest and most concise syntax of languages I now. There are fewer rules than in any other language. One consequence of this, I find, is that Lisp is less in the way of the programmers cognitive efforts to write the program. I am convinced that Lisp programmers are more productive, in part because Lisp is linguistically more plastic and powerfull than others and this is important because I believe there to be deep links between language and cognition. In other words, you cannot program what you cannot articulate and since Lisp supports articulation better than most, Lisp programs becomes shorter in a way that supports the writing process. Programs can become so terse that they become incomprehensible, using one letter variable names does not improve legibility, but apart from that, shorter programs are in general better. They take less to write, less time to debug, less time to read and the programmer can keep relatively more of the program in his head at one point. The last point is important. At university we were told about a studies from the 70s (which I sadly have no reference to) that showed that giving programmers larger screens, improved their productivity. This effect is also the reason why higher level languages are better than lower ones (e.g. FORTRAN vs assembler) since highlevel languages are able to present more knowledge at a given screen.

Christian Lynbech

Дано целое число от 1 до 9. Определить функцию «улитка», которая выводит на экран список чисел по следующему образцу:

```
n = 3
3333333
3222223
3211123
```

```
3210123
3211123
3222223
3333333
```

```
n = 1
111
101
111
```

Решение 2.71.1

```
(defun 1357-row (n)
  (loop for a from 1 to n by 2 collect a))

(defun 0123-row (n)
  (loop for a from 0 to n collect a))

(defun 0111-row (n)
  (mapcar #'(lambda (a b) (loop repeat b collect a))
    (0123-row n) (1357-row n)))

(defun 1234-row (n)
  (loop for a from 1 to n collect a))

(defun 4321-row (n)
  (loop for a downfrom n to 1 collect a))

(defun trimm (w &aux (z (length w)))
  (loop for a from 0 to (1- z) collect (butlast w a)))

(defun ttrim (w &aux (z (length w)))
  (loop for a from 0 to (1- z) collect (nthcdr a w)))

(defun sn (n)
  (mapcar #'append
    (trimm (4321-row n))
    (0111-row (+ n n))
    (ttrim (1234-row n))))

(defun sna (n
  &aux (w (sn n))
  (v (loop repeat (+ n n 1) collect n)))
  (mapcar #'print (append (cons v (reverse (cdr w)))
    w
    ` (,v)))
  nil)

(defun snail (n)
  (when (< 0 n 10) (sna n)))

> (snail 0)
NIL
```

```

> (snail 1)

(1 1 1)
(1 0 1)
(1 1 1)
NIL
> (snail 9)

(9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9)
(9 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8)
(9 8 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 8)
(9 8 7 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 7 8)
(9 8 7 6 5 5 5 5 5 5 5 5 5 5 5 5 5 6 7 8)
(9 8 7 6 5 4 4 4 4 4 4 4 4 4 4 4 5 6 7 8)
(9 8 7 6 5 4 3 3 3 3 3 3 3 3 3 3 4 5 6 7 8)
(9 8 7 6 5 4 3 2 2 2 2 2 2 3 4 5 6 7 8 9)
(9 8 7 6 5 4 3 2 1 1 1 2 3 4 5 6 7 8 9)
(9 8 7 6 5 4 3 2 1 0 1 2 3 4 5 6 7 8 9)
(9 8 7 6 5 4 3 2 1 1 1 2 3 4 5 6 7 8 9)
(9 8 7 6 5 4 3 2 2 2 2 2 2 3 4 5 6 7 8 9)
(9 8 7 6 5 4 3 3 3 3 3 3 3 3 4 5 6 7 8 9)
(9 8 7 6 5 4 4 4 4 4 4 4 4 4 5 6 7 8 9)
(9 8 7 6 5 5 5 5 5 5 5 5 5 5 5 6 7 8 9)
(9 8 7 6 6 6 6 6 6 6 6 6 6 6 6 6 7 8 9)
(9 8 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 8 9)
(9 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 9)
(9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9)
NIL
> (snail 10)
NIL

```

Решение 2.71.2

```

(defun 0111-row (n)
  (mapcar #'(lambda (a b) (loop repeat b collect a))
    (loop for a from 0 to n collect a)
    (loop for a from 1 to n by 2 collect a)))

(defun trimm (w &aux (z (length w)))
  (loop for a from 0 to (1- z) collect (butlast w a)))

(defun ttrim (w &aux (z (length w)))
  (loop for a from 0 to (1- z) collect (nthcdr a w)))

(defun sn (n)
  (mapcar #'append
    (trimm (loop for a downfrom n to 1 collect a))
    (0111-row (+ n n))
    (ttrim (loop for a from 1 to n collect a))))

(defun sna (n &aux (w (sn n)) (v (loop repeat (+ n n 1) collect n)))
  (mapcar #'print (append (cons v (reverse (cdr w))) w `(,v)) nil)

```



```
(defun snail (n)
  (when (< 0 n 10) (sna n)))
```

```
> (snail 5)
```

```
(5 5 5 5 5 5 5 5 5 5)
(5 4 4 4 4 4 4 4 4 5)
(5 4 3 3 3 3 3 3 3 4 5)
(5 4 3 2 2 2 2 2 3 4 5)
(5 4 3 2 1 1 1 2 3 4 5)
(5 4 3 2 1 0 1 2 3 4 5)
(5 4 3 2 1 1 1 2 3 4 5)
(5 4 3 2 2 2 2 2 3 4 5)
(5 4 3 3 3 3 3 3 3 4 5)
(5 4 4 4 4 4 4 4 4 4 5)
(5 5 5 5 5 5 5 5 5 5 5)
NIL
```

Решение 2.71.3

```
(defun 0111-row (n)
  (mapcar #'(lambda (a b) (loop repeat b collect a))
    (loop for a from 0 to n collect a)
    (loop for a from 1 to n by 2 collect a)))

(defun trimm (w &aux (z (length w)))
  (loop for a from 0 to (1- z) collect (butlast w a)))

(defun sn (n &aux (v (trimm (loop for a downfrom n to 1 collect a))))
  (mapcar #'append v (0111-row (+ n n)) (mapcar #'reverse v)))

(defun sna (n &aux (w (sn n)) (v (loop repeat (+ n n 1) collect n)))
  (mapcar #'print (append (cons v (reverse (cdr w))) w `(:,v))) nil)

(defun snail (n)
  (when (< 0 n 10) (sna n)))

> (snail 5)
```

```
(5 5 5 5 5 5 5 5 5 5 5)
(5 4 4 4 4 4 4 4 4 4 5)
(5 4 3 3 3 3 3 3 3 3 4 5)
(5 4 3 2 2 2 2 2 2 3 4 5)
(5 4 3 2 1 1 1 2 3 4 5)
(5 4 3 2 1 0 1 2 3 4 5)
(5 4 3 2 1 1 1 2 3 4 5)
(5 4 3 2 2 2 2 2 3 4 5)
(5 4 3 3 3 3 3 3 3 4 5)
(5 4 4 4 4 4 4 4 4 4 5)
(5 5 5 5 5 5 5 5 5 5 5)
NIL
```

Решение 2.71.4

```

(defun 0111-row (n)
  (mapcar #'(lambda (a b) (loop repeat b collect a))
    (loop for a from 0 to n collect a)
    (loop for a from 1 to n by 2 collect a)))

(defun ttrim (w &aux (z (length w)))
  (loop for a from 0 to (1- z) collect (nthcdr a w)))

(defun sn (n &aux (v (ttrim (loop for a from 1 to n collect a))))
  (append (mapcar #'append (mapcar #'reverse v) (0111-row (+ n n)) v)
    (list (loop repeat (+ n n 1) collect n))))

(defun sna (n &aux (w (sn n)))
  (mapcar #'print (append (reverse (cdr w)) w)) nil)

(defun snail (n)
  (when (< 0 n 10) (sna n)))

```

```
> (snail 5)
```

```

(5 5 5 5 5 5 5 5 5 5)
(5 4 4 4 4 4 4 4 4 4 5)
(5 4 3 3 3 3 3 3 3 4 5)
(5 4 3 2 2 2 2 2 3 4 5)
(5 4 3 2 1 1 1 2 3 4 5)
(5 4 3 2 1 0 1 2 3 4 5)
(5 4 3 2 1 1 1 2 3 4 5)
(5 4 3 2 2 2 2 2 3 4 5)
(5 4 3 3 3 3 3 3 3 4 5)
(5 4 4 4 4 4 4 4 4 4 5)
(5 5 5 5 5 5 5 5 5 5 5)

```

```
NIL
```

Решение 2.71.5

```

(defun 0111-row (n)
  (mapcar #'(lambda (a b) (loop repeat b collect a))
    (loop for a from 0 to n collect a)
    (loop for a from 1 to (+ n 2) by 2 collect a)))

(defun ttrim (w &aux (z (length w)))
  (append (loop for a from 0 to (1- z) collect (nthcdr a w))
    `(), nil)))

(defun sn (n &aux (v (ttrim (loop for a from 1 to n collect a))))
  (mapcar #'append (mapcar #'reverse v) (0111-row (+ n n)) v))

(defun sna (n &aux (w (sn n)))
  (mapcar #'print (append (reverse (cdr w)) w)) nil)

(defun snail (n)
  (when (< 0 n 10) (sna n)))

```

```
> (snail 5)

(5 5 5 5 5 5 5 5 5 5)
(5 4 4 4 4 4 4 4 4 5)
(5 4 3 3 3 3 3 3 3 4 5)
(5 4 3 2 2 2 2 2 3 4 5)
(5 4 3 2 1 1 1 2 3 4 5)
(5 4 3 2 1 0 1 2 3 4 5)
(5 4 3 2 1 1 1 2 3 4 5)
(5 4 3 2 2 2 2 2 3 4 5)
(5 4 3 3 3 3 3 3 3 4 5)
(5 4 4 4 4 4 4 4 4 4 5)
(5 5 5 5 5 5 5 5 5 5)
NIL
```

Решение 2.71.6

```
(defun 0111-row (n)
  (mapcar #'(lambda (a b) (loop repeat b collect a))
    (loop for a from 0 to n collect a)
    (loop for a from 1 to (+ n 2) by 2 collect a)))

(defun ttrim (w &aux (z (length w)))
  (append (loop for a from 0 to (1- z) collect (nthcdr a w))
    ` (,nil)))

(defun sn (n &aux (v (ttrim (loop for a from 1 to n collect a))))
  (mapcar #'append (mapcar #'reverse v) (0111-row (+ n n)) v))

(defun sna (n &aux (w (sn n)))
  (mapcar #'print (revappend (cdr w) w)) nil)

(defun snail (n)
  (when (< 0 n 10) (sna n)))

> (snail 5)
```

```
(5 5 5 5 5 5 5 5 5 5)
(5 4 4 4 4 4 4 4 4 5)
(5 4 3 3 3 3 3 3 3 4 5)
(5 4 3 2 2 2 2 2 3 4 5)
(5 4 3 2 1 1 1 2 3 4 5)
(5 4 3 2 1 0 1 2 3 4 5)
(5 4 3 2 1 1 1 2 3 4 5)
(5 4 3 2 2 2 2 2 3 4 5)
(5 4 3 3 3 3 3 3 3 4 5)
(5 4 4 4 4 4 4 4 4 4 5)
(5 5 5 5 5 5 5 5 5 5)
NIL
```

Решение 2.71.7

```

(defun 0111-row (n)
  (mapcar #'(lambda (a b) (loop repeat b collect a))
    (loop for a from 0 to n collect a)
    (loop for a from 1 to (+ n 2) by 2 collect a)))

(defun ttrim (w &aux (z (length w)))
  (append (loop for a from 0 to (1- z) collect (nthcdr a w))
    `,(nil)))

(defun sn (n &aux (v (ttrim (loop for a from 1 to n collect a))))
  (mapcar #'nconc (mapcar #'reverse v) (0111-row (+ n n)) v))

(defun sna (n &aux (w (sn n)))
  (mapcar #'print (revappend (cdr w) w)) nil)

(defun snail (n)
  (when (< 0 n 10) (sna n)))

> (snail 5)

(5 5 5 5 5 5 5 5 5 5)
(5 4 4 4 4 4 4 4 4 5)
(5 4 3 3 3 3 3 3 3 4 5)
(5 4 3 2 2 2 2 2 3 4 5)
(5 4 3 2 1 1 1 2 3 4 5)
(5 4 3 2 1 0 1 2 3 4 5)
(5 4 3 2 1 1 1 2 3 4 5)
(5 4 3 2 2 2 2 2 3 4 5)
(5 4 3 3 3 3 3 3 3 4 5)
(5 4 4 4 4 4 4 4 4 4 5)
(5 5 5 5 5 5 5 5 5 5)
NIL

```

Решение 2.71.8

```

(defun 0111-row (n)
  (mapcar #'(lambda (a b) (loop repeat b collect a))
    (loop for a from 0 to n collect a)
    (loop for a from 1 to (+ n 2) by 2 collect a)))

(defun ttrim (w)
  (append (loop for a from 0 below (length w)
    collect (nthcdr a w))
    `,(nil)))

(defun sn (n &aux (v (ttrim (loop for a from 1 to n collect a))))
  (mapcar #'nconc (mapcar #'reverse v) (0111-row (+ n n)) v))

(defun sna (n &aux (w (sn n)))
  (mapcar #'print (revappend (cdr w) w)) nil)

(defun snail (n)
  (when (< 0 n 10) (sna n)))

```

```
> (snail 5)

(5 5 5 5 5 5 5 5 5 5)
(5 4 4 4 4 4 4 4 4 5)
(5 4 3 3 3 3 3 3 3 4 5)
(5 4 3 2 2 2 2 2 3 4 5)
(5 4 3 2 1 1 1 2 3 4 5)
(5 4 3 2 1 0 1 2 3 4 5)
(5 4 3 2 1 1 1 2 3 4 5)
(5 4 3 2 2 2 2 2 3 4 5)
(5 4 3 3 3 3 3 3 3 4 5)
(5 4 4 4 4 4 4 4 4 5)
(5 5 5 5 5 5 5 5 5 5)
NIL
```

Решение 2.71.9

```
(defun 0111-row (n)
  (mapcar #'(lambda (a b) (loop repeat b collect a))
    (loop for a from 0 to n collect a)
    (loop for a from 1 to (+ n 2) by 2 collect a)))

(defun ttrim (w)
  (append (loop for a from 0 below (length w)
    collect (nthcdr a w))
    ` (,nil)))

(defun sn (n &aux (v (ttrim (loop for a from 1 to n collect a))))
  (mapcar #'nconc (mapcar #'reverse v) (0111-row (+ n n)) v))

(defun digits-number (w &optional (ac 0))
  (if w (digits-number (cdr w) (+ (car w) (* ac 10))) ac))

(defun sna (n &aux (w (sn n)))
  (mapcar #'print
    (mapcar #'digits-number (revappend (cdr w) w))) nil)

(defun snail (n)
  (when (< 0 n 10) (sna n)))

> (snail 5)
```

```
55555555555
54444444445
54333333345
54322222345
54321112345
54321012345
54321112345
54322222345
54333333345
54444444445
```

55555555555

NIL

Решение 2.71.10

```
(defun 101-row (n &aux (v (loop for a upto n collect a)))
  (nconc (reverse (cdr v)) v))

(defun down-rows (w
                  n
                  &optional
                  (m 1)
                  (a n)
                  (b (1+ n))
                  (v (copy-list w)))
  (when (<= m n) (cons (fill v m :start a :end b)
                        (down-rows w n (1+ m) (1- a) (1+ b)))))

(defun sna (n &aux (z (101-row n)) (v (down-rows z n)))
  (mapcar #'print (nconc (reverse v) `(,z) v)) nil)

(defun snail (n) (when (< 0 n 10) (sna n)))

> (snail 5)
```

```
(5 5 5 5 5 5 5 5 5 5)
(5 4 4 4 4 4 4 4 4 5)
(5 4 3 3 3 3 3 3 3 4 5)
(5 4 3 2 2 2 2 2 3 4 5)
(5 4 3 2 1 1 1 2 3 4 5)
(5 4 3 2 1 0 1 2 3 4 5)
(5 4 3 2 1 1 1 2 3 4 5)
(5 4 3 2 2 2 2 2 3 4 5)
(5 4 3 3 3 3 3 3 3 4 5)
(5 4 4 4 4 4 4 4 4 4 5)
(5 5 5 5 5 5 5 5 5 5 5)
NIL
```

Решение 2.71.11

```
(defun 101-row (n &aux (v (loop for a upto n collect a)))
  (nconc (reverse (cdr v)) v))

(defun down-rows (w
                  n
                  &optional
                  (m 1)
                  (a n)
                  (b (1+ n))
                  (v (copy-list w)))
  (when (<= m n) (cons (digits-number (fill v m :start a :end b))
                        (down-rows w n (1+ m) (1- a) (1+ b)))))
```

```

(defun digits-number (w &optional (ac 0))
  (if w (digits-number (cdr w) (+ (car w) (* ac 10))) ac))

(defun sna (n &aux (z (101-row n)) (v (down-rows z n)))
  (mapcar #'print (nconc (reverse v) ` (, (digits-number z)) v)) nil)

(defun snail (n) (when (< 0 n 10) (sna n)))

> (snail 1)

111
101
111
NIL

```

Решение 2.71.12

```

(defun 101-row (n &aux (v (loop for a upto n collect a)))
  (nconc (reverse (cdr v)) v))

(defun down-rows (w n &optional (m 1) (a n) (b (1+ a))
  (v (copy-list w)))
  (when (<= m n) (cons (fill v m :start a :end b)
    (down-rows w n (1+ m) (1- a) (1+ b)))))

(defun snail (n &aux (z (101-row n)) (v (down-rows z n)))
  (when (< 0 n 10)
    (mapcar #'print (nconc (reverse v) (cons z v))) nil))

> (snail 1)

(1 1 1)
(1 0 1)
(1 1 1)
NIL

```

Задача 2.72 *[[a][ba][aba]].lisp*

Определить функцию, которая возвращает список из n подсписков (A) (B A) (A B A) (B A B A) (A B A B A)

Решение 2.72.1

```

(defun [[a][ba][aba]] (n &optional (ac '(a)))
  (when (plusp n)
    (cons ac ([[a][ba][aba]] (1- n)
      (cons (if (eq (car ac) 'a) 'b 'a)
        ac))))))

> ([[a][ba][aba]] 5)
((A) (B A) (A B A) (B A B A) (A B A B A))

```

Задача 2.73 0102-list.lisp

What language is syntactically the best in the world? Like, is the most simple and coherent? Lisp, the answer is always Lisp.

Jarrod Jeffrey Ingram

Дано целое число n . Определить функцию, которая возвращает список строк "00" "01" "02" "03" "04" "05" "06" "0n".

Решение 2.73.1

```
(defun 0102-list (n &aux (m (1+ (length
                                   (write-to-string n)))))
  (loop for a from 0 to n collect (format01 m a)))

(defun format01 (n m)
  (format nil (concatenate 'string
                           "~"
                           (write-to-string n)
                           ", '0d")
          m))

> (0102-list 7)
("00" "01" "02" "03" "04" "05" "06" "07")
> (0102-list 17)
("000" "001" "002" "003" "004" "005" "006" "007" "008" "009" "010"
"011" "012" "013" "014" "015" "016" "017")
> (0102-list 107)
("0000" "0001" "0002" "0003" "0004" "0005" "0006" "0007" "0008" "0009"
"0010" "0011" "0012" "0013" "0014" "0015" "0016" "0017" "0018" "0019"
"0020" "0021" "0022" "0023" "0024" "0025" "0026" "0027" "0028" "0029"
"0030" "0031" "0032" "0033" "0034" "0035" "0036" "0037" "0038" "0039"
"0040" "0041" "0042" "0043" "0044" "0045" "0046" "0047" "0048" "0049"
"0050" "0051" "0052" "0053" "0054" "0055" "0056" "0057" "0058" "0059"
"0060" "0061" "0062" "0063" "0064" "0065" "0066" "0067" "0068" "0069"
"0070" "0071" "0072" "0073" "0074" "0075" "0076" "0077" "0078" "0079"
"0080" "0081" "0082" "0083" "0084" "0085" "0086" "0087" "0088" "0089"
"0090" "0091" "0092" "0093" "0094" "0095" "0096" "0097" "0098" "0099"
"0100" "0101" "0102" "0103" "0104" "0105" "0106" "0107")
```


Структура 3 (функция (СПИСОК)) > АТОМ

Although high general intelligence is common among hackers, it is not the sine qua non one might expect. Another trait is probably even more important: the ability to mentally absorb, retain, and reference large amounts of 'meaningless' detail, trusting to later experience to give it context and meaning. A person of merely average analytical intelligence who has this trait can become an effective hacker, but a creative genius who lacks it will swiftly find himself outdistanced by people who routinely upload the contents of thick reference manuals into their brains.

New Hacker's Dictionary

A major factor for me in using vi and now emacs is I never need to use my mouse. I despise picking my hand from the keyboard, watching it move over to the mouse, descend... and then move the little cursor.

~

There were two vocational skills that I learned that have helped me out over the years. One was learning to type and the other was learning Emacs.

Christopher Baus

Задача 3.1 *cantinier.lisp*

«Луций Калидий Эротик сделал при жизни гробницу себе и Фаннии Волупта-те. «Трактирщик, сосчитаемся!» — «Вина у тебя был один секстарий (0,544 литра), на один асс хлеба, на два асса закуски».— «Верно».— «За девочку восемь ассов».— «И это верно».— «Сена для мула на два асса».— «Этот мул меня доконает!»».

В Помпеях кубок простого вина стоил 1 асс, более хорошего качества 2 асса, фалернское вино 4 асса. Вино из виноградных выжимок было ещё дешевле.

Асс (устар. «ас», лат. as, assarius) — название древнеримской медной монеты, впервые появившейся, согласно преданию, во времена Сервия Туллия, но, согласно научным данным, — только в период Республики.

На аверсе изображался профиль одного из богов, например Юпитера (на самом ассе — Януса), на реверсе — корабельный нос (т. н. «морские» чеканы) с указанием номинала: точками обозначалось число унций на дробных частях асса; римские цифры чеканились на ассе и кратных ему монетах.

Сколько ассов Луций Калидий Эротик должен был заплатить трактирщику?

Решение 3.1.1

```
(defun cantinier (wine bread snack girl mule)
  (+ wine bread snack girl mule))
```

```
> (cantinier 1 1 2 8 2)
14
```

Решение 3.1.2

```
(defun cantinier (&rest assarius)
  (reduce #'+ assarius))
```

```
> (cantinier 1 1 2 8 2)
14
```

Задача 3.2 count-elm.lisp

Подсчитать количество элементов в списке.

Решение 3.2.1

```
(defun count-elm (a w)
  (cond ((null w) 0)
        ((equalp (car w) a) (1+ (count-elm a (cdr w))))
        ((count-elm a (cdr w)))))
```

```
> (count-elm 'b '(a b b a b a b))
4
```

Решение 3.2.2

```
(defun count-elm (a w &optional (n 0))
  (cond ((null w) n)
        ((equalp (car w) a) (count-elm a (cdr w) (1+ n)))
        (t (count-elm a (cdr w) n))))
```

```
> (count-elm 'b '(a b b a b a b))
4
```

Решение 3.2.3

```
(defun count-elm (a w &optional (n 0))
  (if w
```

```

(count-elm a
  (cdr w)
  (if (equalp (car w) a)
      (1+ n)
      n))

```

```

> (count-elm 'b '(a b b a b a b))
4

```

Решение 3.2.4

```

(defun count-elm (a w) (loop for e in w when (equalp e a) count e))

> (count-elm 'b '(a b b a b a b))
4

```

Решение 3.2.5

```

(defun count-elm (a w) (count a w))

> (count-elm 'b '(a b b a b a b))
4

```

Задача 3.3 sum137.lisp

Дан список. Найти сумму первого, третьего и седьмого элементов списка, если указанные элементы - числа, иначе - вернуть последний элемент списка.

Решение 3.3.1

```

(defun sum137 (w)
  (if (and (numberp (nth 0 w))
          (numberp (nth 2 w))
          (numberp (nth 6 w)))
      (+ (nth 0 w) (nth 2 w) (nth 6 w))
      (car (last w))))

> (sum137 '(1 2 3 4 5 6 7))
11

```

Комментарии (построчно):

1. Определяем функцию/defun sum137 с одним параметром w - список;
2. Если/if одновременно и/and числом/numberp является первый элемент списка w/(nth 0 w) (нумерация в списке начинается с 0),
3. Третий элемент списка w/(nth 2 w),
4. Седьмой элемент списка w/(nth 6 w),
5. То возвращается сумма/+ этих трех элементов/(+ (nth 0 w) (nth 2 w) (nth 6 w));
6. В случае невыполнения условия в строках 2-4: возвращается первый элемент/car списка, состоящего из последнего элемента/last списка w:

(car (last w)), то есть (last '(1 2 3 4 5 6 7)) вернет список из одного (последнего) элемента (7), и затем (car '(7)) вернет 7.

Решение 3.3.2

```
(defun sum137 (w)
  (if (and (numberp (first w))
          (numberp (third w))
          (numberp (seventh w)))
      (+ (first w) (third w) (seventh w))
      (car (last w))))
```

```
> (sum137 '(1 2 3 4 5 6 7))
11
```

Решение 3.3.3

```
(defun sum137 (w)
  (if (and (numberp (car w))
          (numberp (caddr w))
          (numberp (seventh w)))
      (+ (car w) (caddr w) (seventh w))
      (car (last w))))
```

```
> (sum137 '(1 2 3 4 5 6 7))
11
```

Решение 3.3.4

```
(defun sum137 (w &aux (v (list (car w) (caddr w) (seventh w))))
  (if (every #'numberp v) (reduce #'+ v) (car (last w))))
```

```
> (sum137 '(1 2 3 4 5 6 7))
11
```

Решение 3.3.5

```
(defun sum137 (w)
  (if (and (numberp (elt w 0))
          (numberp (elt w 2))
          (numberp (elt w 6)))
      (+ (elt w 0) (elt w 2) (elt w 6))
      (car (last w))))
```

```
> (sum137 '(1 2 3 4 5 6 7))
11
```

Задача 3.4 sum-odd-plus.lisp

Дан список. Составить программу, вычисляющую сумму всех положительных нечетных чисел.

Решение 3.4.1

```
(defun sum-odd-plus (w)
  (cond ((null w) 0)
        ((and (oddp (car w)) (plusp (car w)))
         (+ (car w) (sum-odd-plus (cdr w))))
        ((sum-odd-plus (cdr w)))))

> (sum-odd-plus '(0 5 6 7 8))
12
```

Задача 3.5 rare-num.lisp

Lisp is where most of other programming language features were stolen from. I'd say the only "real" thing coming from somewhere else is OOP - which was stolen from Smalltalk. And even there, you could use Common Lisp's CLOS - which is effectively Smalltalk's OOP with prototyping embedded (i.e. the same idea which Java Script is credited for) - only it got there in the 80s (long before JS was even a notion). Thus if you know Lisp, there's very little any other language can show you - at most others would just have different syntax or various rules to keep you on "their" track instead of "yours" (if that's useful is subjective).

Irné Barnard

Найти наименее часто встречающееся число в списке.

Решение 3.5.1

```
(defun rare-num (w)
  (when w (rar w (car w) (count (car w) w))))

(defun rar (w n f)
  (if w (let* ((a (car w)) (c (count a w)))
          (if (<= c f)
              (rar (delete a w) a c)
              (rar (delete a w) n f)))
    n))

> (rare-num '(2 1 2))
1
```

Решение 3.5.2

```
(defun rare-num (w &optional n (f (length w)))
  (if w (let* ((a (car w)) (c (count a w)))
```

```

      (if (<= c f)
          (rare-num (delete a w) a c)
          (rare-num (delete a w) n f)))
  n))

```

```

> (rare-num '(2 1 2))
1

```

Решение 3.5.3

```

(defun rare-num (w)
  (caar (sort (mapcar #'(lambda (a)
                          (cons a (count a w))) w) #'< :key #'cdr)))

> (rare-num '(2 1 2))
1

```

Решение 3.5.4

```

(defun rare-num (w &aux acc)
  (dolist (a w) (push (cons a (count a w)) acc))
  (caar (sort acc #'< :key #'cdr)))

> (rare-num '(2 1 2))
1

```

Решение 3.5.5

```

(defun rare-num (w)
  (caar (sort (loop for a in w collect
                    (cons a (count a w))) #'< :key #'cdr)))

> (rare-num '(2 1 2))
1

```

Задача 3.6 subp.lisp

Написать программу, проверяющую, является ли первое из двух (заданных в виде списков) множеств подмножеством второго.

Решение 3.6.1

```

(defun subp (v w)
  (cond ((null v)
        (member (car v) w) (subp (cdr v) w))
        (nil)))

> (subp '(0) '(1 2))
NIL
> (subp '(1) '(1 2))
T

```

Функция вернет `t` - истину, когда будет выполнено условие `(null v)` - исчерпан первый список, и до этого не войдет в третью строчку и не вернет `nil` - ложь.

Решение 3.6.2

```
(defun subp (v w)
  (or (null v) (and (member (car v) w) (subp (cdr v) w))))

> (subp '(0) '(1 2))
NIL
> (subp '(1) '(1 2))
T
```

Решение 3.6.3

```
(defun subp (v w)
  (or (null v) (when (-member (car v) w) (subp (cdr v) w))))

(defun -member (a w)
  (cond ((null w) nil)
        ((equalp a (car w)) (cons (car w) (cdr w)))
        ((-member a (cdr w)))))

> (subp '(0) '(1 2))
NIL
> (subp '(1) '(1 2))
T
```

Решение 3.6.4

```
(defun subp (v w)
  (cond ((null v)
        ((-member (car v) w) (subp (cdr v) w))))

(defun -member (a w)
  (cond ((null w) nil)
        ((equalp a (car w)) (cons (car w) (cdr w)))
        ((-member a (cdr w)))))

> (subp '(0) '(1 2))
NIL
> (subp '(1) '(1 2))
T
```

Задача 3.7 *uno-atom.lisp*

Если для построения списка и его подсписков был использован только один атом, выдать его, иначе выдать `nil`. Пример:

```
> (f '(a (a) (a a a) ((a))))
a
```

Решение 3.7.1

```
(defun uno-atom (w &aux (v (flat w)) (z (car v)))
  (when (every #'(lambda (a) (equal z a)) v) z))

(defun flat (w &optional acc)
  (cond ((null w) acc)
        ((atom w) (cons w acc))
        ((flat (car w) (flat (cdr w) acc)))))

> (uno-atom '(a (a) (a a a) ((a))))
A
```

Решение 3.7.2

```
(defun uno-atom (w &aux (v (flat w)) (z (car v)))
  (when (every #'(lambda (a) (equal z a)) v) z))

(defun flat (w)
  (loop for a in w
        if (listp a) append (flat a)
        else collect a))

> (uno-atom '(a (a) (a a a) ((a))))
A
```

Решение 3.7.3

```
(defun uno-atom (w &aux (v (flat w)) (z (car v)))
  (when (every #'(lambda (a) (equal z a)) v) z))

(defun flat (w)
  (mapcan #'(lambda (a) (if (listp a) (flat a) `(,a))) w))

> (uno-atom '(a (a) (a a a) ((a))))
A
```

Решение 3.7.4

```
(defun uno-atom (w &aux (v (flat w)) (z (car v)))
  (when (every #'(lambda (a) (equal z a)) v) z))

(defun flat (w)
  (mapcan #'(lambda (a) (if (atom a) `(,a) (flat a))) w))

> (uno-atom '(a (a) (a a a) ((a))))
A
```

Решение 3.7.5

```
(defun uno-atom (w &aux (v (flat w)) (z (car v)))
  (when (every #'(lambda (a) (equal z a)) v) z))
```



```
(defun flat (w)
  (if (atom w) `(,w) (mapcan #'flat w)))

> (uno-atom '(a (a) (a a a) (((a)))))
A
```

Задача 3.8 leader-atom.lisp

Написать функцию, формальным параметром которой является список. Результат функции – первый атом списка (в учет принимаются списки всех уровней).

Решение 3.8.1

```
(defun leader-atom (w)
  (cond ((null w) nil)
        ((listp (car w)) (or (leader-atom (car w))
                              (leader-atom (cdr w))))
        ((car w))))

> (leader-atom '((( )) (( ))((a)) (b c)) d))
A
```

Решение 3.8.2

```
(defun leader-atom (w &aux (a (car w)))
  (when w (if (listp a)
              (or (leader-atom a) (leader-atom (cdr w)))
              a)))

> (leader-atom '((( )) (( ))((a)) (b c)) d))
A
```

Задача 3.9 num-sym-3rd.lisp

Написать функцию f (x y z) которая проверяет: является ли третий элемент списка целым числом или символом.

Решение 3.9.1

```
(defun num-sym-3rd (w &aux (a (caddr w)))
  (cond ((null a) nil)
        ((symbolp a) (format nil "~a is a ~a" a "symbol"))
        ((integerp a) (format nil "~a is an ~a" a "integer"))
        ((format nil "~a is ~a" a "unknown"))))

> (num-sym-3rd '(a b c))
"C is a symbol"
> (num-sym-3rd '(1 2 3))
"3 is an integer"
```

Задача 3.10 *some-odd.lisp*

Функция возвращает *t*, если в одноуровневом списке есть хотя бы одно нечетное число.

Решение 3.10.1

```
(defun some-odd (w)
  (some #'(lambda (a) (and (numberp a) (oddp a))) w))

> (some-odd '(2 3))
T
> (some-odd '(2 4))
NIL
```

Задача 3.11 *last-odd.lisp*

Написать рекурсивную функцию, которая возвращает последнее нечетное число из числового списка.

Решение 3.11.1

```
(defun last-odd (w &optional (n 0) &aux (a (car w)) (d (cdr w)))
  (cond ((null w) n)
        ((oddp a) (last-odd d a))
        ((last-odd d n))))

(defun last-odd (w) (find-if #'oddp w :from-end t))

> (last-odd '(1 4 2 6 3 5 6 4 16))
5
```

Задача 3.12 *best-marks.lisp*

Дана база данных:

((a 5 5 5) (b 5 4 4) (c 4 4 3)), где

(a 5 5 5) – 1 ученик

(b 5 4 4) – 2 ученик

(c 4 4 3) – 3 ученик.

Определить количество отличников.

Решение 3.12.1

```
(defun best-marks (w &optional (n 0) &aux (a (car w)))
  (cond ((null w) n)
        ((= 5 (cadr a) (caddr a) (caddr a))
         (best-marks (cdr w) (+ n 1)))
        ((best-marks (cdr w) n))))

> (best-marks '((a 5 5 5) (b 5 4 4) (c 4 4 3)))
1
```

Задача 3.13 average-height.lisp

Имеется список студентов группы, в котором указаны фамилия студента, дата рождения, пол, рост и вес. Определить средний рост студентов мужского пола.

Решение 3.13.1

```
(defparameter w '((ivanov 92 m 180 80) (petrova 92 f 160 60) (petrov
92 m 190 90)))

(defun average-height (w)
  (let ((v (remove-if-not #'(lambda (a) (equal (caddr a) 'm)) w)))
    (/ (reduce #'+ v :key #'caddr) (length v))))

> (average-height w)
185
```

Задача 3.14 max-height.lisp

Имеется список студентов группы, в котором указаны фамилия студента, дата рождения, пол, рост и вес. Определить фамилию самого высокого юноши.

Решение 3.14.1

```
(defparameter w '((ivanov 92 m 180 80) (petrova 92 f 160 60) (petrov
92 m 190 90)))

(defun max-height (w) (caar (sort w #'> :key #'caddr)))

> (max-height w)
PETROV
```

Задача 3.15 counter.lisp

Определить функцию для подсчета количества вершин бинарного дерева значения которых лежат в определенном диапазоне.

Решение 3.15.1

```
(defun counter (x z w &aux (a (car w)) (d (cdr w)))
  (cond ((null w) 0)
        ((listp a) (+ (counter x z a) (counter x z d)))
        ((and a (<= x a z)) (1+ (counter x z d)))
        ((counter x z d))))

> (counter 1 3 '((2 (5 (nil 3 1) 7))))
3
```

Решение 3.15.2

```
(defun flat (w &optional acc)
  (cond ((null w) acc)
        ((atom w) (cons w acc))
        ((flat (car w) (flat (cdr w) acc)))))

(defun counter (x z w)
  (loop for a in (flat w) counting (<= x a z)))

> (counter 1 3 '((2 (5 (nil 3 1) 7))))
3
```

Решение 3.15.3

```
(defun flat (w)
  (mapcan #'(lambda (a) (if (listp a) (flat a) `(,a))) w))

(defun counter (x z w)
  (count-if #'(lambda (a) (<= x a z)) (flat w)))

> (counter 1 3 '((2 (5 (nil 3 1) 7))))
3
```

Решение 3.15.4

```
(defun flat (w)
  (mapcan #'(lambda (a) (if (atom a) `(,a) (flat a))) w))

(defun counter (x z w)
  (count-if #'(lambda (a) (<= x a z)) (flat w)))

> (counter 1 3 '((2 (5 (nil 3 1) 7))))
3
```

Решение 3.15.5

```
(defun flat (w)
  (if (atom w) `(,w) (mapcan #'flat w)))

(defun counter (x z w)
  (loop for a in (flat w) counting (<= x a z)))

> (counter 1 3 '((2 (5 (nil 3 1) 7))))
3
```

Задача 3.16 last-atom.lisp

Написать функцию, которая находит последний атом на верхнем уровне списка.

Решение 3.16.1

```
(defun last-atom (w) (find-if #'atom w :from-end t))

> (last-atom '((a b c) d e (f g h)))
E
```

Задача 3.17 number-max.lisp

Определить функцию, вычисляющую максимальный элемент в многоуровневом списке, состоящем из числовых атомов.

Решение 3.17.1

```
(defun maximus (w ac)
  (cond ((null w) (car ac))
        ((listp (car w))
         (maximus (car w) (list (maximus (cdr w) ac))))
        ((null (car ac)) (maximus (cdr w) (list (car w))))
        ((> (car w) (car ac)) (maximus (cdr w) (list (car w))))
        ((maximus (cdr w) ac))))

(defun number-max (w)
  (maximus w nil))

> (number-max '(1 2 3 (4 5) (((6))) 4 5))
6
```

Задача 3.18 almost-last.lisp

Определить функцию, которая выдает в качестве результата предпоследний элемент списка, если он есть, а иначе nil.

Решение 3.18.1

```
(defun almost-last (w)
  (cond ((null (cddr w)) (when (cdr w) (car w)))
        ((almost-last (cdr w)))))

> (almost-last '(1 2))
1
> (almost-last '(1))
NIL
```

Решение 3.18.2

```
(defun almost-last (w)
  (when (cdr w) (car (last w 2))))

> (almost-last '(1 2))
1
> (almost-last '(1))
NIL
```

Решение 3.18.3

```
(defun almost-last (w)
  (cadr (reverse w)))

> (almost-last '(1 2))
1
> (almost-last '(1))
NIL
```

Задача 3.19 *palindrome.lisp*

Есть список из атомов, имена которых – буквы. Написать функцию, которая выдает значение истина `t`, если слово из этих букв является палиндромом и ложь `nil` в противном случае.

Решение 3.19.1

```
(defun palindrome (w)
  (equal w (reverse w)))

> (palindrome '(a b c b a))
T
> (palindrome "abba")
T
```

Задача 3.20 *not-equal-last.lisp*

Написать программу, которая подсчитывала бы количество элементов в списке отличных от последнего.

Решение 3.20.1

```
(defun not-equal-last (w)
  (cond ((null (cdr w)) 0)
        ((equalp (car w) (car (last w))) (not-equal-last (cdr w)))
        ((1+ (not-equal-last (cdr w))))))

> (not-equal-last '(a b b a))
2
```

Задача 3.21 *sum-number.lisp*

Дан список произвольной структуры, в который входят как числа, так и символьные атомы. Написать функцию, которая возвращает список из двух чисел, первое из них равно сумме чисел исходного списка, а второе их количеству.

Решение 3.21.1

```

(defun flat (w acc)
  (cond ((null w) acc)
        ((atom w) (cons w acc))
        ((flat (car w) (flat (cdr w) acc)))))

(defun sum-number (w
                  &aux
                  (v (remove-if-not #'numberp (flat w nil))))
  (values (reduce #'+ v) (length v)))

> (sum-number '(a (2) ((3) b 4) 5))
14
4

```

Решение 3.21.2

```

(defun flat (w &optional acc)
  (cond ((null w) acc)
        ((atom w) (cons w acc))
        ((flat (car w) (flat (cdr w) acc)))))

(defun sum-number (w
                  &aux
                  (v (remove-if-not #'numberp (flat w nil))))
  (values (reduce #'+ v) (length v)))

> (sum-number '(a (2) ((3) b 4) 5))
14
4

```

Решение 3.21.3

```

(defun flat (w)
  (mapcan #'(lambda (a) (if (listp a) (flat a) `(,a)))
          w))

(defun sum-number (w
                  &aux
                  (v (remove-if-not #'numberp (flat w nil))))
  (values (reduce #'+ v) (length v)))

> (sum-number '(a (2) ((3) b 4) 5))
14
4

```

Решение 3.21.4

```

(defun flat (w)
  (loop for a in w
        if (listp a) append (flat a)
        else collect a))

```

```
(defun sum-number (w
                  &aux
                  (v (remove-if-not #'numberp (flat w nil))))
  (values (reduce #'+ v) (length v)))
> (sum-number '(a (2) ((3) b 4) 5))
14
4
```

Решение 3.21.5

```
(defun flat (w)
  (mapcan #'(lambda (a) (if (atom a) `(:,a) (flat a)))
          w))

(defun sum-number (w
                  &aux
                  (v (remove-if-not #'numberp (flat w nil))))
  (values (reduce #'+ v) (length v)))
> (sum-number '(a (2) ((3) b 4) 5))
14
4
```

Решение 3.21.6

```
(defun flat (w)
  (if (atom w) `(:,w)
      (mapcan #'flat w)))

(defun sum-number (w
                  &aux
                  (v (remove-if-not #'numberp (flat w nil))))
  (values (reduce #'+ v) (length v)))
> (sum-number '(a (2) ((3) b 4) 5))
14
4
```

Задача 3.22 *equal-set.lisp*

Определите предикат равенство-множеств, проверяющий совпадение двух множеств (независимо от порядка следования элементов).

Решение 3.22.1 (автор - VH, www.cyberforum.ru)

```
(defun equal-set (w v)
  (cond ((null w) (null v))
        ((member (car w) v) (equal-set (cdr w) (remove (car w) v)))))
> (equal-set '(1 2) '(2 1))
T
> (equal-set '(1 2) '(2 2))
NIL
```


Задача 3.23 *sum-plusp.lisp*

Найти сумму неотрицательных элементов заданного списка целых чисел.

Решение 3.23.1

```
(defun sum-plusp (w &aux (a (car w)) (d (cdr w)))
  (cond ((null w) 0)
        ((plusp a) (+ a (sum-plusp d)))
        ((sum-plusp d))))
```

```
> (sum-plusp '(2 -3 4 -1 -6))
6
```

Задача 3.24 *tops-number.lisp*

Определить функцию вычисления количества атомов на верхнем уровне многоуровневого списка.

Решение 3.24.1

```
(defun tops-number (w)
  (cond ((null w) 0)
        ((atom (car w)) (1+ (tops-number (cdr w))))
        ((tops-number (cdr w)))))
```

```
> (tops-number '(a (a) a (a a) a))
3
```

Решение 3.24.2

```
(defun tops-number (w) (if w (1+ (tops-number (cdr w))) 0))
```

```
> (tops-number '(a (a) a (a a) a))
5
```

Решение 3.24.3

```
(defun tops-number (w)
  (cond ((null w) 0) ((1+ (tops-number (cdr w))))))
```

```
> (tops-number '(a (a) a (a a) a))
5
```

Задача 3.25 *sum-unique.lisp*

Определить функцию, которая выводит сумму разных элементов списка.

Решение 3.25.1

```
(defun sum-unique (w)
```

```
(cond ((null w) 0)
      ((member (car w) (cdr w)) (sum-unique (cdr w)))
      ((+ (car w) (sum-unique (cdr w)))))
```

```
> (sum-unique '(1 2 3 1 2 3))
6
```

Решение 3.25.2

```
(defun sum-unique (w)
  (reduce #'(lambda (x y) (+ x y)) (remove-duplicates w)))
```

```
> (sum-unique '(1 2 3 1 2 3))
6
```

Задача 3.26 *our-car-cdaddr.lisp*

Составьте композицию из функций CAR и CDR, для которой результатом применения этой композиции к списку (1 (2) ((3 4) 5)) будет 5.

Решение 3.26.1

```
(defun our-car-cdaddr (w) (car (cdr (car (cdr (cdr w))))))
```

```
> (our-car-cdaddr '(1 (2) ((3 4) 5)))
5
```

Задача 3.27 *descend.lisp*

Проверять является ли отсортированным в порядке убывания список, состоящий из чисел.

Решение 3.27.1

```
(defun descend (w)
  (cond ((null (cdr w)) t)
        ((< (car w) (cadr w)) nil)
        ((descend (cdr w)))))
```

```
> (descend '(3 2 2 1))
T
```

Решение 3.27.2

```
(defun descend (w)
  (cond ((null (cdr w)) t)
        ((< (car w) (cadr w)) nil)
        ((descend (cdr w)))))
```

```
> (descend '(3 2 2 1))
T
```

Решение 3.27.3

```
(defun descend (w)
  (if (null (cdr w)) t (and (>= (car w) (cadr w)) (descend (cdr w)))))

> (descend '(3 2 2 1))
T
> (descend '(3 2 2 10))
NIL
```

Решение 3.27.4

```
(defun descend (w)
  (if (cdr w) (and (>= (car w) (cadr w)) (descend (cdr w))) t))

> (descend '(3 2 2 1))
T
> (descend '(3 2 2 10))
NIL
```

Решение 3.27.5

```
(defun descend (w) (apply #'> w))

> (descend '(3 2 2 1))
T
> (descend '(3 2 2 10))
NIL
```

Задача 3.28 our-caddr.lisp

Определите функцию (caddr x) с помощью базовых функций. Используйте имя our-caddr, чтобы не переопределять одноименную встроенную функцию Лиспа.

Решение 3.28.1

```
(defun our-caddr (w) (car (cdr (cdr w))))

> (our-caddr '(1 2 3))
3
```

Задача 3.29 count-low-atoms.lisp

Задан многоуровневый список, найти количество атомов списка, находящихся на нижних уровнях.

Решение 3.29.1

```
(defun del-upper-atoms (w)
```

```

(cond ((null w) nil)
      ((atom (car w)) (del-upper-atoms (cdr w)))
      ((cons (car w) (del-upper-atoms (cdr w))))))

(defun flatten (w &optional ac)
  (cond ((null w) ac)
        ((atom w) (cons w ac))
        ((flatten (car w) (flatten (cdr w) ac)))))

(defun count-atoms (w &optional (n 0))
  (cond ((null w) n)
        ((count-atoms (cdr w) (1+ n)))))

(defun count-low-atoms (w)
  (count-atoms (flatten (del-upper-atoms w))))

> (count-low-atoms '((a)))
1

```

Решение 3.29.2

```

(defun flatten (w)
  (mapcan #'(lambda (a) (if (listp a) (flatten a) `(,a))) w))

(defun count-low-atoms (w)
  (count-if #'atom (flatten (delete-if #'atom w))))

> (count-low-atoms '((a)))
1

```

Решение 3.29.3

```

(defun flatten (w)
  (loop for a in w
        if (listp a) append (flatten a)
        else collect a))

(defun count-low-atoms (w)
  (count-if #'atom (flatten (delete-if #'atom w))))

> (count-low-atoms '((a)))
1

```

Решение 3.29.3

```

(defun flatten (w)
  (mapcan #'(lambda (a) (if (atom a) `(,a) (flatten a)))
          w))

(defun count-low-atoms (w)
  (count-if #'atom (flatten (delete-if #'atom w))))

> (count-low-atoms '((a)))
1

```

1

Решение 3.29.3

```
(defun flatten (w)
  (if (atom w) `(,w) (mapcan #'flatten w)))

(defun count-low-atoms (w)
  (count-if #'atom (flatten (delete-if #'atom w))))

> (count-low-atoms '((a)))
1
```

Задача 3.30 even.lisp

If you come from an imperative/procedural world, and you plan to learn Lisp and use Lisp's functional style, Lisp will help you learn very useful concepts that you may even be able to apply when you go back to your imperative language (or perhaps will never go back again). Some of these concepts are: higher order functions (pass functions around as if they were data), recursion (particularly with tail call optimization, which allows you to recurse indefinitely without running out of stack), lexical closures (pass state around safely), continuations (continuation passing style is a style of programming where the program flow is passed as a parameter of the function) and macros (the godly experience of creating a Domain Specific Languages that can make you far more productive by letting the Lisp runtime type your programs for you).

Flavio Villanustre

Определять, является ли сумма элементов списка из целых чисел, четным числом.

Решение 3.30.1

```
(defun even (w &optional (ac 0))
  (cond ((null w) (evenp ac))
        ((even (cdr w) (+ ac (car w))))))

> (even '(1 1))
T
```

Задача 3.31 *sum-odd.lisp*

Подсчитать сумму всех нечётных элементов списка (по месту нахождения).

Решение 3.31.1

```
(defun sum-odd (w &optional (ac 0))
  (cond ((null w) ac)
        ((oddp (car w)) (sum-odd (cdr w) (+ ac (car w))))
        ((sum-odd (cdr w) ac))))
```

```
> (sum-odd '(1 2 3 4))
4
```

Задача 3.32 *sum-even.lisp*

Подсчитать сумму всех чётных элементов списка.

Решение 3.32.1

```
(defun sum-even (w)
  (cond ((null w) 0)
        ((evenp (car w)) (+ (car w) (sum-even (cdr w))))
        ((sum-even (cdr w)))))
```

```
> (sum-even '(1 2 3 4))
6
```

Решение 3.32.2

```
(defun sum-even (w &aux (a (car w)))
  (if w (+ (if (evenp a) a 0) (sum-even (cdr w))) 0))
```

```
> (sum-even '(1 2 3 4))
6
```

Решение 3.32.3

```
(defun sum-even (w &optional (ac 0))
  (cond ((null w) ac)
        ((evenp (car w)) (sum-even (cdr w) (+ ac (car w))))
        ((sum-even (cdr w) ac))))
```

```
> (sum-even '(1 2 3 4))
6
```

Решение 3.32.4

```
(defun sum-even (w)
  (loop for a in w when (evenp a) sum a))
```

```
> (sum-even '(1 2 3 4))
```

6

Решение 3.32.5

```
(defun sum-even (w)
  (apply #'+ (remove-if-not #'evenp w)))

> (sum-even '(1 2 3 4))
6
```

Задача 3.33 count-zero.lisp

Заданы четыре числа *a*, *b*, *c*, *d*. Определить количество нулевых значений.

Решение 3.33.1

```
(defun count-zero (a b c d)
  (count-if #'zerop (list a b c d)))

> (count-zero 1 0 1 0)
2
```

Решение 3.33.2

```
(defun count-zero (a b c d)
  (+ (if (zerop a) 1 0)
     (if (zerop b) 1 0)
     (if (zerop c) 1 0)
     (if (zerop d) 1 0)))

> (count-zero 1 0 1 0)
2
```

Решение 3.33.3

```
(defun count-zero (w &optional (n 0))
  (cond ((null w) n)
        ((zerop (car w)) (count-zero (cdr w) (1+ n)))
        ((count-zero (cdr w) n))))

> (count-zero '(1 0 1 0))
2
```

Решение 3.33.4

```
(defun count-x (w &optional (n 0))
  (cond ((null w) n)
        ((zerop (car w)) (count-x (cdr w) (1+ n)))
        ((count-x (cdr w) n))))

(defun count-zero (a b c d)
```

```
(count-x (list a b c d)))
```

```
> (count-zero 1 0 1 0)
2
```

Решение 3.33.5

```
(defun count-x (w &optional (n 0))
  (cond ((null w) n)
        ((zerop (car w)) (count-x (cdr w) (1+ n)))
        ((count-x (cdr w) n))))
```

```
(defun count-zero (&rest w)
  (count-x w))
```

```
> (count-zero 1 0 1 0)
2
```

Решение 3.33.6

```
(defun count-zero (a b c d &optional (w (list a b c d)) (n 0))
  (cond ((null w) n)
        ((zerop (car w)) (count-zero a b c d (cdr w) (1+ n)))
        ((count-zero a b c d (cdr w) n))))
```

```
> (count-zero 0 2 3 0)
2
```

Решение 3.33.7

```
(defun count-zero (a b c d)
  (reduce #'+ (mapcar #'(lambda (e) (if (zerop e) 1 0))
                     (list a b c d)))))
```

```
> (count-zero 0 1 0 2)
2
```

Решение 3.33.8

```
(defun count-zero (a b c d)
  (labels ((counter (v n)
            (cond ((null v) n)
                  ((zerop (car v)) (counter (cdr v) (1+ n)))
                  ((counter (cdr v) n)))))
    (counter (list a b c d) 0)))
```

```
> (count-zero 0 1 0 2)
2
```

Решение 3.33.9

```
(defun count-zero (&rest w)
  (labels ((counter (v n)
```



```

      (cond ((null v) n)
            ((zerop (car v)) (counter (cdr v) (1+ n)))
            ((counter (cdr v) n))))
  (counter w 0)))

> (count-zero 0 1 0 2)
2

```

Задача 3.34 *product-even.lisp*

An interesting observation is how, once you start grasping the concepts of the functional style, your programs start to look different: you start focusing more in data structures to drive your program rather than control structures, and you realize that even the layout of your old imperative programs looks odd; your old procedures would probably look like an inside-out version of your new functional style functions.

Flavio Villanustre

Дан список целых чисел. Составить программу, вычисляющую произведение всех четных элементов списка за исключением нуля.

Решение 3.34.1

```

(defun product-even (w)
  (reduce #'* (remove-if #'zerop (remove-if-not #'evenp w))))

> (product-even '(0 1 2 3 4))
8

```

Задача 3.35 *average-cadr.lisp*

Напишите функцию для вычисления среднего возраста группы людей, которая описывается ассоциативным списком, например таким — ((mark 40) (kathy 30) (sally 57) (doran 35) (anthony 3)), здесь возраст является вторым элементом подписка.

Решение 3.35.1

```

(defun sm (w m n)
  (cond ((null w) (/ m n))
        ((sm (cdr w) (+ m (cadr w)) (1+ n)))))

(defun average-cadr (w)
  (sm w 0 0))

```

```
> (average-cadr '((mark 40) (kathy 30) (sally 57) (doran 35) (anthony
3)))
33
```

Задача 3.36 *binary-arithmetic.lisp*

Написать функцию сложения, вычитания, умножения и деления двоичных чисел.

Решение 3.36.1

```
(defun binary-arithmetic (w)
  (read-from-string
   (format
    nil "~b"
    (reduce
     (car w)
     (mapcar
      #'(lambda (a)
          (parse-integer (write-to-string a) :radix 2))
      (cdr w))))))

> (binary-arithmetic '(+ 1011 111001 101))
1001001
7
> (binary-arithmetic '(- 1011 111001 101))
-110011
7
> (binary-arithmetic '(* 1011 111001 101))
110000111111
12
> (binary-arithmetic '(/ 1011 111001 101))
1011/100011101
14
```

Решение 3.36.2

```
(defun binary-arithmetic (w)
  (read-from-string
   (format
    nil "~b"
    (reduce
     (car w)
     (mapcar
      #'(lambda (a)
          (read-from-string
           (concatenate 'string "#b" (write-to-string a))))
      (cdr w))))))

> (binary-arithmetic '(+ 1011 111001 101))
1001001
7
```

```

> (binary-arithmetic '(- 1011 111001 101))
-110011
7
> (binary-arithmetic '(* 1011 111001 101))
110000111111
12
> (binary-arithmetic '(/ 1011 111001 101))
1011/100011101
14

```

Задача 3.37 *binary+.lisp*

Определить функцию сложения двух двоичных чисел, представленных в виде списков.

Решение 3.37.1

```

(defun deci-bin (n &optional ac)
  (if (zerop n)
      ac
      (deci-bin (floor n 2) (cons (mod n 2) ac))))

(defun bin-deci (w &optional (n 0))
  (if (null w) n (bin-deci (cdr w) (+ (car w) n n))))

(defun bin-deci (w &optional (n 0))
  (if (null w)
      n
      (bin-deci (cdr w) (+ (car w) n n))))

> (binary+ '(1 0 1 1) '(1 1 1 0 0 1) '(1 0 1))
(1 0 0 1 0 0 1)

```

Решение 3.37.2

```

(defun deci-bin (n &optional ac)
  (if (zerop n)
      ac
      (deci-bin (floor n 2) (cons (mod n 2) ac))))

(defun bin (w)
  (read-from-string
   (apply
    #'concatenate
    'string
    "#b"
    (mapcar
     #'(lambda (a)
         (write-to-string a))
      w))))

(defun binary+ (&rest w)

```

```
(deci-bin (reduce #'(lambda (w) (mapcar #'bin w)))))

> (binary+ '(1 0 1 1) '(1 1 1 0 0 1) '(1 0 1))
(1 0 0 1 0 0 1)
```

Задача 3.38 count-atoms.lisp

Напишите функцию, считающую полное количество атомов (не равных nil) в многоуровневом списке.

Решение 3.38.1

```
(defun count-atoms (w &optional (n 0))
  (cond ((null w) n)
        ((atom (car w)) (count-atoms (cdr w) (1+ n)))
        ((count-atoms (car w) (count-atoms (cdr w) n)))))

> (count-atoms '(a (b c (d) e)))
5
```

Комментарии (построчно):

1. Определяем функцию count-atoms с параметрами: w - список и опциональным параметром, по умолчанию 0.
2. Если список пуст, то возвращаем n.
3. Если первый элемент списка это атом, рекурсивно вызываем эту же функцию count-atoms с хвостом списка и увеличенным на единицу n.
4. Если не выполнено ни одно из условий, то рекурсивно вызываем count-atoms с параметрами: голова списка (список) и рекурсивным вызовом функции count-atoms с параметрами хвост списка и n.

Решение 3.38.2

```
(defun count-atoms (w)
  (cond ((null w) 0)
        ((atom w) 1)
        ((+ (count-atoms (car w)) (count-atoms (cdr w))))))

> (count-atoms '(a (b c (d) e)))
5
```

Комментарии (построчно):

1. Определяем функцию count-atoms с параметром w - список.
2. Если список пуст, то возвращаем 0.
3. Если список это атом, то возвращаем 1.
4. Если не выполнено ни одно из условий, то возвращаем сумму рекурсивного вызова count-atoms с параметром голова списка и рекурсивного вызова count-atoms с параметром хвост списка.

Решение 3.38.3

```
(defun count-atoms (w)
  (loop for a in w sum (if (atom a) 1 (count-atoms a))))

> (count-atoms '(a (b c (d) e)))
5
```

Комментарии (построчно):

1. Определяем функцию count-atoms с параметром w – список.
 2. Бежим по списку, суммируя: если элемент атом, то единицы, если элемент список, то результаты рекурсивного вызова count-atoms с параметром список.

Задача 3.39 sum-137-last.lisp

Дан список. Найти сумму 1,3 и 7 элемента, если это цифры, иначе вернуть последний элемент списка.

Решение 3.39.1

```
(defun sum-137-last
  (w &aux (a (first w)) (b (third w)) (c (seventh w)))
  (if (every #'numberp (list a b c)) (+ a b c) (car (last w))))

> (sum-137-last '(1 2 3 4 5 6 7))
11
```

Задача 3.40 double-plusp.lisp

Дан список целых чисел. Получить удвоенную сумму всех положительных элементов списка.

Решение 3.40.1

```
(defun double-plusp (w)
  (* (apply #'+ (delete-if-not #'plusp w)) 2))

> (double-plusp '(1 -2 3))
8
```

Задача 3.41 avarage-num.lisp

Найти среднее арифметическое списка вещественных чисел.

Решение 3.41.1

```
(defun avarage-num (w)
  (when w (/ (reduce #'+ w) (count-if #'realp w))))

> (avarage-num '(11 12 13))
12
```

Решение 3.41.2

```
(defun avarage-num (w)
  (when w (/ (reduce #'+ w) (length w))))

> (avarage-num '(11 12 13))
12
```

Задача 3.42 *more-nums.lisp*

Написать функцию, которая принимает значение *t*, если в списке больше чисел, чем атомов.

Решение 3.42.1

```
(defun more-nums (w &optional (n 0))
  (cond ((null w) (plusp n))
        ((numberp (car w)) (more-nums (cdr w) (1+ n)))
        ((more-nums (cdr w) (1- n)))))

> (more-nums '(1 1 a))
T
```

Задача 3.43 *list2-3.lisp*

Напишите функцию, которая выдает истину, если ее аргумент удовлетворяет хотя бы одному из следующих условий:

- а) является списком из двух элементов;
- б) является списком из двух атомов;
- в) является списком из трех элементов.

Решение 3.43.1

```
(defun list2-3 (x)
  (and (consp x) (cdr x) (null (cdddr x))))

> (list2-3 '(a))
NIL
> (list2-3 '(a b))
T
> (list2-3 '(a b c))
T
```

Задача 3.44 *matrix-main.lisp*

Определить функцию, вычисляющую сумму элементов главной диагонали матрицы.

Решение 3.44.1

```

(defun matrix-main (w &optional (n 0))
  (if (null w) 0
      (+ (nth n (car w))
         (matrix-main (cdr w) (1+ n)))))

> (setq a '((1 0 0) (0 1 0) (0 0 1)))
((1 0 0) (0 1 0) (0 0 1))
> (matrix-main a)
3
> (setf a '((1 0 0 0) (0 1 0 0) (0 0 1 0) (0 0 0 1)))
((1 0 0 0) (0 1 0 0) (0 0 1 0) (0 0 0 1))
> (matrix-main a)
4

```

Решение 3.44.2

```

(defun matrix-main (w)
  (loop for a in w
        for b upfrom 0
        sum (nth b a)))

> (matrix-main '((1 0 0) (0 1 0) (0 0 1)))
3
> (matrix-main '((1 0 0 0) (0 1 0 0) (0 0 1 0) (0 0 0 1)))
4

```

Решение 3.44.3

```

(defun matrix-main (w)
  (loop with b = -1 for a in w do
        (incf b)
        sum (nth b a)))

> (matrix-main '((1 0 0) (0 1 0) (0 0 1)))
3
> (matrix-main '((1 0 0 0) (0 1 0 0) (0 0 1 0) (0 0 0 1)))
4

```

Решение 3.44.4

```

(defun matrix-main (w)
  (loop with b = -1 for a in w
        sum (nth (incf b) a)))

> (matrix-main '((1 0 0) (0 1 0) (0 0 1)))
3
> (matrix-main '((1 0 0 0) (0 1 0 0) (0 0 1 0) (0 0 0 1)))
4

```

Решение 3.44.5

```

(defun matrix-main (w &aux (n -1))
  (reduce #' + w :key #'(lambda (a) (nth (incf n) a))))

```

```
> (matrix-main '((1 0 0) (0 1 0) (0 0 1)))
3
> (matrix-main '((1 0 0 0) (0 1 0 0) (0 0 1 0) (0 0 0 1)))
4
```

Задача 3.45 *max-depth-0.lisp*

It completely breaks the imperative programming mindset. You just have to think about everything completely differently, because you're manipulating atoms on lists which may themselves be lists...

Stan Hanks

Определить функцию, вычисляющую максимальный уровень вложенности списка, считая с верхнего - нулевого уровня.

Решение 3.45.1

```
(defun max-depth-0 (w)
  (cond ((null w) 0)
        ((atom (car w)) (max-depth-0 (cdr w)))
        ((max (1+ (max-depth-0 (car w))) (max-depth-0 (cdr w))))))

> (max-depth-0 '(1 2 3))
0
> (max-depth-0 '(1 ((2)) 3))
2
```

Решение 3.45.2

```
(defun max-depth-0 (w) (if w (depth w) 0))

(defun depth (w)
  (if (atom w) -1 (max (1+ (depth (car w))) (depth (cdr w)))))

> (max-depth-0 '(a b (c d (e f) g)))
2
> (max-depth-0 '(a))
0
> (max-depth-0 '())
0
```

Задача 3.46 *evenp-sum.lisp*

Написать программу, определяющую, является ли сумма элементов многоуровневого списка целых чисел четным числом.

Решение 3.46.1


```

(defun flat (w &optional acc)
  (cond ((null w) acc)
        ((atom w) (cons w acc))
        ((flat (car w) (flat (cdr w) acc)))))

(defun sum (w)
  (if (null w)
      0
      (+ (car w) (sum (cdr w)))))

(defun evenp-sum (w)
  (evenp (sum (flat w))))

> (evenp-sum '(1 (2 (3)) (4)))
T
> (evenp-sum '(0 (2 (3)) (4)))
NIL

```

Решение 3.46.2

```

(defun sum (w)
  (cond ((null w) 0)
        ((atom (car w)) (+ (car w) (sum (cdr w))))
        ((+ (sum (car w)) (sum (cdr w))))))

(defun evenp-sum (w)
  (evenp (sum w)))

> (evenp-sum '(1 (2 (3)) (4)))
T
> (evenp-sum '(0 (2 (3)) (4)))
NIL

```

Решение 3.46.3

```

(defun sum (w)
  (cond ((null w) 0)
        ((atom w) w)
        ((+ (sum (car w)) (sum (cdr w)))))

(defun evenp-sum (w)
  (evenp (sum w)))

> (evenp-sum '(1 (2 (3)) (4)))
T
> (evenp-sum '(0 (2 (3)) (4)))
NIL

```

Решение 3.46.4

```

(defun flat (w)
  (loop for a in w
        if (listp a) append (flat a)

```

```

        else collect a))

(defun evenp-sum (w)
  (evenp (loop for a in (flat w)
               summing a)))

> (evenp-sum '(1 (2 (3)) (4)))
T
> (evenp-sum '(0 (2 (3)) (4)))
NIL

```

Решение 3.46.5

```

(defun flat (w)
  (mapcan #'(lambda (a) (if (listp a) (flat a) `(,a)))
          w))

(defun evenp-sum (w)
  (evenp (reduce #'+ (flat w))))

> (evenp-sum '(1 (2 (3)) (4)))
T
> (evenp-sum '(0 (2 (3)) (4)))
NIL

```

Решение 3.46.6

```

(defun flat (w)
  (mapcan #'(lambda (a) (if (atom a) `(,a) (flat a)))
          w))

(defun evenp-sum (w)
  (evenp (reduce #'+ (flat w))))

> (evenp-sum '(1 (2 (3)) (4)))
T
> (evenp-sum '(0 (2 (3)) (4)))
NIL

```

Решение 3.46.7

```

(defun flat (w)
  (if (atom w) `(,w) (mapcan #'flat w)))

(defun evenp-sum (w)
  (evenp (reduce #'+ (flat w))))

> (evenp-sum '(1 (2 (3)) (4)))
T
> (evenp-sum '(0 (2 (3)) (4)))
NIL

```

Задача 3.47 *|parens left|.lisp*

Определить функцию, проверяющую, что все левые скобки в s-выражении, представляющем список, являются крайними левыми скобками.

Решение 3.47.1

```
(defun |parens left| (w)
  (if w
    (and (or (atom (car w)) (|parens left| (car w)))
         (every #'atom (cdr w)))
    t))

> (|parens left| '(((3 4) 5) 6 7) 1))
T
> (|parens left| '(((3 4) 5) (6 7) 1)))
NIL
```

Решение 3.47.2

```
(defun |parens left| (w)
  (cond ((null w) t)
        ((every #'atom (cdr w))
         (or (atom (car w)) (|parens left| (car w))))
        (t nil)))

> (|parens left| '(((3 4) 5) 6 7) 1))
T
> (|parens left| '(((3 4) 5) (6 7) 1)))
NIL
```

Решение 3.47.3

```
(defun atoms (w)
  (every #'atom w))

(defun |parens left| (w)
  (cond ((null w) t)
        ((atoms (cdr w)) (or (atom (car w)) (|parens left| (car w))))
        (t nil)))

> (|parens left| '(((3 4) 5) 6 7) 1))
T
> (|parens left| '(((3 4) 5) (6 7) 1)))
NIL
```

Решение 3.47.4

```
(defun atoms (w)
  (cond ((null w) t)
        ((atom (car w)) (atoms (cdr w)))
        (t nil)))
```

```
(defun |parens left| (w)
  (cond ((null w) t)
        ((atoms (cdr w)) (or (atom (car w)) (|parens left| (car w))))
        (t nil)))
```

```
> (|parens left| '(((3 4) 5) 6 7) 1))
T
> (|parens left| '(((3 4) 5) (6 7) 1)))
NIL
```

Решение 3.47.5

```
(defun |parens left| (w)
  (cond ((cdr w) t)
        ((and (not (atom (car w))) (cdr w)) (|parens left| (car w)))
        (t nil)))
```

```
> (|parens left| '(((3 4) 5) 6 7) 1))
T
> (|parens left| '(((3 4) 5) (6 7) 1)))
NIL
```

Решение 3.47.6

```
(defun atoms (w)
  (cond ((null w) t)
        ((atom (car w)) (atoms (cdr w)))
        (t nil)))

(defun |parens left| (w)
  (cond ((cdr w) t)
        ((or (atom (car w)) (null (cdr w))) nil)
        ((|parens left| (car w)))))
```

```
> (|parens left| '(((3 4) 5) 6 7) 1))
T
> (|parens left| '(((3 4) 5) (6 7) 1)))
NIL
```

Решение 3.47.7

```
(defun |parens left| (w)
  (cond ((null w) t)
        ((every #'atom (cdr w))
         (or (atom (car w)) (|parens left| (car w))))
        (nil)))
```

```
> (|parens left| '(((3 4) 5) 6 7) 1))
T
> (|parens left| '(((3 4) 5) (6 7) 1)))
NIL
```

Задача 3.48 max-occur.lisp

Определить функцию, которая находит элемент с максимальным числом вхождений в список.

Решение 3.48.1

```
(defun max-occur (w &optional (n 0) z &aux (m (count (car w) w)))
  (cond ((null w) z)
        ((> m n) (max-occur (cdr w) m (car w)))
        ((max-occur (cdr w) n z))))
```

```
> (max-occur '(1 1 1 1 2 3 1))
1
```

Решение 3.48.2

```
(defun our-count (a w &optional (p 0))
  (cond ((null w) p)
        ((equal a (car w)) (our-count a (cdr w) (1+ p)))
        ((our-count a (cdr w) p))))

(defun max-occur (w &optional (n 0) z &aux (m (our-count (car w) w)))
  (cond ((null w) z)
        ((> m n) (max-occur (cdr w) m (car w)))
        ((max-occur (cdr w) n z))))
```

```
> (max-occur '(1 1 1 1 2 3 1))
1
```

Решение 3.48.3

```
(defun our-count (a w)
  (cond ((null w) 0)
        ((equal a (car w)) (1+ (our-count a (cdr w))))
        ((our-count a (cdr w)))))

(defun max-occur (w &optional (n 0) z)
  (cond ((null w) z)
        ((> (our-count (car w) w) n)
         (max-occur (cdr w) (f (car w) w) (car w)))
        ((max-occur (cdr w) n z))))
```

```
> (max-occur '(1 1 1 1 2 3 1))
1
```

Решение 3.48.4 * без использования счетчика

```
(defun drop-one (w &optional acc ac)
  (cond ((null w) acc)
        ((member (car w) ac) (drop-one (cdr w) (cons (car w) acc) ac))
        ((drop-one (cdr w) acc (cons (car w) ac)))))
```

```
(defun max-occur (w)
  (if (cdr w) (max-occur (drop-one w)) (car w)))

> (max-occur '(1 1 1 1 2 3 3 1))
1
```

Задача 3.49 caar-cdaaar.lisp

Используя только `car` и `cdr` вытащить `aaa` и следующего списка: `(((((ddd (aaa)) eee) nnn) kkk))`.

Решение 3.49.1

```
> (car (car (cdr (car (car (car '((((ddd (aaa)) eee) nnn) kkk)))))))
AAA
```

Решение 3.49.2

```
> (caar (cdaaar '((((ddd (aaa)) eee) nnn) kkk)))
AAA
```

Задача 3.50 caddd.lisp

Используя только `car` и `cdr` вытащить `aaa` и следующего списка: `(((((ddd (aaa)) eee) nnn) kkk))`.

Решение 3.50.1

```
> (car (cdr (cdr (cdr '(hhh iii ddd aaa))))))
AAA
```

Решение 3.50.2

```
> (cadddr '(hhh iii ddd aaa))
AAA
```

Задача 3.51 count-till*.lisp

Определить функцию, которая вычисляет количество элементов списка, предшествующих звездочке `(*)`.

Решение 3.51.1

```
(defun count-till* (w)
  (cond ((null w) 0)
        ((equal (car w) '*) 0)
        ((+ 1 (count-till* (cdr w)))))

> (count-till* '(1 2 3 4 * 6 7))
4
```

Задача 3.52 winner.lisp

Известны результаты бега спортсменов на длинную дистанцию. Время представлено в минутах и секундах. Определить фамилию победителя соревнования.

Решение 3.52.1

```
(defstruct run name mm ss)

(defun res (n)
  (when (> n 0) (cons (make-run :name (read) :mm (read) :ss (read))
    (res (1- n)))))

(defun name-ss (w)
  (mapcar #'(lambda (a) (list (run-name a)
    (+ (* (run-mm a) 60) (run-ss a))))
    w))

(defun r-sort (w)
  (sort (name-ss w) #'< :key #'cadr))

(defun winner (w)
  (caar (r-sort w)))

> (setq 3km (res 2))
ivanov
10
54
petrov
9
37
(#S(RUN :NAME IVANOV :MM 10 :SS 54) #S(RUN :NAME PETROV :MM 9 :SS 37))
> (winner 3km)
PETROV
```

Задача 3.53 advantage.lisp

Известны результаты бега спортсменов на длинную дистанцию. Время представлено в минутах и секундах. Определить разницу между лучшим и худшим результатом.

Решение 3.53.1

```
(defstruct run name mm ss)

(defun res (n)
  (when (> n 0) (cons (make-run :name (read) :mm (read) :ss (read))
    (res (1- n)))))

(defun name-ss (w)
  (mapcar #'(lambda (a) (list (run-name a)
```

```

                                (+ (* (run-mm a) 60) (run-ss a))))
      w))

(defun r-sort (w)
  (sort (name-ss w) #'< :key #'cadr))

(defun advantage (w &aux (v (r-sort w)))
  (- (cadar (last v)) (cadar v)))

> (setq 3km (res 2))
ivanov
10
54
petrov
9
37
(#S(RUN :NAME IVANOV :MM 10 :SS 54) #S(RUN :NAME PETROV :MM 9 :SS 37))
> (advantage 3km)
77

```

Задача 3.54 *clear-average.lisp*

Используя рекурсивный подход исключить из списка все вхождения элемента nil и вычислить среднее геометрическое элементов списка.

Решение 3.54.1

```

(defun clear-average (w &optional ac (n 0))
  (cond ((null w) (/ (apply #'* ac) n))
        ((null (car w)) (clear-average (cdr w) ac n))
        ((clear-average (cdr w) (cons (car w) ac) (1+ n)))))

> (setf w '(nil 46 45 44 41 nil 44 45 46 43 44 47 nil 46 45 46 46 44
42 41 49 nil))
(NIL 46 45 44 41 NIL 44 45 46 43 44 47 NIL 46 45 46 46 44 42 41 49
NIL)
> (clear-average w)
27324001339715137104165888000
> (setf w '(nil 46 45 44 41 nil 44 45 46 43 44 47 nil 46 45 46 nil))
(NIL 46 45 44 41 NIL 44 45 46 43 44 47 NIL 46 45 46 NIL)
> (clear-average w)
2879897744844185472000/13

```

Задача 3.55 *lead-atom.lisp*

Определить Функцию, которая вернет первый атом в списке, а если такого в списке нет, вернет nil.

Решение 3.55.1

```

(defun lead-atom (w)

```



```

(cond ((null w) nil)
      ((atom (car w)) (car w))
      ((lead-atom (cdr w)))))

> (lead-atom '((a) ((b) c) d))
D
> (lead-atom '((a) ((b) c)))
NIL

```

Решение 3.55.2

```

(defun lead-atom (w)
  (loop for a in w when (atom a) return a))

> (lead-atom '((a) ((b) c) d))
D
> (lead-atom '((a) ((b) c)))
NIL

```

Решение 3.55.3

```

(defun lead-atom (w) (find-if #'atom w))

> (lead-atom '((a) ((b) c)))
NIL
> (lead-atom '((a) ((b) c) d))
D
> (lead-atom '((a) ((b) c) d e))
D

```

Задача 3.56 *product-even.lisp*

Дан список целых чисел. Составить программу, вычисляющую произведение всех четных элементов списка за исключением нуля.

Решение 3.56.1

```

(defun product-even (w &aux (a (car w)))
  (cond ((null w) 1)
        ((or (oddp a) (zerop a)) (product-even (cdr w)))
        ((* a (product-even (cdr w))))))

> (product-even '(1 2 3 4 5 6 7))
48

```

Задача 3.57 *our-length.lisp*

Определить рекурсивную функцию, возвращающую количество элементов в списке.

Решение 3.57.1

```
(defun our-length (w) (if (null w) 0 (1+ (our-length (cdr w)))))

> (our-length '(a b c d))
4
> (our-length '())
0
```

Задача 3.58 *sumx.lisp*

Найти сумму чисел одноуровневого списка.

Решение 3.58.1

```
(defun sumx (w)
  (cond ((null w) 0)
        ((numberp (car w)) (+ (car w) (sumx (cdr w))))
        ((sumx (cdr w)))))

> (sumx '(54 gh 5 fd 7 9 12 -2 f))
85
```

Решение 3.58.2

```
(defun sumx (w) (loop for a in w when (numberp a) sum a))

> (sumx '(54 gh 5 fd 7 9 12 -2 f))
85
```

Задача 3.59 *mult.lisp*

Найти произведение чисел одноуровневого списка.

Решение 3.59.1

```
(defun mult (w)
  (cond ((null w) 1)
        ((numberp (car w)) (* (car w) (mult (cdr w))))
        ((mult (cdr w)))))

> (mult '(54 gh 5 fd 7 9 12 -2 f))
-408240
```

Решение 3.59.2

```
(defun mult (w)
  (loop for a in w
        when (numberp a) collect a into v
        finally return (reduce #'* v)))

> (mult '(2 gh 2 fd 2 2 2 -2 f))
```

-64

Задача 3.60 longest.lisp

В списке однобуквенных строк S1, S2, ..., ST найти длину самого длинного слова, если разделителем между словами является один или несколько пробелов.

Решение 3.60.1

```
(defun longest (w)
  (apply #'max
    (mapcar #'length
      (mapcar #'string
        (read-from-string
          (concatenate 'string
            "("
            (apply #'concatenate 'string w)
            ")")))))
    ))

> (longest '("t" "e" "s" "t" " " "s" "t" "r" "o" "k" "a" " " "s"))
6
```

Задача 3.61 prod-digits.lisp

Найти произведение цифр в символах одноуровневого списка.

Решение 3.61.1

```
(defun prod-digits (w &aux (a (car w)) (d (cdr w)))
  (cond ((null w) 1)
        ((symbolp a) (* (apply
                          #'* (mapcar
                                #'digit-char-p
                                (remove-if-not
                                  #'digit-char-p
                                  (coerce (string a) 'list))))
                          (prod-digits d)))
        ((* (abs a) (prod-digits d)))))

> (prod-digits '(a s ss d 1 -2 h3-4d))
24
```

Решение 3.61.1

```
(defun prod-digits (w &aux (a (car w)) (d (cdr w)))
  (cond ((null w) 1)
        ((numberp a) (* (abs a) (prod-digits d)))
        ((* (apply #'* (mapcar #'digit-char-p
                                (remove-if-not
                                  #'digit-char-p
```

```

                                (coerce (string a) 'list)))
      (prod-digits d))))))
> (prod-digits '(a s ss d 1 -2 h3-4d))
24

```

Задача 3.62 *sum-digits.lisp*

Найти сумму цифр в символах одноуровневого списка.

Решение 3.62.1

```

(defun sum-digits (w &aux (a (car w)) (d (cdr w)))
  (cond ((null w) 0)
        ((numberp a) (+ (abs a) (sum-digits d)))
        ((+ (apply #'(lambda (x) (remove-if-not
                                #'digit-char-p
                                (coerce (string x) 'list))))
             (sum-digits d)))))
> (sum-digits '(1a3 e34d w12 r3r1))
18

```

Решение 3.62.2

```

(defun sum-digits (w)
  (loop for a across
        (remove-if-not
         #'digit-char-p
         (format nil "~{~a~}"
                  (mapcar 'string w)))
        sum (digit-char-p a)))
> (sum-digits '(1a3 e34d w12 r3r1))
18

```

Решение 3.62.3

```

(defun sum-digits (w)
  (if w (+ (add
            (dig
             (dro
              (coerce (string (car w)) 'list))))
            (sum-digits (cdr w)))
    0))

(defun dig (w)
  (when w (cons (digit-char-p (car w)) (dig (cdr w)))))

(defun add (w) (if w (+ (car w) (add (cdr w))) 0))

```

```
(defun dro (w &aux (a (car w)))
  (cond ((null w) nil)
        ((digit-char-p a) (cons a (dro (cdr w))))
        ((dro (cdr w)))))

> (sum-digits '(1a3 e34d w12 r3r1))
18
```

Задача 3.63 *magic-square.lisp*

Написать программу, которая проверяет, является ли квадратная матрица "магическим" квадратом. "Магическим" квадратом называется матрица, у которой сумма чисел в каждом горизонтальном ряду, в каждом вертикальном и по каждой из диагоналей одна и та же. Пример матрицы 3x3:

```
2 9 4
7 5 3
6 1 8
```

Решение 3.63.1

```
(defun transpose (m) (apply #'mapcar #'list m))

(defun sum-main (w &optional (n 0))
  (if (null w) 0 (+ (nth n (car w)) (sum-main (cdr w) (1+ n)))))

(defun magic-sq (m)
  (apply #'=
    (sum-main m)
    (sum-main (reverse m))
    (mapcar #'(lambda (a) (reduce #'+ a))
      (nconc m (transpose m)))))

> (magic-sq '((2 9 4) (7 5 3) (6 1 8)))
T
> (magic-sq '((2 9 4) (7 5 3) (1 6 8)))
NIL
```

Задача 3.64 *count-not-numbers.lisp*

Посчитать количество нечисловых элементов в списке.

Решение 3.64.1

```
(defun count-not-numbers (w)
  (cond ((null w) 0)
        ((numberp (car w)) (count-not-numbers (cdr w)))
        ((1+ (count-not-numbers (cdr w)))))

> (count-not-numbers '(a a 1))
2
```

Решение 3.64.2

```
(defun count-not-numbers (w)
  (loop for a in w counting (not (numberp a))))

> (count-not-numbers '(a a 1))
2
```

Решение 3.64.3

```
(defun count-not-numbers (w) (count-if-not #'numberp w))

> (count-not-numbers '(a a 1))
2
```

Задача 3.65 *double-zero.lisp*

Lisp has, among other things: symbols, macros, minimalist syntax (data and code, both being s-expressions, look the same), functions as first class objects, map and reduce, manifest typing, generic functions, conses, eval. The above, incomplete, laundry list are not just ad hoc features. They are all powerful, organically related concepts about programs and problem decomposition. So, learning them teaches a way of decomposing problems in a naturally extensible way.

Eric Moss

Определить наличие в числовом списке двух подряд идущих нулевых элементов.

Решение 3.65.1

```
(defun double-zero (w)
  (cond ((null (cdr w)) nil)
        ((= (car w) (cadr w) 0) t)
        ((double-zero (cdr w)))))

> (double-zero '(0 0))
T
> (double-zero '(0 1))
NIL
```

Решение 3.65.2

```
(defun double-zero (w)
  (mapcan #'(lambda (a b)
              (when (= a b 0) (list 'ok)))
```

```

      w (cdr w)))

> (double-zero '(0 1))
NIL
> (double-zero '(0 0))
(OK)

```

Решение 3.65.3

```

(defun double-zero (w)
  (if (loop for a on w never
      (when (cdr a) (= (car a) (cadr w) 0))) nil t))

> (double-zero '(0 0))
T
> (double-zero '(0 1))
NIL

```

Решение 3.65.4

```

(defun double-zero (w)
  (loop for a on w thereis (when (cdr a) (= (car a) (cadr w) 0))))

> (double-zero '(0 1))
NIL
> (double-zero '(0 0))
T

```

Решение 3.65.5

```

(defun double-zero (w)
  (loop for a in w for b in (cdr w) thereis (= a b 0)))

> (double-zero '(0 0))
T
> (double-zero '(0 1))
NIL

```

Задача 3.66 *max-min.lisp*

Дан список. Если максимальный элемент предшествует минимальному, то посчитать их сумму, если нет то просто вывести 0.

Решение 3.66.1

```

(defun max-min (w &aux (x (apply #'max w)) (n (apply #'min w)))
  (if (< (position x w) (position n w)) (+ x n) 0))

> (max-min '(9 7 6 5 4 3 2 1))
10
> (max-min '(7 6 5 4 3 2 1 9))
0

```

Задача 3.67 expt-7-8.lisp

Дан числовой список. Возвести седьмой элемента списка в степень, равную восьмому элементу.

Решение 3.67.1

```
(defun expt-7-8 (w)
  (labels ((_expt (n m)
              (if (= m 1) n (* n (_expt n (1- m))))))
    (_expt (nth 6 w) (nth 7 w))))

> (expt-7-8 '(1 2 3 4 5 6 7 8 9))
5764801
> (expt 7 8)
5764801
```

Задача 3.68 max-depth-1.lisp

Написать функцию, определяющую максимальную глубину вложенности под-списков многоуровневого списка, считая с верхнего - первого уровня.

Решение 3.68.1

```
(defun max-depth-1 (w)
  (cond ((atom w) 0)
        ((max (+ 1 (max-depth-1 (car w))) (max-depth-1 (cdr w))))))

> (max-depth-1 '(a b (c d (e f) g)))
3
```

Решение 3.68.2

```
(defun max-depth-1 (w)
  (if (atom w)
      0
      (max (1+ (max-depth-1 (car w)))
            (max-depth-1 (cdr w)))))

> (max-depth-1 '(a b (c d (e f) g)))
3
```

Задача 3.69 average-wage.lisp

Дан список, содержащий сведения о зарплате рабочих завода. Каждая запись содержит поля - фамилия работника, наименование цеха, размер зарплаты за месяц. Вычислить среднемесячный заработок рабочего этого завода.

Решение 3.69.1


```
(defun average-wage (w) (float (/ (sum-wage w) (num-workers w))))

(defun sum-wage (w) (if (null w) 0 (+ (caddr w) (sum-wage (cdr w)))))

(defun num-workers (w) (if (null w) 0 (1+ (num-workers (cdr w)))))

> (average-wage '((ivanov 2 15000) (petrov 2 12000) (sidorov 2
16000)))
14333.333
```

Решение 3.69.2

```
(defun average-wage (w)
  (float (/ (loop for a in w sum (caddr a)) (length w))))

> (average-wage '((ivanov 2 15000) (petrov 2 12000) (sidorov 2
16000)))
14333.333
```

Задача 3.70 min-plusp.lisp

I'd written a recursive descent parser for C, so I decided to write one in Lisp. It was such a kick that I wrote the rest of the compiler in Lisp as well. Not exactly what Lisp was envisioned for, but it was a really pleasant experience.

Stan Hanks

Написать программу поиска минимального положительного элемента.

Решение 3.70.1

```
(defun min-plusp (w)
  (cond ((null w) nil)
        ((plusp (car w)) (mp (cdr w) (car w)))
        ((min-plusp (cdr w)))))

(defun mp (w n)
  (cond ((null w) n)
        ((and (plusp (car w)) (< (car w) n)) (mp (cdr w) (car w)))
        ((mp (cdr w) n))))

> (min-plusp '(-1 0 1 2))
1
```

Решение 3.70.2

```
(defun min-plusp (w) (reduce #'min (remove-if-not #'plusp w)))

> (min-plusp '(-1 0 1 2))
```

1

Задача 3.71 max-line.lisp

Определить номер строки матрицы, сумма элементов которой наибольшая.

Решение 3.71.1

```
(defun max-line (w &aux (v (mapcar #'(lambda (a) (apply #' + a)) w)))
  (position (reduce #'max v) v))
```

```
> (max-line '((1 2 3) (1 2 4) (1 2 3)))
```

1

Задача 3.72 sum-eventh.lisp

Дан числовой список. Определить функцию, вычисляющую сумму чисел, стоящих на четных местах.

Решение 3.72.1

```
(defun sum-eventh (w)
  (if (cdr w) (+ (cadr w) (sum-eventh (cddr w))) 0))
```

```
> (sum-eventh '(0 1 0 1 1))
```

2

Задача 3.73 compare-first-last.lisp

В списке, состоящем из подсписков типа ((a b) (s h) (h a)) проверить, совпадают ли первый и последний символы.

Решение 3.73.1

```
(defun deep-last (w &aux (a (car w)))
  (cond ((null (cdr w)) (if (atom a) a (deep-last a)))
        ((deep-last (cdr w)))))
```

```
(defun compare-first-last (w) (eq (caar w) (deep-last w)))
```

```
> (compare-first-last '((1 2 3) (4 5 6) (7 8 9)))
```

NIL

```
> (compare-first-last '((1 2 3) (4 5 6) (7 8 1)))
```

T

Решение 3.73.2

```
(defun deep-last (w &aux (a (car w)))
  (if (cdr w) (deep-last (cdr w)) (if (atom a) a (deep-last a))))
```

```
(defun compare-first-last (w) (eq (caar w) (deep-last w)))
```

```
> (compare-first-last '((1 2 3) (4 5 6) (7 8 1)))
T
> (compare-first-last '((1 2 3) (4 5 6) (7 8 9)))
NIL
```

Задача 3.74 non-number.lisp

Написать функцию, возвращающую первый нечисловой элемент заданного списка.

Решение 3.74.1

```
(defun non-number (w) (loop for a in w unless (numberp a) return a))

> (non-number '(1 2 a b 3))
A
```

Задача 3.75 exp.lisp

Написать функцию, проверяющую корректность произвольного выражения в форме списка - например, $a^b \cdot 5 + a \cdot x$.

Решение 3.75.1

```
(defun divide (w &aux (n (truncate (/ (length w) 2))))
  (mapcar #'list (subseq w 0 (1+ n)) (reverse (subseq w (1- n)))))

(defun collect (w) (when w (append (car w) (collect (cddr w)))))

(defun check-s (w &aux (a (car w)))
  (cond ((null w) t)
        ((or (eq a '+) (eq a '*)) nil)
        ((check-s (cdr w)))))

(defun check-m (w &aux (a (car w)))
  (cond ((null w) t)
        ((or (eq a '+) (eq a '*)) (check-m (cdr w)))
        (t nil)))

(defun exp (w &aux (v (divide w)))
  (and (check-s (collect v)) (check-m (collect (cdr v)))))

> (exp '(a * b2 * 5 + a * x))
T
```

Решение 3.75.2

```
(defun exp (w &optional f &aux (a (car w)))
  (cond ((null w) f)
        ((and f (or (eq a '+) (eq a '*))) (exp (cdr w) nil)))
```

```
((expp (cdr w) t))))
```

```
> (expp '(a * b2 * 5 + a * x))
T
```

Решение 3.75.3

```
(defun expp (w &optional f)
  (cond ((null w) f)
        ((and f (or (eq (car w) '+) (eq (car w) '*)))
         (expp (cdr w) nil))
        ((expp (cdr w) t))))
```

```
> (expp '(a * b2 * 5 + a * x))
T
```

Решение 3.75.4

```
(defun expp (w &optional f &aux (a (car w)))
  (if w (expp (cdr w) (not (and f (or (eq a '+) (eq a '*))))) f))
```

```
> (expp '(a * b2 * 5 + a * x))
T
```

Задача 3.76 min-number.lisp

Дан список, элементами которого являются списки, состоящие из чисел. Определить номер элемента списка, содержащего наименьшее число. Например, в случае '((1 3) (0) (6 1)), ответ: n = 2. Не рассматривать случай, когда такой элемент не единственный.

Решение 3.76.1

```
(defun min-number (w &aux (v (mapcar
                               #'(lambda (a)
                                   (reduce #'min a)) w)))
  (position (reduce #'min v) v))
```

```
> (min-number '((1 1) (1 0) (1 1)))
1
```

Задача 3.77 sum-elms.lisp

Определить функцию, которая считает сумму элементов многоуровневого списка.

Решение 3.77.1

```
(defun sum-elms (w)
  (cond ((null w) 0)
        ((atom w) w)))
```

```

      ((+ (sum-elms (car w)) (sum-elms (cdr w)))))
> (sum-elms '(1 ((1) (1)) (1) 1))
5

```

Задача 3.78 *deep-product.lisp*

Определить функцию, которая считает произведение элементов многоуровневого списка.

Решение 3.78.1

```

(defun deep-product (w)
  (cond ((null w) 1)
        ((atom (car w)) (* (car w) (deep-product (cdr w))))
        (t (* (deep-product (car w)) (deep-product (cdr w)))))
> (deep-product '(1 2 3 (3 6) 2 1 (2 4) 2))
3456

```

Решение 3.78.2 (автор – helter, www.cyberforum.ru)

```

(defun deep-product (w)
  (if (listp w)
      (reduce #'* w :key #'deep-product)
      w))
> (deep-product '(1 2 3 (3 6) 2 1 (2 4) 2))
3456

```

Решение 3.78.3

```

(defun deep-product (w)
  (if (atom w) w (reduce #'* w :key #'deep-product)))
> (deep-product '(1 2 3 (3 6) 2 1 (2 4) 2))
3456

```

Решение 3.78.4

```

(defun deep-product (w) (reduce #'* (deep-collect w)))
(defun deep-collect (w)
  (loop for a in w
        if (and a (atom a)) collect a
        else nconc (deep-collect a)))
> (deep-product '(1 2 3 (3 6) 2 1 (2 4) 2))
3456

```

Решение 3.78.5

```
(defun deep-product (w)
  (reduce #'* (labels ((deep-collect (v)
                        (loop for a in v
                              if (and a (atom a)) collect a
                              else nconc (deep-collect a))))
    (deep-collect w))))

> (deep-product '(1 2 3 (3 6) 2 1 (2 4) 2))
3456
```

Решение 3.78.6

```
(defun deep-product (w &aux (p 1))
  (loop for a in w do
    (if (atom a) (setf p (* p a))
        (setf p (* p (reduce #'* a)))))
  p)

> (deep-product '(1 2 3 (3 6) 2 1 (2 4) 2))
3456
```

Решение 3.78.7

```
(defun deep-product (w &aux (p 1))
  (loop for a in w do
    (setf p (* p (if (atom a) a (reduce #'* a)))))
  p)

> (deep-product '(1 2 3 (3 6) 2 1 (2 4) 2))
3456
```

Решение 3.78.8

```
(defun deep-product (w &aux (p 1))
  (loop for a in w do
    (if (atom a) (setf p (* p a))
        (setf p (apply #'* p a))))
  p)

> (deep-product '(1 2 3 (3 6) 2 1 (2 4) 2))
3456
```

Решение 3.78.9

```
(defun deep-product (w)
  (loop with p = 1 for a in w do
    (if (atom a) (setf p (* p a))
        (setf p (apply #'* p a)))
    finally (return p)))

> (deep-product '(1 2 3 (3 6) 2 1 (2 4) 2))
3456
```

Решение 3.78.10

```
(defun deep-product (w)
  (loop with p = 1 for a in w do
    (setf p (* p (if (atom a) a (reduce #'* a))))
    finally (return p)))

> (deep-product '(1 2 3 (3 6) 2 1 (2 4) 2))
3456
```

Задача 3.79 sum-10+.lisp

Определить функцию, которая считает сумму чисел больше 10 в одноуровневом числовом списке.

Решение 3.79.1

```
(defun sum-10+ (w) (loop for a in w when (> a 10) sum a))

> (sum-10+ '(8 9 10 11 12))
23
```

Задача 3.80 sum-twins.lisp

Определить функцию, которая находит сумму элементов списка, встречающихся ровно дважды.

Решение 3.80.1

```
(defun sum-twins (w &optional (v w))
  (cond ((null w) 0)
        ((= (count (car w) v) 2) (+ (car w) (sum-twins (cdr w) v)))
        ((sum-twins (cdr w) v))))

> (sum-twins '(1 2 3 4 5 2 3))
10
```

Решение 3.80.2

```
(defun sum-twins (w &optional (v w))
  (if w
      (if (= (count (car w) v) 2)
          (+ (car w) (sum-twins (cdr w) v))
          (sum-twins (cdr w) v))
      0))

> (sum-twins '(1 2 3 4 5 2 3))
10
```

Решение 3.80.3

```
(defun sum-twins (w &optional (v w))
  (if w
    (+ (if (= (count (car w) v) 2)
          (car w)
          0)
      (sum-twins (cdr w) v))
    0))
```

```
> (sum-twins '(1 2 3 4 5 2 3))
10
```

Решение 3.80.4

```
(defun sum-twins (w &aux (v w))
  (loop for a in w when (= (count a v) 2) sum a))
```

```
> (sum-twins '(1 2 3 4 5 2 3))
10
```

Задача 3.81 *count-atoms.lisp*

Learning Lisp isn't a 21-day course in boilerplate features and syntax memorization. It means learning about organically related ideas in problem decomposition whose minimalist representations happen to actually evaluate.

Eric Moss

Дан многоуровневый список. Определить функцию, возвращающую число атомов на всех уровнях списка.

Решение 3.81.1

```
(defun count-atoms (w)
  (cond ((null w) 0)
        ((atom (car w)) (1+ (count-atoms (cdr w))))
        (t (+ (count-atoms (car w)) (count-atoms (cdr w))))))
```

```
> (count-atoms '((a (b) ((c)) d) e))
5
```

Решение 3.81.2

```
(defun count-atoms (w)
  (cond ((null w) 0)
        ((atom w) 1)
        ((+ (count-atoms (car w)) (count-atoms (cdr w))))))
```

```
> (count-atoms '((a (b) ((c)) d) e))
5
```


Решение 3.81.3

```
(defun count-atoms (w)
  (if w
      (if (atom w)
          1
          (+ (count-atoms (car w)) (count-atoms (cdr w))))
      0))

> (count-atoms '((a (b) ((c)) d) e))
5
```

Решение 3.81.4 (автор - korvin, www.cyberforum.ru)

```
(defun count-atoms (w)
  (if (atom w) 1 (reduce #'+ w :key #'count-atoms)))

> (count-atoms '((a (b) ((c)) d) e))
5
```

Задача 3.82 glue-numbers.lisp

Определить функцию, преобразующую список (123 34 1234 45) в число 12334123445.

Решение 3.82.1 (автор - helter, www.cyberforum.ru)

```
(defun num-dgt (n)
  (do ((n n (floor n 10))
      (k 0 (1+ k)))
      ((< n 10) (1+ k))))

(defun glue-numbers (w)
  (reduce #'(lambda (a n)
              (+ (* a (expt 10 (num-dgt n)))
                 n))
          w
          :initial-value 0))

> (glue-numbers '(123 34 1234 45))
12334123445
> (glue-numbers #(123 34 1234 45))
12334123445
> (glue-numbers #(123 0 1234 45))
1230123445
> (glue-numbers #(1234 45 0))
1234450
> (glue-numbers #(0 1234 45))
123445
```

Решение 3.82.2

```

(defun digits (n) (if (< n 10) 1 (1+ (digits (floor n 10)))))

(defun glue (w &optional (n 0) &aux (a (car w)))
  (if w (+ (* a (expt 10 n)) (glue (cdr w) (+ n (digits a))))) 0))

(defun glue-numbers (w) (glue (reverse w)))

> (glue-numbers '(123 34 1234 45))
12334123445
> (glue-numbers '(123 0 1234 45))
1230123445
> (glue-numbers '(0 1234 45))
123445
> (glue-numbers '(10 0))
100

```

Решение 3.82.3

```

(defun glue-numbers (w)
  (parse-integer (apply #'concatenate
                        'string (mapcar #'write-to-string w))))

> (glue-numbers '(123 34 1234 45))
12334123445
11
> (glue-numbers '(0 123 34 0 1234 45 0))
1233401234450
14

```

Задача 3.83 *prns.lisp*

Определить функцию, которая считает произведение элементов многоуровневого списка, при этом проверять - если элемент не число, то пропустить его.

Решение 3.83.1

```

(defun prns (w &aux (a (car w)) (d (cdr w)))
  (cond ((null w) 1)
        ((atom a) (if (numberp a)
                       (* a (prns d))
                       (prns d)))
        ((* (prns a) (prns d)))))

> (prns '(1 2 a 3 (3 b 6) 2 1 (2 4 c) 2 d))
3456

```

Решение 3.83.2

```

(defun prns (w &aux (a (car w)) (d (cdr w)))
  (if w (* (if (atom a) (if (numberp a) a 1) (prns a)) (prns d) 1))

```

```
> (prns '(1 2 a 3 (3 b 6) 2 1 (2 4 c) 2 d))
3456
```

Решение 3.83.3

```
(defun prns (w)
  (if (atom w) (if (numberp w) w 1) (reduce #'* w :key #'prns)))

> (prns '(1 2 a 3 (3 b 6) 2 1 (2 4 c) 2 d))
3456
```

Задача 3.84 sum-ns.lisp

Дан список положительных целых чисел. Составить программу, вычисляющую сумму всех чисел, начинающихся на 1.

Решение 3.84.1

```
(defun sum-ns (w)
  (loop for a in w
        when (equal (subseq (write-to-string a) 0 1) "1")
        sum a))

> (sum-ns '(12 22 13))
25
```

Решение 3.84.2

```
(defun sum-ns (w)
  (loop for a in w when (= (ns a) 1) sum a))

(defun ns (n)
  (if (< n 10) n (ns (truncate (/ n 10)))))

> (sum-ns '(12 22 13))
25
```

Решение 3.84.3

```
(defun sum-ns (w)
  (cond ((null w) 0)
        ((= (ns (car w)) 1) (+ (car w) (sum-ns (cdr w))))
        ((sum-ns (cdr w)))))

(defun ns (n)
  (if (< n 10) n (ns (truncate (/ n 10)))))

> (sum-ns '(12 22 13))
25
```

Решение 3.84.4

```
(defun sum-ns (w)
  (cond ((null w) 0)
        ((> (ns (car w)) 1) (sum-ns (cdr w)))
        ((+ (car w) (sum-ns (cdr w))))))
```

```
(defun ns (n)
  (if (< n 10) n (ns (truncate (/ n 10)))))
```

```
> (sum-ns '(12 22 13))
25
```

Решение 3.84.5

```
(defun sum-ns (w)
  (apply #'+ (remove-if #'(lambda (a) (> (ns a) 1)) w)))
```

```
(defun ns (n)
  (if (< n 10) n (ns (truncate (/ n 10)))))
```

```
> (sum-ns '(12 22 13))
25
```

Задача 3.85 escalade.lisp

Для числового списка определить количество отрезков, на которых элементы строго возрастают (отрезок возрастания имеет минимальную длину 2).

Решение 3.85.1

```
(defun escalade (w &optional (n 0) f)
  (cond ((null (cdr w)) n)
        ((< (car w) (cadr w)) (escalade (cdr w) (if f n (1+ n)) t))
        ((escalade (cdr w) n nil))))
```

```
> (escalade '(1 2 3 2 3 4 1))
2
> (escalade '(1 2 3 2 3 4 1 1 7 7))
3
> (escalade '(1 2 3 2 3 4 1 1 1 7))
3
```

Решение 3.85.2

```
(defun escalade (w &optional (n 0) f)
  (if (cdr w)
      (if (< (car w) (cadr w))
          (escalade (cdr w) (if f n (1+ n)) t)
          (escalade (cdr w) n nil))
      n))
```

```

> (escalade '(1 2 3 2 3 4 1))
2
> (escalade '(1 2 3 2 3 4 1 1 7 7))
3
> (escalade '(1 2 3 2 3 4 1 1 1 7))
3

```

Задача 3.86 *sum-int-10+.lisp*

Дан числовой список. Найти сумму целых чисел больше 10.

Решение 3.86.1

```

(defun sum-int-10+ (w)
  (cond ((null w) 0)
        ((and (integerp (car w)) (> (car w) 10))
         (+ (car w) (sum-int-10+ (cdr w))))
        ((sum-int-10+ (cdr w)))))

> (sum-int-10+ '(20 1 1 1 1 1 1 1 20 20.5))
40

```

Решение 3.86.2

```

(defun sum-int-10+ (w)
  (loop for a in w when (and (integerp a) (> a 10)) sum a))

> (sum-int-10+ '(20 1 1 1 1 1 1 1 20 20.5))
40

```

Задача 3.87 *list-inside.lisp*

Составить функцию, которая определяет, есть ли список внутри списка.

Решение 3.87.1

```

(defun list-inside (w) (some #'listp w))

> (list-inside '(1 2 3 (4 5) 6))
T

```

Решение 3.87.2

```

(defun list-inside (w)
  (cond ((null w) nil)
        ((atom (car w)) (list-inside (cdr w)))
        (t t)))

> (list-inside '(1 2 3 (4 5) 6))
T

```

Решение 3.87.3

```
(defun list-inside (w)
  (when w (if (atom (car w)) (list-inside (cdr w)) t)))

> (list-inside '(1 2 3 (4 5) 6))
T
```

Решение 3.87.4

```
(defun list-inside (w)
  (when w (or (listp (car w)) (list-inside (cdr w)))))

> (list-inside '(1 2 3 (4 5) 6))
T
```

Задача 3.88 $1+z+z^2/2+z^3/3.lisp$

Определить функцию, считающую сумму ряда $1 + z + z^2/2 + z^3/3 + z^4/4$.

Решение 3.88.1

```
(defun 1+z+z^2/2+z^3/3 (z n)
  (1+ (loop for n from 1 to n sum (/ (expt z n) n))))

> (1+z+z^2/2+z^3/3 -0.475 4)
0.61481524
```

Задача 3.89 *over-n.lisp*

Определить функцию, которая возвращает количество атомов-чисел из одноуровневого числового списка w больше, чем n .

Решение 3.89.1

```
(defun over-n (n w)
  (cond ((null w) 0)
        ((> (car w) n) (1+ (over-n n (cdr w)))))
  (t (over-n n (cdr w)))))

> (over-n 3 '(1 2 3 4 5 6 7))
4
```

Решение 3.89.2

```
(defun over-n (n w)
  (if w
      (if (> (car w) n)
          (1+ (over-n n (cdr w)))
          (over-n n (cdr w)))
      0))
```

```
0))
```

```
> (over-n 3 '(1 2 3 4 5 6 7))
4
```

Решение 3.89.3

```
(defun over-n (n w)
  (loop for a in w
        when (> a n) count a))
```

```
> (over-n 3 '(1 2 3 4 5 6 7))
4
```

Решение 3.89.4

```
(defun over-n (n w) (count-if #'(lambda (a) (> a n)) w))
```

```
> (over-n 3 '(1 2 3 4 5 6 7))
4
```

Задача 3.90 8th.lisp

Дан список, состоящий не менее чем из восьми элементов. Определить функцию, которая возвращает восьмой элемент списка.

Решение 3.90.1

```
(defun 8th (w) (car (cdr (cdr (cdr (cdr (cdr (cdr (cdr (cdr w))))))))))
```

```
> (8th '(1 2 3 4 5 6 7 8 9))
8
```

Решение 3.90.2

```
(defun 8th (w)
  (car (cdr (cdr (cdr (cdr (cdr (cdr (cdr (cdr w))))))))))
```

```
> (8th '(1 2 3 4 5 6 7 8 9))
8
```

Решение 3.90.3

```
(defun 8th (w) (nth 7 w))
```

```
> (8th '(1 2 3 4 5 6 7 8 9))
8
```

Решение 3.90.4

```
(defun 8th (w) (elt w 7))
```

```
> (8th '(1 2 3 4 5 6 7 8 9))
8
```

Решение 3.90.5

```
(defun 8th (w) (eighth w))

> (8th '(1 2 3 4 5 6 7 8 9))
8
```

Задача 3.91 count-aas.lisp

Определить функцию, которая возвращает количество уровней сложного списка на которых есть два не числовых атома.

Решение 3.91.1

```
(defun count-aas (w)
  (if w
      (+
        (if (> (count-if
              #'(lambda (a) (and (atom a) (not (numberp a)))) w) 1)
          1
          0)
        (count-if #'(lambda (a) (and (listp a) (count-aas a))) w)
        0))
    0))

> (count-aas '((a (2 b c 6) 7) d e ((9 f g) 12 (h 14) 15)))
3
```

Решение 3.91.2

```
(defun count-aas (w)
  (cond ((null w) 0)
        (t (+ (if (> (count-if
                    #'(lambda (a) (and (atom a)
                                       (not (numberp a)))) w) 1)
              1
              0)
            (count-if #'(lambda (a) (and (listp a)
                                         (count-aas a))) w))))))

> (count-aas '((a (2 b c 6) 7) d e ((9 f g) 12 (h 14) 15)))
3
```

Решение 3.91.3

```
(defun count-aas (w)
  (cond ((null w) 0)
        (t (+ (if (> (count-if
                    #'(lambda (a) (and (atom a)
                                       (not (numberp a)))) w) 1)
              1
              0)
            (count-if #'(lambda (a) (and (listp a)
                                         (count-aas a))) w))))))

> (count-aas '((a (2 b c 6) 7) d e ((9 f g) 12 (h 14) 15)))
3
```



```

      1
      0)
      (count-if #'(lambda (a) (and (listp a)
                                   (count-aas a)))) w))))))
> (count-aas '((a (2 b c 6) 7) d e ((9 f g) 12 (h 14) 15)))
3

Решение 3.91.4

(defun count-aas (w)
  (cond (w (+ (if (> (count-if #'(lambda (a) (and (atom a)
                                                    (not (numberp a)))) w) 1)
              1
              0)
        (count-if #'(lambda (a) (and (listp a)
                                      (count-aas a))) w)))
  (0)))

> (count-aas '((a (2 b c 6) 7) d e ((9 f g) 12 (h 14) 15)))
3

```

Задача 3.92 *lastt.lisp*

Определить функцию, которая возвращает предпоследний элемент списка.

Решение 3.92.1

```

(defun llast (w) (car (last w 2)))

> (llast '(a b c))
В

```

Задача 3.93 *unique.lisp*

Имеется список, например (1 2 3 4 5 2 4 1). Нужно чтобы функция проверяла имеются ли повторяющиеся символы и писала в ответе или nil или t.

Решение 3.93.1

```

(defun unique (w)
  (cond ((null w) t)
        ((member (car w) (cdr w)) nil)
        ((unique (cdr w)))))

> (unique '(1 2 3))
Т
> (unique '(1 2 2))
NIL

```

1. Если (null w), то достигли конца списка, вернули t - истина;

2. Если `(member (car w) (cdr w))`, то есть `(car w)` - голова списка входит в `(cdr w)` - хвост того же списка, то вернули `nil` - ложь, элемент встречается более одного раза;

3. Если не выполнилось ни одно из условий, то возвращаем `(unique (cdr w))` - применяем функцию к хвосту списка (`t` - истина, в третьей строке - условие, которое всегда выполняется, если не выполнены условия в первой или во второй строке).

Решение 3.93.2

```
(defun unique (w)
  (or (null w) (and (not (member (car w) (cdr w))) (unique (cdr w)))))

> (unique '(1 2 3))
T
> (unique '(1 2 2))
NIL
```

Решение 3.93.3

```
(defun unique (w)
  (or (null w) (when (not (member (car w) (cdr w)))
                    (unique (cdr w)))))

> (unique '(1 2 3))
T
> (unique '(1 2 2))
NIL
```

Решение 3.93.4

```
(defun unique (w)
  (or (null w) (unless (member (car w) (cdr w)) (unique (cdr w)))))

> (unique '(1 2 3))
T
> (unique '(1 2 2))
NIL
```

Решение 3.93.5

```
(defun unique (w)
  (cond ((null w)
        ((member (car w) (cdr w)) nil)
        ((unique (cdr w)))))

> (unique '(1 2 3))
T
> (unique '(1 2 2))
NIL
```

Решение 3.93.6

```
(defun unique (w)
  (if w (unless (member (car w) (cdr w)) (unique (cdr w))) t))

> (unique '(1 2 3))
T
> (unique '(1 2 2))
NIL
```

Решение 3.93.7

```
(defun unique (w) (every #'(lambda (a) (= (count a w) 1)) w))

> (unique '(1 2 3))
T
> (unique '(1 2 2))
NIL
```

Решение 3.93.8

```
(defun unique (w &optional acc ac (a (car w)) (d (cdr w)))
  (cond ((null w) (nreverse acc))
        ((or (member a ac) (member a d))
         (unique d (cons nil acc) (cons a ac)))
        ((unique d (cons t acc) (cons a ac)))))

> (unique '(1 2 3 4 5 2 4 1))
(NIL NIL T NIL T NIL NIL NIL)
```

Решение 3.93.9

```
(defun unique (w &optional acc)
  (maplist #'(lambda (d) (let ((a (car d)))
                           (progl (and (not (member a acc))
                                         (not (member a (cdr d))))
                                     (push a acc))))
           w))

> (unique '(1 2 3 4 5 2 4 1))
(NIL NIL T NIL T NIL NIL NIL)
```

Решение 3.93.10 * А может в качестве ответа быть так: (unique '(1 2 3 4 5 2 4 1)) -> NIL NIL T NIL T T T T

```
(defun unique (w)
  (when w (cond ((member (car w) (cdr w)) (cons nil (unique (cdr w))))
                ((cons t (unique (cdr w)))))))

> (unique '(1 2 3 4 5 2 4 1))
(NIL NIL T NIL T T T T)
```

Решение 3.93.11

```
(defun unique (w)
  (when w (cond ((member (car w) (cdr w))
                 (cons nil (unique (cdr w))))
                ((cons t (unique (cdr w)))))))

> (unique '(1 2 3 4 5 2 4 1))
(NIL NIL T NIL T T T T)
```

Задача 3.94 count-min.lisp

Lisp encourages the thing that improves code the most — ruthless refactoring — by getting out of the way. It gets out of the way by being based on constructs that naturally capture basic, powerful concepts of problem decomposition, and by having an extensible representations of data and code. That is, less code and brain cycles go toward supporting the needs of the language and more toward thinking about the problem itself. It may sound like mysticism on some level, but experience with it is pretty convincing.

Eric Moss

Определите функцию, возвращающую количество элементов, равных минимальному элементу числового списка.

Решение 3.94.1

```
(defun count-min (w &aux (n (reduce #'min w)))
  (count n w))

> (count-min '(1 2 3 1))
2
```

Задача 3.95 count-odd.lisp

Определите функцию, возвращающую количество нечетных чисел в списке.

Решение 3.95.1

```
(defun count-odd (w)
  (cond ((null w) 0)
        ((oddp (car w)) (1+ (count-odd (cdr w))))
        ((count-odd (cdr w)))))

> (count-odd '(1 2 3 4 5 6 7))
4
```

Решение 3.95.1

```
(defun count-odd (w) (count-if #'oddp w))

> (count-odd '(1 2 3 4 5 6 7))
4
```

Задача 3.96 (*lambda (w) (if...)*.lisp

Известно, что один из четырех элементов списка (a1 a2 a3 a4) отличен от других. Получить номер этого элемента. Составить нерекурсивную программу в форме лямбда-вызова, используя функцию if.

Решение 3.96.1

```
> ((lambda (w) (if (and (equal (car w) (cadr w))
                        (equal (cadr w) (caddr w)))
                    4
    (if (and (equal (car w) (cadr w))
            (equal (cadr w) (caddr w)))
        3
    (if (and (equal (car w) (caddr w))
            (equal (caddr w) (caddr w)))
        2
    1)))) '(1 0 0 0))
1
> ((lambda (w) (if (and (equal (car w) (cadr w))
                        (equal (cadr w) (caddr w)))
                    4
    (if (and (equal (car w) (cadr w))
            (equal (cadr w) (caddr w)))
        3
    (if (and (equal (car w) (caddr w))
            (equal (caddr w) (caddr w)))
        2
    1)))) '(0 1 0 0))
2
> ((lambda (w) (if (and (equal (car w) (cadr w))
                        (equal (cadr w) (caddr w)))
                    4
    (if (and (equal (car w) (cadr w))
            (equal (cadr w) (caddr w)))
        3
    (if (and (equal (car w) (caddr w))
            (equal (caddr w) (caddr w)))
        2
    1)))) '(0 0 1 0))
3
> ((lambda (w) (if (and (equal (car w) (cadr w))
                        (equal (cadr w) (caddr w)))
                    4
    (if (and (equal (car w) (cadr w))
            (equal (cadr w) (caddr w)))
        3
    (if (and (equal (car w) (caddr w))
            (equal (caddr w) (caddr w)))
        2
    1)))) '(0 0 1 0))
3
```

```

3
  (if (and (equal (car w) (caddr w))
           (equal (caddr w) (caddr w))))
2
1)))) '(0 0 0 1))
4

```

Задача 3.97 sum-oddp.lisp

Вычислить сумму нечетных элементов списка.

Решение 3.97.1

```

(defun sum-oddp (w)
  (cond ((null w) 0)
        ((oddp (car w)) (+ (car w) (sum-oddp (cdr w))))
        (t (sum-oddp (cdr w)))))

> (sum-oddp '(1 2 3 4 5))
9

```

Решение 3.97.2

```

(defun sum-oddp (w)
  (if w (+ (if (oddp (car w)) (car w) 0) (sum-oddp (cdr w)) 0)))

> (sum-oddp '(1 2 3 4 5))
9

```

Решение 3.97.3

```

(defun sum-oddp (w)
  (loop for a in w when (oddp a) sum a))

> (sum-oddp '(1 2 3 4 5))
9

```

Решение 3.97.4

```

(defun sum-oddp (w)
  (reduce #'+ (remove-if-not #'oddp w)))

> (sum-oddp '(1 2 3 4 5))
9

```

Решение 3.97.5

```

(defun sum-oddp (w)
  (loop for a in w sum (if (oddp a) a 0)))

> (sum-oddp '(1 2 3 4 5))
9

```

Решение 3.97.6

```
(defun sum-oddp (w)
  (reduce #'+ w :key #'(lambda (a) (if (oddp a) a 0))))

> (sum-oddp '(1 2 3 4 5))
9
```

Задача 3.98 *monotonic.lisp*

Определить предикат, который проверяет, является ли простой список чисел монотонной последовательностью.

Решение 3.98.1

```
(defun monotonic (w)
  (or (mono w #'<) (mono w #'>)))

(defun mono (w p)
  (cond ((null (cdr w))
        ((funcall p (car w) (cadr w)) nil)
        ((mono (cdr w) p))))

> (monotonic '(3 2 2 1))
T
> (monotonic '(1 2 2 3))
T
> (monotonic '(10 2 2 3))
NIL
```

Решение 3.98.2

```
(defun monotonic (w)
  (or (mono w #'<=) (mono w #'>=)))

(defun mono (w p)
  (if (cdr w)
      (and (funcall p (car w) (cadr w))
            (mono (cdr w) p))
      t))

> (monotonic '(3 2 2 1))
T
> (monotonic '(1 2 2 3))
T
> (monotonic '(10 2 2 3))
NIL
```

Решение 3.98.3

```
(defun monotonic (w)
```

```

(or (mono w #'<=) (mono w #'>=)))

(defun mono (w p)
  (loop for (a b) on w
        while b always (funcall p a b)))

> (monotonic '(3 2 2 1))
T
> (monotonic '(1 2 2 3))
T
> (monotonic '(10 2 2 3))
NIL

```

Решение 3.98.4

```

(defun monotonic (w)
  (or (apply #'>= w) (apply #'<= w)))

> (monotonic '(3 2 2 1))
T
> (monotonic '(1 2 2 3))
T
> (monotonic '(10 2 2 3))
NIL

```

Решение 3.98.5

```

(defun monotonic (w)
  (or (eval (cons '>= w)) (eval (cons '<= w)))))

> (monotonic '(3 2 2 1))
T
> (monotonic '(1 2 2 3))
T
> (monotonic '(10 2 2 3))
NIL

```

Задача 3.99 count-minus.lisp

Определить функцию, вычисляющую количество отрицательных элементов в списке.

Решение 3.99.1

```

(defun count-minus (w)
  (cond ((null w) 0)
        ((minusp (car w)) (1+ (count-minus (cdr w))))
        ((count-minus (cdr w)))))

> (count-minus '(-3 -2 -1 0))
3

```


Решение 3.99.2

```
(defun count-minus (w)
  (if w (+ (if (minusp (car w)) 1 0)
           (count-minus (cdr w))) 0))

> (count-minus '(-3 -2 -1 0))
3
```

Решение 3.99.3

```
(defun count-minus (w)
  (loop for a in w count (minusp a)))

> (count-minus '(-3 -2 -1 0))
3
```

Решение 3.99.4

```
(defun count-minus (w)
  (count-if #'minusp w))

> (count-minus '(-3 -2 -1 0))
3
```

Задача 3.100 check-<.lisp

Определить, является ли список упорядоченным по возрастанию. В случае отрицательного ответа, определить номер первого элемента, нарушившего такую упорядоченность.

Решение 3.100.1

```
(defun check-< (w &optional (n 1))
  (cond ((or (null w) (null (cdr w))))
        ((< (car w) (cadr w)) (check-< (cdr w) (1+ n)))
        (n)))

> (check-< '(1 2 3 4 5))
T
> (check-< '(1 2 3 1 5))
3
```

Задача 3.101 atom-n.lisp

Написать функцию, которая проверяет является ли аргумент списком из n атомов.

Решение 3.101.1

```
(defun atom-n (w n)
```

```

    (and (= (length w) n)
          (every #'atom w)))

> (atom-n '(1 2 3) 3)
T
> (atom-n '(1 2) 3)
NIL
> (atom-n '(1 2 nil) 3)
T
> (atom-n '(1 2 '(a)) 3)
NIL

```

Задача 3.102 count-all.lisp

Дан список целых чисел. Требуется дать описание фрагмента программы, в котором определяется количество нечетных элементов, количество отрицательных четных и количество прочих элементов списка.

Решение 3.102.1

```

(defun count-all (w)
  (values (count-if #'oddp w)
          (count-if #'(lambda (a) (and (minusp a) (evenp a))) w)
          (count-if-not #'(lambda (a)
                            (or (oddp a)
                                (and (minusp a) (evenp a))))
                w)))

> (count-all '(11 -63 51 -32 -6 44 -59 55 -67 -24 0 -85 29))
8
3
2

```

Решение 3.102.2

```

(defun count-all (w &aux
  (a (count-if #'oddp w))
  (b (count-if #'(lambda (a)
                    (and (minusp a)
                        (evenp a)))
                w)))
  (values a b (- (length w) a b)))

> (count-all '(11 -63 51 -32 -6 44 -59 55 -67 -24 0 -85 29))
8
3
2

```

Решение 3.102.3

```

(defun count-all (w)
  (loop for a in w

```

```

if (oddp a) count a into o
else
if (and (minusp a) (evenp a))
count a into me
else count a into r
finally (return (values o me r)))

```

```

> (count-all '(11 -63 51 -32 -6 44 -59 55 -67 -24 0 -85 29))
8
3
2

```

Решение 3.102.4

```

(defun count-all (w &optional (o 0) (me 0) (r 0)
  &aux (a (car w)) (d (cdr w)))
  (cond ((null w) `(), me, r))
  ((oddp a) (count-all d (1+ o) me r))
  ((and (minusp a) (evenp a)) (count-all d o (1+ me) r))
  ((count-all d o me (1+ r)))))

```

```

> (count-all '(11 -63 51 -32 -6 44 -59 55 -67 -24 0 -85 29))
(8 3 2)

```

Решение 3.102.5

```

(defun count-all (w)
  (loop for a in w
    if (oddp a)
    collect a into o
    else
    if (and (minusp a) (evenp a))
    collect a into me
    else collect a into r
    finally (return (values (length o)
                           (length me)
                           (length r)
                           o
                           me
                           r))))

```

```

> (count-all '(11 -63 51 -32 -6 44 -59 55 -67 -24 0 -85 29))
8
3
2
(11 -63 51 -59 55 -67 -85 29)
(-32 -6 -24)
(44 0)

```

Решение 3.102.6

```

(defun count-all (w &optional o me r &aux (a (car w)) (d (cdr w)))
  (cond ((null w) (values (length o) (length me) (length r) o me r))

```

```

((oddp a) (count-all d (cons a o) me r))
((and (minusp a) (evenp a)) (count-all d o (cons a me) r))
(t (count-all d o me (cons a r))))

> (count-all '(11 -63 51 -32 -6 44 -59 55 -67 -24 0 -85 29))
8
3
2
(29 -85 -67 55 -59 51 -63 11)
(-24 -6 -32)
(0 44)

```

Задача 3.103 anarchy-three.lisp

Определить функцию, вычисляющую количество троек подряд идущих элементов числового списка, в каждой из которых все элементы различны.

Решение 3.103.1

```

(defun anarchy-three (w)
  (cond ((null (cddr w)) 0)
        ((and (/= (car w) (cadr w))
              (/= (car w) (caddr w))
              (/= (cadr w) (caddr w)))
         (+ 1 (anarchy-three (cdr w))))
        ((anarchy-three (cdr w)))))

> (anarchy-three '(1 1 3 5 4 5 2 3 2 2 1 1 1))
4

```

Задача 3.104 palindrome-number.lisp

Дано натуральное число n . Проверить, является ли число палиндромом (перевёртышем).

Решение 3.104.1

```

(defun palindrome-number (n)
  (cond ((< n 10))
        ((= (take-left n) (take-right n))
         (palindrome-number (trim-left (trim-right n)))))

(defun take-right (n)
  (rem n 10))

(defun take-left (n)
  (if (< n 10) n (take-left (truncate n 10))))

(defun trim-left (n &optional (z n) (m 0))
  (if (< n 10)
      (rem z (expt 10 m))

```

```

(trim-left (truncate n 10) z (1+ m)))

(defun trim-right (n)
  (truncate n 10))

> (palindrome-number 6116)
T
> (palindrome-number 6117)
NIL

```

Решение 3.104.2 (автор - Catstail, www.cyberforum.ru)

```

(defun palindrome-number (n &optional (z n) (m 0))
  (if (zerop n)
      (= z m)
      (palindrome-number (truncate (/ n 10))
                          z
                          (+ (* m 10) (rem n 10)))))

> (palindrome-number 61516)
T
> (palindrome-number 61517)
NIL

```

Решение 3.104.3

```

(defun palindrome-number (w)
  (equal (write-to-string w) (reverse (write-to-string w))))

> (palindrome-number 6116)
T
> (palindrome-number 6117)
NIL

```

Решение 3.104.4

```

(defun palindrome-number (w &aux (v (write-to-string w)))
  (equal v (reverse v)))

> (palindrome-number 6116)
T
> (palindrome-number 6117)
NIL

```

Решение 3.104.5

```

(defun palindrome-number (w &aux (v (write-to-string w)))
  (string= v (reverse v)))

> (palindrome-number 6116)
T
> (palindrome-number 6117)
NIL

```

Задача 3.105 *verify-first-second-identity.lisp*

Дан список символов. Определить функцию, проверяющую тождественность первого и второго символов.

Решение 3.105.1

```
(defun verify-first-second-identity (w)
  (eq (car w) (cadr w)))

> (verify-first-second-identity '(a a b))
T
> (verify-first-second-identity '(a b b))
NIL
```

Задача 3.106 *zerop-car.lisp*

Определить функцию, проверяющую голову списка на равенство нулю.

Решение 3.106.1

```
(defun zerop-car (lst)
  (and (numberp (car lst)) (zerop (car lst))))

> (zerop-car '())
NIL
> (zerop-car '(a))
NIL
> (zerop-car '(1))
NIL
> (zerop-car '(0))
T
```

Задача 3.107 *deep-sum.lisp*

Определить функцию, которая считает сумму чисел многоуровневого списка.

Решение 3.107.1

```
(defun deep-sum (w)
  (cond ((null w) 0)
        ((atom (car w)) (+ (car w) (deep-sum (cdr w))))
        ((+ (deep-sum (car w)) (deep-sum (cdr w))))))

> (deep-sum '(1 2 3 (3 6) 2 1 (2 4) 2))
26
```

Решение 3.107.2

```
(defun deep-sum (w)
```

```
(if (atom w) w (reduce #'+ w :key #'deep-sum)))

> (deep-sum '(1 2 3 (3 6) 2 1 (2 4) 2))
26
```

Решение 3.107.3

```
(defun deep-sum (w)
  (loop for a in w
        if (atom a) sum a
        else sum (deep-sum a)))

> (deep-sum '(1 2 3 (3 6) 2 1 (2 4) 2))
26
```

Задача 3.108 chosen-sum.lisp

Определить функцию, возвращающую сумму числовых элементов списка, который может содержать также пустые подсписки.

Решение 3.108.1

```
(defun chosen-sum (w)
  (cond ((null w) 0)
        ((first w) (+ (first w) (chosen-sum (rest w))))
        ((chosen-sum (rest w)))))

> (chosen-sum '(1 2 nil 3 4 5))
15
```

Решение 3.108.2

```
(defun chosen-sum (w)
  (cond ((null w) 0)
        ((car w) (+ (car w) (chosen-sum (cdr w))))
        ((chosen-sum (cdr w)))))

> (chosen-sum '(1 2 nil 3 4 5))
15
```

Решение 3.108.3

```
(defun chosen-sum (w)
  (if (null w)
      0
      (if (car w)
          (+ (car w) (chosen-sum (cdr w)))
          (chosen-sum (cdr w)))))

> (chosen-sum '(1 2 nil 3 4 5))
15
```

Решение 3.108.4

```
(defun chosen-sum (w)
  (loop for a in w when a sum a))

> (chosen-sum '(1 2 nil 3 4 5))
15
```

Задача 3.109 *smallest-multiple.lisp*

2520 is the smallest number that can be divided by each of the numbers from 1 to 10 without any remainder. What is the smallest positive number that is evenly divisible by all of the numbers from 1 to 20?

2520 – самое маленькое число, которое делится без остатка на все числа от 1 до 10. Какое самое маленькое число делится нацело на все числа от 1 до 20?

Решение 3.109.1

```
(defun smallest-multiple (w &optional (n (car w)))
  (if (every #'(lambda (a) (zerop (rem n a))) w)
      n
      (smallest-multiple w (+ n (car w)))))

> (smallest-multiple '(20 19 18 17 16 15 14 13 12 11))
232792560
```

Решение 3.109.2

```
(defun smallest-multiple (w)
  (reduce #'lcm w))

> (smallest-multiple '(20 19 18 17 16 15 14 13 12 11))
232792560
```

Задача 3.110 *min-even.lisp*

Определить функцию, которая принимает целое значение и возвращает из заданного списка с подписками наименьшее из четных чисел, больших аргумента.

Решение 3.110.1

```
(defun min-even (a w)
  (loop for e in (flat w)
        when (and (evenp e) (< a e))
        minimize e))

(defun flat (w &optional acc)
  (cond ((null w) acc)
```



```
((atom w) (cons w acc))
((flat (car w) (flat (cdr w) acc)))))
```

```
> (min-even 2 '(1 ((2 (3 4)) 5) 6))
4
```

Решение 3.110.2

```
(defun min-even (a w)
  (loop for e in (flat w)
        when (and (evenp e) (< a e))
        minimize e))
```

```
(defun flat (w)
  (loop for a in w
        if (listp a) append (flat a)
        else collect a))
```

```
> (min-even 2 '(1 ((2 (3 4)) 5) 6))
4
```

Решение 3.110.3

```
(defun min-even (a w)
  (loop for e in (flat w)
        when (and (evenp e) (< a e))
        minimize e))
```

```
(defun flat (w)
  (mapcan #'(lambda (a) (if (listp a) (flat a) `(,a)))
          w))
```

```
> (min-even 2 '(1 ((2 (3 4)) 5) 6))
4
```

Решение 3.110.4

```
(defun min-even (a w)
  (loop for e in (flat w)
        when (and (evenp e) (< a e))
        minimize e))
```

```
(defun flat (w)
  (mapcan #'(lambda (a) (if (atom a) `(,a) (flat a)))
          w))
```

```
> (min-even 2 '(1 ((2 (3 4)) 5) 6))
4
```

Решение 3.110.5

```
(defun min-even (a w)
  (loop for e in (flat w)
```

```

      when (and (evenp e) (< a e))
      minimize e))

(defun flat (w)
  (if (atom w) `(,w) (mapcan #'flat w)))

> (min-even 2 '(1 ((2 (3 4)) 5) 6))
4

```

Решение 3.110.6

```

(defun even (a w &optional acc)
  (cond ((null w) acc)
        ((atom w) (if (and (evenp w) (< a w))
                        (cons w acc)
                        acc))
        ((even a
                 (car w)
                 (even a
                       (cdr w)
                       acc)))))

(defun min-even (a w)
  (reduce #'min (even a w)))

> (min-even 2 '(1 ((2 (3 4)) 5) 6))
4

```

Задача 3.111 *every-numberp.lisp*

Определить функцию, которая проверяет, все ли элементы списка являются числами.

Решение 3.111.1

```

(defun every-numberp (w)
  (cond ((null w) t)
        ((numberp (car w)) (every-numberp (cdr w)))))

> (every-numberp '(a 1))
NIL
> (every-numberp '(1 1))
T

```

Решение 3.111.2

```

(defun every-numberp (w)
  (cond ((null w) t)
        ((numberp (car w)) (every-numberp (cdr w)))))

> (every-numberp '(a 1))
NIL

```

```
> (every-numberp '(1 1))
T
```

Решение 3.111.3

```
(defun every-numberp (w)
  (if w (when (numberp (car w)) (every-numberp (cdr w))) t))

> (every-numberp '(a 1))
NIL
> (every-numberp '(1 1))
T
```

Решение 3.111.4

```
(defun every-numberp (w)
  (if w (and (numberp (car w)) (every-numberp (cdr w))) t))

> (every-numberp '(a 1))
NIL
> (every-numberp '(1 1))
T
```

Решение 3.111.5

```
(defun every-numberp (w)
  (or (null w) (and (numberp (car w)) (every-numberp (cdr w)))))

> (every-numberp '(a 1))
NIL
> (every-numberp '(1 1))
T
```

Решение 3.111.6

```
(defun every-numberp (w)
  (eval (cons 'and (mapcar #'numberp w))))

> (every-numberp '(a 1))
NIL
> (every-numberp '(1 1))
T
```

Решение 3.111.7

```
(defun every-numberp (w)
  (loop for a in w always (numberp a)))

> (every-numberp '(a 1))
NIL
> (every-numberp '(1 1))
T
```

Решение 3.111.8

```
(defun every-numberp (w) (every #'numberp w))

> (every-numberp '(a 1))
NIL
> (every-numberp '(1 1))
T
```

Задача 3.112 *most-expensive.lisp*

На аптечном складе хранятся лекарства. Сведения о лекарствах содержатся в специальной ведомости: наименование препарата, количество, цена, срок хранения (в месяцах). Выяснить, сколько стоит самый дорогой препарат.

Решение 3.112.1

```
((defstruct pharmacy name quantity price shelf-life)

(defun res (n)
  (when (> n 0)
    (cons (make-pharmacy :name (read)
                        :quantity (read)
                        :price (read)
                        :shelf-life (read))
          (res (1- n)))))

(defun most-expensive (w)
  (loop for a in w
        maximize (pharmacy-price a)))

> (setf w (res 2))
aspirin
2
10
6
analgin
3
20
1
(#S(PHARMACY :NAME ASPIRIN :QUANTITY 2 :PRICE 10 :SHELF-LIFE 6)
 #S(PHARMACY :NAME ANALGIN :QUANTITY 3 :PRICE 20 :SHELF-LIFE 1))
> (most-expensive w)
20
```

Задача 3.113 *cheapest.lisp*

На аптечном складе хранятся лекарства. Сведения о лекарствах содержатся в специальной ведомости: наименование препарата, количество, цена, срок хранения (в месяцах). Выяснить, сколько стоит самый дешевый пре-

парат.

Решение 3.113.1

```
((defstruct pharmacy name quantity price shelf-life)

(defun res (n)
  (when (> n 0)
    (cons (make-pharmacy :name (read)
                        :quantity (read)
                        :price (read)
                        :shelf-life (read))
          (res (1- n)))))

(defun cheapest (w)
  (loop for a in w
        minimize (pharmacy-price a)))

> (setf w (res 2))
aspirin
2
10
6
analgin
3
20
1
(#S(PHARMACY :NAME ASPIRIN :QUANTITY 2 :PRICE 10 :SHELF-LIFE 6)
 #S(PHARMACY :NAME ANALGIN :QUANTITY 3 :PRICE 20 :SHELF-LIFE 1))
> (cheapest w)
10
```

Задача 3.114 stock-amount.lisp

На аптечном складе хранятся лекарства. Сведения о лекарствах содержатся в специальной ведомости: наименование препарата, количество, цена, срок хранения (в месяцах). Выяснить, сколько препаратов хранится на складе.

Решение 3.114.1

```
((defstruct pharmacy name quantity price shelf-life)

(defun res (n)
  (when (> n 0)
    (cons (make-pharmacy :name (read)
                        :quantity (read)
                        :price (read)
                        :shelf-life (read))
          (res (1- n)))))

(defun stock-amount (w)
```

```

(loop for a in w count a))

> (setf w (res 2))
aspirin
2
10
6
analgin
3
20
1
(#S(PHARMACY :NAME ASPIRIN :QUANTITY 2 :PRICE 10 :SHELF-LIFE 6)
 #S(PHARMACY :NAME ANALGIN :QUANTITY 3 :PRICE 20 :SHELF-LIFE 1))
> (stock-amount w)
2

```

Задача 3.115 stock-cost.lisp

На аптечном складе хранятся лекарства. Сведения о лекарствах содержатся в специальной ведомости: наименование препарата, количество, цена, срок хранения (в месяцах). Выяснить, сколько стоят все препараты, хранящиеся на складе.

Решение 3.115.1

```

((defstruct pharmacy name quantity price shelf-life)

(defun res (n)
  (when (> n 0)
    (cons (make-pharmacy :name (read)
                        :quantity (read)
                        :price (read)
                        :shelf-life (read))
          (res (1- n)))))

(defun stock-cost (w)
  (loop for a in w
        sum (pharmacy-price a)))

> (setf w (res 2))
aspirin
2
10
6
analgin
3
20
1
(#S(PHARMACY :NAME ASPIRIN :QUANTITY 2 :PRICE 10 :SHELF-LIFE 6)
 #S(PHARMACY :NAME ANALGIN :QUANTITY 3 :PRICE 20 :SHELF-LIFE 1))
> (stock-cost w)
30

```

Задача 3.116 *sum-first-last.lisp*

Определить функцию, возвращающую сумму первого и последнего атома в одноуровневом числовом списке.

Решение 3.116.1

```
(defun sum-first-last (w)
  (apply #'+ (car w) (last w)))

> (sum-first-last '(1 2 3 4))
5
```

Задача 3.117 *largest-number.lisp*

In any language, a better decomposition of the problem leads to more elegant code, but Lisp is especially rewarding of this, and many classes of problems have very concise, obvious code because of it. The part of this that helps one become better is that the easiest way to write Lisp is with rapid prototyping, and the syntax and typing in Lisp makes it easy to get something working, and makes obvious what improves the model and what makes it "clunkier".

Eric Moss

Given a list of non negative integers, arrange them such that they form the largest number. For example, given (3 30 34 5 9), the largest formed number is 9534330.

Дан список положительных целых чисел. Организовать их так, чтобы они образовали наибольшее число. Например, если дан список (3 30 34 5 9), то наибольшее число - 9534330.

Решение 3.117.1

```
(defun largest-number (w)
  (parse-integer
    (apply #'concatenate 'string
      (selection-sort (mapcar #'write-to-string w))))))

(defun selection-sort (w)
  (when w (let ((m (reduce #'mx w)))
    (cons m (selection-sort (remove m w :count 1))))))

(defun mx (a b
  &aux
    (ab (parse-integer (concatenate 'string a b)))
    (ba (parse-integer (concatenate 'string b a))))
```

```

      (if (> ab ba) a b))

> (largest-number '(3 30 34 5 9))
9534330
7

```

Решение 3.117.2

```

(defun largest-number (w)
  (parse-integer
    (apply #'concatenate 'string
      (selection-sort (mapcar #'write-to-string w))))))

(defun selection-sort (w)
  (when w (let ((m (reduce #'mx w)))
    (cons m (selection-sort (remove m w :count 1))))))

(defun mx (a b)
  (if (> (parse-integer (concatenate 'string a b))
    (parse-integer (concatenate 'string b a)))
    a
    b))

> (largest-number '(3 30 34 5 9))
9534330
7

```

Решение 3.117.3

```

(defun largest-number (w)
  (parse-integer
    (apply #'concatenate 'string
      (selection-sort (mapcar #'write-to-string w))))))

(defun selection-sort (w)
  (when w
    (let ((m (reduce
      #'(lambda (a b)
        (if (> (parse-integer (concatenate 'string a b))
          (parse-integer (concatenate 'string b a)))
          a
          b))
      w)))
      (cons m (selection-sort (remove m w :count 1))))))

> (largest-number '(3 30 34 5 9))
9534330
7

```

Решение 3.117.4

```

(defun largest-number (w)
  (parse-integer

```



```

    (apply #'concatenate 'string
      (selection-sort (mapcar #'write-to-string w))))

(defun selection-sort (w)
  (when w
    (let ((m (reduce #'(lambda (a b) (if (> (glue a b) (glue b a))
                                           a
                                           b))
                     w)))
      (cons m (selection-sort (remove m w :count 1))))))

(defun glue (a b)
  (parse-integer (concatenate 'string a b)))

> (largest-number '(3 30 34 5 9))
9534330
7

```

Решение 3.117.5

```

(defun largest-number (w)
  (parse-integer
    (format nil "~{~a~}"
      (selection-sort
        (mapcar #'write-to-string w)))))

(defun selection-sort (w)
  (when w (let ((m (reduce #'mx w)))
    (cons m (selection-sort
      (remove m w :count 1))))))

(defun mx (a b)
  (if (> (glue a b) (glue b a)) a b))

(defun glue (a b)
  (parse-integer (concatenate 'string a b)))

> (largest-number '(3 30 34 5 9))
9534330
7

```

Решение 3.117.6

```

(defun largest-number (w)
  (parse-integer
    (format nil "~{~a~}"
      (selection-sort
        (mapcar #'write-to-string w)))))

(defun selection-sort (w)
  (when w (let ((m (reduce #'mx w)))
    (cons m (selection-sort
      (remove m w :count 1))))))

```

```
(defun mx (a b)
  (if (> (glue a b) (glue b a)) a b))

(defun glue (a b)
  (parse-integer (format nil "~{~a~}" (list a b))))

> (largest-number '(3 30 34 5 9))
9534330
7
```

Задача 3.118 *maximum-gap.lisp*

Given an unsorted array, find the maximum difference between the successive elements in its sorted form. Return 0 if the array contains less than 2 elements. You may assume all elements in the array are non-negative integers and fit in the 32-bit signed integer range.

Дан список положительных целых чисел. Найти максимальную разницу между последовательными элементами в отсортированном списке. Вернуть 0, если массив содержит менее 2 элементов.

Решение 3.118.1

```
(defun maximum-gap (v &aux (w (sort v #'<)))
  (if (cdr w)
      (reduce #'max (loop for a in (butlast w)
                          for b in (cdr w)
                          collect (- b a)))
      0))

> (maximum-gap '(7 3 1))
4
> (maximum-gap '(3 1))
2
> (maximum-gap '(1 1))
0
> (maximum-gap '())
0
> (time (maximum-gap (loop repeat 100000 collect (random 1000000))))
Real time: 1.7346032 sec.
Run time: 1.7316111 sec.
Space: 2804232 Bytes
GC: 5, GC time: 0.0780005 sec.
103
```

Решение 3.118.2

```
(defun maximum-gap (v &aux (w (sort v #'<)))
  (if (cdr w)
      (reduce #'max
              (mapcar #'(lambda (a b) (- b a))
                      (loop for a in (butlast w)
                          for b in (cdr w)
                          collect a))))
      0))
```

```

                                (butlast w)
                                (cdr w)))
                                0))

> (maximum-gap '(7 3 1))
4
> (maximum-gap '(3 1))
2
> (maximum-gap '(1 1))
0
> (maximum-gap '())
0
> (time (maximum-gap (loop repeat 100000 collect (random 1000000))))
Real time: 1.6692029 sec.
Run time: 1.6692107 sec.
Space: 2804232 Bytes
GC: 5, GC time: 0.0624004 sec.
132

```

Задача 3.119 count-symbol.lisp

Определить функцию, возвращающую количество символов в списке.

Решение 3.119.1

```

(defun count-symbol (w)
  (cond ((null w) 0)
        ((symbolp (car w)) (1+ (count-symbol (cdr w))))
        (t (count-symbol (cdr w)))))

> (count-symbol '(2 s 3 f 7 f a v))
5

```

Решение 3.119.2

```

(defun count-symbol (w)
  (if w
      (+ (if (symbolp (car w)) 1 0)
         (count-symbol (cdr w)))
      0))

> (count-symbol '(2 s 3 f 7 f a v))
5

```

Решение 3.119.3

```

(defun count-symbol (w)
  (count-if #'symbolp w))

> (count-symbol '(2 s 3 f 7 f a v))
5

```

Задача 3.120 dominance.lisp

Определить доминирующий элемент списка (с частотой более 50%)

Решение 3.120.1

```
(defun dominance (w
                  &aux
                    (v (remove-duplicates w))
                    (m (loop for a in v maximizing (count a w))))
  (when (> m (* 0.5 (length w)))
    (find-if #'(lambda (a) (= (count a w) m)) v)))

> (dominance '(1 2 3 3 3 1 3 3 2))
3
> (dominance '(1 2 3 3 3 1 3 3 2 2 2 2))
NIL
> (dominance '(1 2 3 3 3 1 3 3 2 2 2 2 2 2))
2
```

Задача 3.121 last-elm.lisp

Определить рекурсивную функцию, возвращающую последний элемент списка.

Решение 3.121.1

```
(defun last-elm (w) (if (cdr w) (last-elm (cdr w)) (car w)))

> (last-elm '(1 2 3))
3
```

Задача 3.122 sum-numbers.lisp

Определить функцию, которая находила бы сумму всех числовых элементов в списке. Список может содержать подсписки произвольной глубины.

Решение 3.122.1

```
(defun sum-numbers (w)
  (reduce
    #' + w
    :key
    #'(lambda (a)
        (if (numberp a)
            a
            (if (listp a)
                (sum-numbers a)
                0)))))

> (sum-numbers '(1 2 ((3)) (4 a ((b)) 5 (((6 7)))) 8))
36
```

Решение 3.122.2

```
(defun sum-numbers (w)
  (loop for a in w
        if (numberp a) sum a
        else if (listp a) sum (sum-numbers a)))

> (sum-numbers '(1 2 (((3))) (4 a ((b)) 5 (((6 7)))) 8))
36
```

Задача 3.123 *sum-evenp.lisp*

Определить функцию, которая находила бы сумму всех четных чисел в списке. Список может содержать подсписки произвольной глубины.

Решение 3.122.1

```
(defun sum-evenp (w)
  (reduce
   #' + w
   :key
   #'(lambda (a)
       (if (and (numberp a) (evenp a))
           a
           (if (listp a)
               (sum-evenp a)
               0)))))

> (sum-evenp '(1 2 (((3))) (4 a ((b)) 5 (((6 7)))) 8))
20
```

Решение 3.123.2

```
(defun sum-evenp (w)
  (loop for a in w
        if (and (numberp a) (evenp a)) sum a
        else if (listp a) sum (sum-evenp a)))

> (sum-evenp '(1 2 (((3))) (4 a ((b)) 5 (((6 7)))) 8))
20
```

Задача 3.124 *reduce-gcd.lisp*

Дан список натуральных чисел. Определить функцию, которая находит их наибольший общий делитель.

Решение 3.124.1

```
(defun reduce-gcd (w) (reduce #'greatest-common-divisor w))
```

```
(defun greatest-common-divisor (a b)
  (if (zerop b) a (greatest-common-divisor b (mod a b))))

> (reduce-gcd '(9 27 81 1000))
1
> (reduce-gcd '(9 27 81 1008))
9
```

Решение 3.124.2

```
(defun reduce-gcd (w) (reduce #'gcd w))

> (reduce-gcd '(9 27 81 1000))
1
> (reduce-gcd '(9 27 81 1008))
9
```

Решение 3.124.3

```
(defun reduce-gcd (&rest w) (reduce #'gcd w))

> (reduce-gcd '(9 27 81 1000))
1
> (reduce-gcd '(9 27 81 1008))
9
```

Задача 3.125 min-number.lisp

Задан список произвольных чисел. Написать функцию нахождения минимального элемента.

Решение 3.125.1

```
(defun min-number (w &optional x &aux (a (car w)))
  (cond ((null w) x)
        ((or (not x) (> x a)) (min-number (cdr w) a))
        ((min-number (cdr w) x))))

> (min-number '(1 2))
1
```

Решение 3.125.2

```
(defun min-number (w &optional x &aux (a (car w)))
  (if w (min-number (cdr w) (if (or (not x) (> x a)) a x)) x))

> (min-number '(1 2))
1
```

Решение 3.125.3

```
(defun min-number (w) (reduce #'min w))
```

```
> (min-number '(1 2))
1
```

Задача 3.126 square-matrix-main-diagonal-below-sum.lisp

Дана квадратная матрица. Составить программу нахождения суммы элементов, лежащих ниже главной диагонали.

Решение 3.126.1

```
(defun square-matrix-main-diagonal-below-sum (w)
  (loop for a in (cdr w) for b upfrom 1
        sum (reduce #'+ (subseq a 0 b))))

> (square-matrix-main-diagonal-below-sum '((1 1 1 1) (1 1 1 1) (1 1 1
1) (1 1 1 1)))
6
```

Задача 3.127 square-matrix-main-diagonal-above-sum.lisp

Дана квадратная матрица. Составить программу нахождения суммы элементов, лежащих выше главной диагонали.

Решение 3.127.1

```
(defun square-matrix-main-diagonal-above-sum (w)
  (loop for a in w for b upfrom 1
        sum (reduce #'+ (subseq a b))))

> (square-matrix-main-diagonal-above-sum '((1 10 10 10) (1 1 10 10) (1
1 1 10) (1 1 1 1)))
60
```

Решение 3.127.2

```
(defun square-matrix-main-diagonal-above-sum (w)
  (loop for a in (butlast w) for b upfrom 1
        sum (reduce #'+ (subseq a b))))

> (square-matrix-main-diagonal-above-sum '((1 10 10 10) (1 1 10 10) (1
1 1 10) (1 1 1 1)))
60
```

Решение 3.127.3

```
(defun square-matrix-main-diagonal-above-sum (w &optional ac)
  (if w
      (square-matrix-main-diagonal-above-sum
       (mapcar #'cdr (cdr w))
       (nconc (cdar w) ac))
      0))
```

```

(reduce #' + ac)))

> (square-matrix-main-diagonal-above-sum '((1 10 10 10) (1 1 10 10) (1
1 1 10) (1 1 1 1)))
60

```

Решение 3.127.4

```

(defun square-matrix-main-diagonal-above-sum (w &optional (n 0))
  (if w
    (square-matrix-main-diagonal-above-sum
      (mapcar #'cdr (cdr w))
      (+ (reduce #' + (cdar w)) n))
    n))

> (square-matrix-main-diagonal-above-sum '((1 10 10 10) (1 1 10 10) (1
1 1 10) (1 1 1 1)))
60

```

Решение 3.127.5

```

(defun square-matrix-main-diagonal-above-sum (w &optional (n 0))
  (if w
    (square-matrix-main-diagonal-above-sum
      (mapcar #'cdr (cdr w))
      (reduce #' + (cons n (cdar w)))))
    n))

> (square-matrix-main-diagonal-above-sum '((1 10 10 10) (1 1 10 10) (1
1 1 10) (1 1 1 1)))
60

```

Решение 3.127.6

```

(defun square-matrix-main-diagonal-above-sum (w &optional (n 0))
  (if w
    (square-matrix-main-diagonal-above-sum
      (loop for a in (cdr w) collect (cdr a))
      (loop for a in (cons n (cdar w)) sum a))
    n))

> (square-matrix-main-diagonal-above-sum '((1 10 10 10) (1 1 10 10) (1
1 1 10) (1 1 1 1)))
60

```

Задача 3.128 *product.lisp*

Определить функцию, которая перемножает числа.

Решение 3.128.1

```

(defun product (w)

```



```
(cond ((null w) 1)
      (t (* (car w) (product (cdr w))))))

> (product '(10 15 23))
3450
```

Решение 3.128.2

```
(defun product (w)
  (if (null w) 1
      (* (car w) (product (cdr w)))))

> (product '(10 15 23))
3450
```

Решение 3.128.3

```
(defun product (w)
  (if w (* (car w) (product (cdr w))) 1))

> (product '(10 15 23))
3450
```

Решение 3.128.4

```
(defun product (w) (eval (cons '* w)))

> (product '(10 15 23))
3450
```

Решение 3.128.5

```
(defun product (w) (reduce #'* w))

> (product '(10 15 23))
3450
```

Задача 3.129 min-char.lisp

В списке из атомов-символов выбрать символ, код которого наименьший.

Решение 3.129.1

```
(defun min-char (w)
  (nth-value 0 (intern
    (string
      (code-char
        (car (sort (list-code (list-char w)) #'<)))))))

(defun list-char (w)
  (mapcar #'(lambda (a) (character a)) w))
```

```
(defun list-code (w)
  (mapcar #'(lambda (a) (char-code a)) w))
```

```
> (min-char '(b c a))
A
```

Решение 3.129.2

```
(defun min-char (w)
  (nth-value 0 (intern (car (sort (list-str w) #'string-lessp)))))
```

```
(defun list-str (w)
  (mapcar #'(lambda (a) (string a)) w))
```

```
> (min-char '(b c a))
A
```

Решение 3.129.3

```
(defun min-char (w)
  (nth-value 0 (intern (string (car (sort (list-char w) #'char<))))))
```

```
(defun list-char (w)
  (mapcar #'(lambda (a) (character a)) w))
```

```
> (min-char '(b c a))
A
```

Решение 3.129.4

```
(defun min-char (w)
  (car (sort w #'string<)))
```

```
(min-char '(b c a))
A
```

Задача 3.130 *some-duhplicates.lisp*

Вывести `t` если в списке есть повторяющиеся символы, иначе `nil`.

Решение 3.130.1

```
(defun some-duhplicates (w)
  (cond ((null w) nil)
        ((find (car w) (cdr w)) t)
        ((some-duhplicates (cdr w)))))
```

```
> (some-duhplicates '(a b c a))
T
> (some-duhplicates '(a b c d))
NIL
```

Решение 3.130.2

```
(defun some-duhplicates (w)
  (loop for a in w
        for d on (cdr w)
        when (find a d) return t))

> (some-duhplicates '(a b c a))
T
> (some-duhplicates '(a b c d))
NIL
```

Решение 3.130.2

```
(defun some-duhplicates (w)
  (not (equal w (remove-duplicates w))))

> (some-duhplicates '(a b c a))
T
> (some-duhplicates '(a b c d))
NIL
```

Задача 3.131 *average-w.lisp*

Дан числовой список. Найти среднее арифметическое.

Решение 3.131.1

```
(defun average-w (w &optional (s 0) (n 0))
  (cond ((null w) (unless (zerop n) (float (/ s n))))
        (t (average-w (cdr w) (+ s (car w)) (1+ n)))))

> (average-w '(1 2 3 4 5))
3.0
```

Решение 3.131.2

```
(defun average-w (w &optional (s 0) (n 0))
  (if w
      (average-w (cdr w) (+ s (car w)) (1+ n))
      (unless (zerop n) (float (/ s n)))))

> (average-w '(1 2 3 4 5))
3.0
```

Решение 3.131.3

```
(defun average-w (w)
  (when w (float (/ (reduce #'+ w) (length w)))))

> (average-w '(1 2 3 4 5))
3.0
```

Задача 3.132 sum-even.lisp

Дан числовой список. Определить функцию, которая возвращает сумму четных чисел.

Решение 3.132.1

```
(defun sum-even (w)
  (cond ((null w) 0)
        ((evenp (car w)) (+ (car w) (sum-even (cdr w))))
        ((sum-even (cdr w)))))
```

```
> (sum-even '(1 2 3 4 5 6))
12
```

Решение 3.132.2

```
(defun sum-even (w &aux (a (car w)))
  (cond ((null w) 0)
        ((evenp a) (+ a (sum-even (cdr w))))
        ((sum-even (cdr w)))))
```

```
> (sum-even '(1 2 3 4 5 6))
12
```

Решение 3.132.3

```
(defun sum-even (w)
  (loop for a in w when (evenp a) sum a))
```

```
> (sum-even '(1 2 3 4 5 6))
12
```

Решение 3.132.4

```
(defun sum-even (w &optional (ac 0) &aux (a (car w)))
  (cond ((null w) ac)
        ((evenp a) (sum-even (cdr w) (+ ac a)))
        ((sum-even (cdr w) ac))))
```

```
> (sum-even '(1 2 3 4 5 6))
12
```

Решение 3.132.5

```
(defun sum-even (w &optional (ac 0) &aux (a (car w)))
  (if w (sum-even (cdr w) (if (evenp a) (+ ac a) ac)) ac))
```

```
> (sum-even '(1 2 3 4 5 6))
12
```

Решение 3.132.6

```
(defun sum-even (w)
  (reduce #' + (remove-if-not #'evenp w)))

> (sum-even '(1 2 3 4 5 6))
12
```

Задача 3.133 count-n.lisp

Дан числовой список. Определить функцию, возвращающую количество чисел, которые делятся на n .

Решение 3.133.1

```
(defun count-n (w n)
  (cond ((null w) 0)
        ((zerop (mod (car w) n)) (1+ (count-n (cdr w) n)))
        ((count-n (cdr w) n))))

> (count-n '(1 2 3 4 5 6) 3)
2
```

Решение 3.133.2

```
(defun count-n (w n &optional (m 0))
  (cond ((null w) m)
        ((zerop (mod (car w) n)) (count-n (cdr w) n (1+ m)))
        ((count-n (cdr w) n m))))

> (count-n '(1 2 3 4 5 6) 3)
2
```

Решение 3.133.3

```
(defun count-n (w n &optional (m 0))
  (if w (count-n (cdr w) n (if (zerop (mod (car w) n)) (1+ m) m)) m))

> (count-n '(1 2 3 4 5 6) 3)
2
```

Решение 3.133.4

```
(defun count-n (w n)
  (loop for a in w when (zerop (mod a n)) count a))

> (count-n '(1 2 3 4 5 6) 3)
2
```

Решение 3.133.5

```
(defun count-n (w n)
  (count-if #'(lambda (a) (zerop (mod a n))) w))
```

```
> (count-n '(1 2 3 4 5 6) 3)
2
```

Задача 3.134 double-empty.lisp

Определить, есть ли в списке два подряд идущих пустых списка.

Решение 3.134.1

```
(defun double-empty (w)
  (when (cdr w) (cond ((and (null (car w)) (null (cadr w)))
                       ((double-empty (cdr w))))))

> (double-empty '((1) (1) ()))
NIL
> (double-empty '((1) () ()))
T
```

Решение 3.134.2

```
(defun double-empty (w)
  (when (cdr w)
    (if (or (car w) (cadr w)) (double-empty (cdr w)) t)))

> (double-empty '((1) (1) ()))
NIL
> (double-empty '((1) () ()))
T
```

Задача 3.135 flatp.lisp

Определить предикат, проверяющий, является ли аргумент одноуровневым списком.

Решение 3.135.1

```
(defun flatp (w)
  (every #'atom w))

> (flatp '(a a))
T
> (flatp '(a (a)))
NIL
```

Задача 3.136 sum.lisp

Входной список содержит только элементы nil и числа. Получить сумму элементов списка, не равных nil.

Решение 3.136.1

```
(defun sum (w)
  (if w
      (let ((a (car w)))
        (if (null a)
            (sum (cdr w))
            (+ a (sum (cdr w))))))
    0))
```

```
> (sum '(5 nil 5 nil))
10
```

Решение 3.136.2

```
(defun sum (w)
  (if w
      (let ((a (car w)))
        (if a
            (+ a (sum (cdr w)))
            (sum (cdr w))))
    0))
```

```
> (sum '(5 nil 5 nil))
10
```

Решение 3.136.3

```
(defun sum (w)
  (if w
      (let ((a (car w)))
        (+ (if a a 0) (sum (cdr w))))
    0))
```

```
> (sum '(5 nil 5 nil))
10
```

Решение 3.136.4

```
(defun sum (w &aux (a (car w)))
  (if w (+ (if a a 0)
          (sum (cdr w)) 0))
```

```
> (sum '(5 nil 5 nil))
10
```

Задача 3.137 *count-parities.lisp*

Определить функцию, которая по заданным двум числовым простым спискам одинаковой длины вычисляет количество элементов одинаковой чётности, стоящих на одних и тех же местах.

Решение 3.137.1

```
(defun count-parities (w v)
  (cond ((null w) 0)
        ((or (and (oddp (car w)) (oddp (car v)))
              (and (evenp (car w)) (evenp (car v))))
         (1+ (count-parities (cdr w) (cdr v)))
         (t (count-parities (cdr w) (cdr v)))))

> (count-parities '(1 2 3 4 5 6 7 8 9) '(1 5 6 4 6 3 9 1 5))
4
```

Решение 3.137.2

```
(defun count-parities (w v)
  (if w
      (if (or (and (oddp (car w)) (oddp (car v)))
              (and (evenp (car w)) (evenp (car v))))
          (1+ (count-parities (cdr w) (cdr v)))
          (count-parities (cdr w) (cdr v)))
      0))

> (count-parities '(1 2 3 4 5 6 7 8 9) '(1 5 6 4 6 3 9 1 5))
4
```

Решение 3.137.3

```
(defun count-parities (w v &aux (a (car w)) (b (car v)))
  (if w
      (if (or (and (oddp a) (oddp b))
              (and (evenp a) (evenp b)))
          (1+ (count-parities (cdr w) (cdr v)))
          (count-parities (cdr w) (cdr v)))
      0))

> (count-parities '(1 2 3 4 5 6 7 8 9) '(1 5 6 4 6 3 9 1 5))
4
```

Решение 3.137.4

```
(defun count-parities (w v &aux (a (car w)) (b (car v)))
  (if w
      (+ (if (or (and (oddp a) (oddp b))
                  (and (evenp a) (evenp b)))
          1
          0)
         (count-parities (cdr w) (cdr v)))
      0))

> (count-parities '(1 2 3 4 5 6 7 8 9) '(1 5 6 4 6 3 9 1 5))
4
```

Решение 3.137.5


```
(defun count-parities (w v)
  (count-if #'identity
    (mapcar #'(lambda (a b)
      (or (and (oddp a) (oddp b))
        (and (evenp a) (evenp b))))
      w v)))

> (count-parities '(1 2 3 4 5 6 7 8 9) '(1 5 6 4 6 3 9 1 5))
4
```

Решение 3.137.6

```
(defun count-parities (w v)
  (loop for a in w for b in v
    when (or (and (oddp a) (oddp b))
      (and (evenp a) (evenp b)))
    count a))

> (count-parities '(1 2 3 4 5 6 7 8 9) '(1 5 6 4 6 3 9 1 5))
4
```

Задача 3.138 max-zeros.lisp

Для заданного одноуровневого числового списка определить максимальное количество идущих подряд нулей.

Решение 3.138.1

```
(defun max-zeros (w &optional (n 0) (m 0) f)
  (cond ((null w) (max n m))
        ((zerop (car w)) (max-zeros (cdr w) (1+ n) m t))
        ((max-zeros (cdr w) 0 (if f (max n m) m) nil))))

> (max-zeros '(0 1 0 0 1 0 0 0 1 0 0 0 0))
4
> (max-zeros '(0 1 0 0 1 0 0 0 1))
3
> (max-zeros '(0 0 0 0 0 1 0 0 1 0 0 0 1))
5
> (max-zeros '(0))
1
> (max-zeros '())
0
```

Задача 3.139 max-zero-seq.lisp

Для заданного многоуровневого числового списка определить максимальное количество идущих подряд нулей.

Решение 3.139.1

```

(defun max-zero-seq (w &optional (n 0) (m 0) f)
  (cond ((null w) (max n m))
        ((listp (car w)) (max-zero-seq
                           (car w)
                           0
                           (max-zero-seq (cdr w) 0 m nil)
                           nil))
        ((zerop (car w)) (max-zero-seq (cdr w) (1+ n) m t))
        (f (max-zero-seq (cdr w) 0 (max n m) nil))
        ((max-zero-seq (cdr w) 0 m nil))))

> (max-zero-seq '(1 2 0 0 0 (2 0 0 0 0 0 3) 0 0 0 0))
6
> (max-zero-seq '(1 0 0 4 0 0 (1 0 0 0 0 3 (6 0 0 0) 1) 0 0 0 5))
4
> (max-zero-seq '(0))
1
> (max-zero-seq '())
0

```

Решение 3.139.2

```

(defun max-zero-seq (w &optional (n 0) (m 0) f)
  (cond ((null w) (max n m))
        ((listp (car w)) (max-zero-seq
                           (car w)
                           0
                           (max-zero-seq (cdr w) 0 m nil)
                           nil))
        ((zerop (car w)) (max-zero-seq (cdr w) (1+ n) m t))
        ((max-zero-seq (cdr w) 0 (if f (max n m) m) nil))))

> (max-zero-seq '(1 2 0 0 0 (2 0 0 0 0 0 3) 0 0 0 0))
6
> (max-zero-seq '(1 0 0 4 0 0 (1 0 0 0 0 3 (6 0 0 0) 1) 0 0 0 5))
4
> (max-zero-seq '(0))
1
> (max-zero-seq '())
0

```

Решение 3.139.3

```

(defun max-zero-seq (w
                     &optional (n 0) (m 0) f
                     &aux (a (car w)) (d (cdr w)))
  (cond ((null w) (max n m))
        ((listp a) (max-zero-seq a 0 (max-zero-seq d 0 m nil) nil))
        ((zerop a) (max-zero-seq d (1+ n) m t))
        ((max-zero-seq d 0 (if f (max n m) m) nil))))

> (max-zero-seq '(1 2 0 0 0 (2 0 0 0 0 0 3) 0 0 0 0))
6

```

```
> (max-zero-seq '(1 0 0 4 0 0 (1 0 0 0 0 3 (6 0 0 0) 1) 0 0 0 5))
4
> (max-zero-seq '(0))
1
> (max-zero-seq '())
0
```

Задача 3.140 count-zero.lisp

Таблица задана списком, каждый элемент которой представляет собой список чисел. Определить количество нулей в четных строках таблицы.

Решение 3.140.1

```
(defun count-zero (w)
  (cond ((null (cdr w)) 0)
        (t (+ (count 0 (cadr w)) (count-zero (cddr w))))))

> (count-zero '((1 1 1) (1 0 0)))
2
> (count-zero '((1 1 1) (1 0 0) (1 1 1) (1 0 0)))
4
```

Решение 3.140.2

```
(defun count-zero (w)
  (if (cdr w) (+ (count 0 (cadr w)) (count-zero (cddr w))) 0))

> (count-zero '((1 1 1) (1 0 0)))
2
> (count-zero '((1 1 1) (1 0 0) (1 1 1) (1 0 0)))
4
```

Решение 3.140.3

```
(defun count-zero (w)
  (loop for a in (cdr w) by #'cddr sum (count 0 a)))

> (count-zero '((1 1 1) (1 0 0)))
2
> (count-zero '((1 1 1) (1 0 0) (1 1 1) (1 0 0)))
4
```

Задача 3.141 last-element.lisp

Определить функцию, которая возвращает последний элемент списка.

Решение 3.141.1

```
(defun last-element (w)
  (if (cdr w) (last-element (cdr w)) (car w)))
```

```
> (last-element '(a b))
B
```

Задача 3.142 count-exclusive.lisp

Lisp requires an ability to think in terms of "recursive substitution", which when/if mastered definitely comes in handy, in any programming language.

Henry Norman

Определить функцию, которая возвращает количество различных элементов многоуровневого списка.

Решение 3.142.1

```
(defun count-exclusive (w) (length (exclusive w)))

(defun exclusive (w &optional ac)
  (cond ((null w) ac)
        ((atom w) (adjoin w ac))
        ((exclusive (car w) (exclusive (cdr w) ac)))))

> (count-exclusive '(1 2 3 (((4) (1 2 3)))))
4
```

Задача 3.143 sum-max-even-min-odd.lisp

Найти сумму максимального числа из четных по значению элементов и минимального числа из нечетных элементов списка.

Решение 3.143.1

```
(defun sum-max-even-min-odd (w)
  (when w (+ (reduce #'max (remove-if-not #'evenp w))
             (reduce #'min (remove-if-not #'oddp w)))))

> (sum-max-even-min-odd '(1 2 3 4 5 6 7))
7
> (sum-max-even-min-odd '())
NIL
```

Решение 3.143.2

```
(defun sum-max-even-min-odd (w)
  (when w (+ (f w #'max #'evenp) (f w #'min #'oddp))))

(defun f (w m p)
  (reduce m (remove-if-not p w)))
```

```
> (sum-max-even-min-odd '(1 2 3 4 5 6 7))
7
> (sum-max-even-min-odd '())
NIL
```

Задача 3.144 *sum-max-even-min-odd-order.lisp*

Найти сумму максимального числа из четных по порядку элементов и минимального числа из нечетных элементов списка.

Решение 3.144.1

```
(defun sum-max-even-min-odd-order (w)
  (when w (+ (reduce #'max (one (cdr w)))
             (reduce #'min (one w)))))

(defun one (w)
  (when w (cons (car w) (one (cddr w)))))

> (sum-max-even-min-odd-order '(1 2 3 4 5 6 7))
7
> (sum-max-even-min-odd-order '(10 2 3 4 5 60 7))
63
```

Задача 3.145 *consistent-content.lisp*

Modern man is conditioned to expect instant gratification but any success or triumph realized quickly, with only marginal effort is necessarily shallow. Meaningful achievement takes time, hard work, persistence, patience, proper intent and constant self-awareness. The path to such success is punctuated by failure, consolidation and renewed effort. Personal reconstruction is art.

Mark Twight

Определить функцию, проверяющую список и его подписки на то, что в них содержатся только числа, либо только символы.

Решение 3.145.1

```
(defun numbers (w)
  (loop for a in w always (numberp a)))

(defun symbols (w)
  (loop for a in w always (symbolp a)))

(defun number-symbol-consistent (w)
```

```

(or (numbers w) (symbols w)))

(defun consistent-content (w
                          &aux
                          (v (remove-if-not #'atom w))
                          (z (remove-if-not #'listp w)))
  (and (number-symbol-consistent v)
       (loop for a in z
             always (number-symbol-consistent a))))

> (consistent-content '(1 2 3 (4 5 6) (f g h)))
T
> (consistent-content '(1 2 3 (4 5 k) (f g h)))
NIL

```

Решение 3.145.2

```

(defun number-symbol-consistent (w)
  (or (every #'numberp w) (every #'symbolp w)))

(defun consistent-content (w)
  (and (number-symbol-consistent (remove-if-not #'atom w))
       (loop for a in (remove-if-not #'listp w)
             always (number-symbol-consistent a))))

> (consistent-content '(1 2 3 (4 5 6) (f g h)))
T
> (consistent-content '(1 2 3 (4 5 k) (f g h)))
NIL

```

Решение 3.145.3

```

(defun number-symbol-consistent (w)
  (or (every #'numberp w) (every #'symbolp w)))

(defun consistent-content (w)
  (and (number-symbol-consistent (remove-if-not #'atom w))
       (every #'consistent-content (remove-if-not #'listp w)))))

> (consistent-content '(1 2 3 (4 5 6) (f g h)))
T
> (consistent-content '(1 2 3 (4 5 k) (f g h)))
NIL

```

Решение 3.145.4

```

(defun number-symbol-consistent (w)
  (or (every #'numberp w) (every #'symbolp w)))

(defun consistent-content (w)
  (and (number-symbol-consistent (remove-if-not #'atom w))
       (every #'consistent-content (remove-if #'atom w)))))

```

```
> (consistent-content '(1 2 3 (4 5 6) (f g h)))
T
> (consistent-content '(1 2 3 (4 5 k) (f g h)))
NIL
```

Решение 3.145.5

```
(defun consistent-content (w &aux (v (remove-if #'listp w)))
  (and (or (every #'numberp v) (every #'symbolp v))
        (every #'consistent-content (remove-if #'atom w))))

> (consistent-content '(1 2 3 (4 5 6) (f g h)))
T
> (consistent-content '(1 2 3 (4 5 k) (f g h)))
NIL
```

Задача 3.146 *product-flat.lisp*

Определить функцию, возвращающую произведение всех чисел заданного сложного числового списка.

Решение 3.146.1

```
(defun product-flat (w &optional (ac 1))
  (cond ((null w) ac)
        ((atom w) (* w ac))
        ((product-flat (car w) (product-flat (cdr w) ac)))))

> (product-flat '(1 2 3 4 5 6 7))
5040
> (product-flat '(1 ((2) ((3 4)) 5)) (((6)) 7)))
5040
> (product-flat '(1 ((2) ((3 4)) 5)) (((6)) 0)))
0
```

Задача 3.147 *product-flat-sine-zero.lisp*

Определить функцию, возвращающую произведение всех ненулевых чисел заданного сложного числового списка.

Решение 3.147.1

```
(defun product-sine-zero (w &optional (ac 1))
  (cond ((null w) ac)
        ((atom w) (* (if (zerop w) 1 w) ac))
        ((product-sine-zero
          (car w)
          (product-sine-zero (cdr w) ac)))))

> (product-sine-zero '(0 1 ((2) ((3 4)) 5)) (((6 0)) 7)))
5040
```

Задача 3.148 *product-number-of-positive.lisp*

In C, Java, or even new-fangled languages like Haskell, the syntax very directly affects how you end up solving a problem. At least, that is what I have observed with most people. The tendency of many is to reduce the amount they type. In C, one might prefer to just use arrays as opposed to linked lists because linked lists requires having these extra functions and syntaxes, whereas arrays can use things like brackets to access elements and so forth. In Lisp, for better or for worse, most everything looks the same. A function call, an array access, some control flow... all of it looks like (this and this). As a result, in my experience, you're less prone to think about and implement a solution that otherwise might look ugly or inelegant in C or Haskell. Many people don't like this fact, and think it leads to verbosity. It is common in Lisp for code to be somewhat verbose; fully parenthesized syntax and spelled out names leads to that. By learning lisp, it's something you will probably come to appreciate, and will certainly affect the way you write programs in the future.

Robert Smith

Дан произвольный список. Определить функцию, возвращающую точечную пару, элементами которой являются произведение всех положительных чисел и количество положительных чисел.

Решение 3.148.1

```
(defun product-number-of-positive
  (w &aux (v (remove-if-not #'plusp (flat w))))
  (cons (reduce #'* v) (length v)))

(defun flat (w &optional acc)
  (cond ((null w) acc)
        ((atom w) (cons w acc))
        ((flat (car w) (flat (cdr w) acc)))))

> (product-number-of-positive '(2 -1 3 (3 -1)))
(18 . 3)
```

Решение 3.148.2

```
(defun product-number-of-positive
```



```

(w &aux (v (remove-if-not #'plusp (flat w))))
  (cons (reduce #'* v) (length v)))

(defun flat (w)
  (loop for a in w
        if (listp a) append (flat a)
        else collect a))

> (product-number-of-positive '(2 -1 3 (3 -1)))
(18 . 3)

```

Решение 3.148.3

```

(defun product-number-of-positive
  (w &aux (v (remove-if-not #'plusp (flat w))))
  (cons (reduce #'* v) (length v)))

(defun flat (w)
  (if (atom w) `(,w) (mapcan #'flat w)))

> (product-number-of-positive '(2 -1 3 (3 -1)))
(18 . 3)

```

Задача 3.149 average-of-positive.lisp

Дан список чисел. Определить функцию, возвращающую среднее арифметическое положительных чисел.

Решение 3.149.1

```

(defun average-of-positive (w
                           &optional (s 0) (n 0)
                           &aux (a (car w)))
  (cond ((null w) (/ s n))
        ((plusp a) (average-of-positive (cdr w) (+ s a) (1+ n)))
        ((average-of-positive (cdr w) s n))))

> (average-of-positive '(-10 100 1000))
550

```

Решение 3.149.2

```

(defun average-of-positive
  (w &aux (v (remove-if-not #'plusp w)))
  (/ (reduce #'+ v) (length v)))

> (average-of-positive '(-10 100 1000))
550

```

Решение 3.149.3

```

(defun average-of-positive (w)

```

```

(/ (loop for a in w when (plusp a) sum a)
   (loop for a in w when (plusp a) count a)))

> (average-of-positive '(-10 100 1000))
550

```

Задача 3.150 numbers.lisp

Within Emacs you can also edit text, in fact it has many text editors that run in Emacs, and it's perfectly possible that there are other text editors which sport some better features (at least better from some points of view). However those are text editors, not systems. They are not to be compared with Emacs, they are to be compared with an Emacs mode (which is a program, possibly a text editor, written to run in the Emacs system). They would be very very good if they run on a dynamic system like a Lisp or a Smalltalk or indeed Emacs (which is a Lisp system) but they do not: they run on static systems (funny how my iPhone just autocorrected that 'static' into 'septic'; for once I could actually agree with its autocorrect) like Windows and/or some Unix derivative.

Andrea Ferro

Проверить, является ли список списком чисел, или нет.

Решение 3.150.1

```

(defun numbersp (w)
  (loop for a in w always (or (numberp a)
                              (and (listp a)
                                   (numbersp a)))))

> (numbersp '(1 (2 (3 (((4)) 5)))))
T
> (numbersp '(1 (2 (3 (((4)) a)))))
NIL

```

Задача 3.151 sum-min-max.lisp

Дан список (a1, a2,..., aN). Вычислить значение выражения $\max(a2, a4, \dots) + \min(a1, a3, \dots)$.

Решение 3.151.1

```
(defun sum-min-max (w &optional f (a (car w)) (b (cadr w)))
  (cond ((null w) (+ a b))
        (f (sum-min-max (cdr w) nil a (max b (car w)))))
        ((sum-min-max (cdr w) t (min a (car w)) b))))

> (sum-min-max '(1 2 3 -4 5 6 7))
7
> (sum-min-max '(1 2 3 4 -5 6 7))
1
```

Задача 3.152 alternating-sum.lisp

Определите функцию (f w), вычисляющую знакопеременную сумму $a_1 - a_2 + a_3 - a_4 + \dots + a_k \cdot (-1)^k$ для списка w, имеющего вид (a1 a2 a3 ... ak).

Решение 3.152.1

```
(defun alternating-sum (w &optional f)
  (if w (+ (if f (- (car w)) (car w))
           (alternating-sum (cdr w) (not f)))
        0))

> (alternating-sum '(1 2 3 4 5 6 7))
4
```

Решение 3.152.2

```
(defun alternating-sum (w &optional f)
  (if w (+ (* (if f -1 1) (car w))
           (alternating-sum (cdr w) (not f)))
        0))

> (alternating-sum '(1 2 3 4 5 6 7))
4
```

Задача 3.153 max-product.lisp

Дан список целых чисел. Определить функцию, возвращающую самое большое произведение трех чисел из этого списка.

Решение 3.153.1

```
(defun max-product (w
  &aux
    (n1 (first w))
    (n2 (second w))
    (n3 (third w))
    (n4 (fourth w))
    (n5 (fifth w)))
  (cond ((null n3) 'less-than-3-items)
        ((null n4) (* n1 n2 n3))
```

```

((null n5) (max (* n1 n2 n3)
                 (* n1 n3 n4)
                 (* n1 n2 n4)
                 (* n2 n3 n4)))
(t (let* ((v (sort (list n1 n2 n3 n4 n5) #'>)))
      (max-product- (nthcdr 5 w)
                    (first v)
                    (second v)
                    (third v)
                    (fourth v)
                    (fifth v))))))

(defun max-product- (w m1 m2 m3 m4 m5
                    &aux
                      (a (car w))
                      (d (cdr w)))
  (cond ((null w) (max (* m1 m2 m3) (* m1 m4 m5)))
        ((> a m1) (max-product- d a m1 m2 m4 m5))
        ((> a m2) (max-product- d m1 a m2 m4 m5))
        ((> a m3) (max-product- d m1 m2 a m4 m5))
        ((< a m4) (max-product- d m1 m2 m3 a m4))
        ((< a m5) (max-product- d m1 m2 m3 m4 a))
        (t (max-product- d m1 m2 m3 m4 m5))))

> (max-product '(-10 -10 1 3 2))
300
> (max-product '(-1 -2 -3 -4 -5))
-6
> (max-product '(-1 -2 3 4 5))
60

```

Решение 3.153.2

```

(defun max-product (w
                    &aux
                      (n1 (first w))
                      (n2 (second w))
                      (n3 (third w))
                      (n4 (fourth w))
                      (n5 (fifth w)))
  (cond ((null n3) 'less-than-3-items)
        ((null n4) (* n1 n2 n3))
        ((null n5) (max (* n1 n2 n3)
                        (* n1 n3 n4)
                        (* n1 n2 n4)
                        (* n2 n3 n4))))
  (t (let ((v (sort (list n1 n2 n3 n4 n5) #'>)))
      (max-product- (nthcdr 5 w)
                    (first v)
                    (second v)
                    (third v)
                    (fourth v)
                    (fifth v))))))

```

```
(defun max-product- (w m1 m2 m3 m4 m5)
  (loop for a in w do
    (cond ((> a m1) (setf m3 m2) (setf m2 m1) (setf m1 a))
          ((> a m2) (setf m3 m2) (setf m2 a))
          ((> a m3) (setf m3 a))
          ((< a m4) (setf m5 m4) (setf m4 a))
          ((< a m5) (setf m5 a)))
    finally (return (max (* m1 m2 m3)
                        (* m1 m4 m5))))))

> (max-product '(-10 -10 1 3 2))
300
> (max-product '(-1 -2 -3 -4 -5))
-6
> (max-product '(-1 -2 3 4 5))
60
```

Задача 3.154 first-most-boring-element.lisp

Дан список символов. Определить функцию, возвращающую первый символ, который встречается подряд максимальное число раз.

Решение 3.154.1

```
(defun ws (w &optional (n 1) ac)
  (cond ((null w) (reverse ac))
        ((or (not (equalp (car w) (cadr w))) (equal w '(nil))))
        (ws (cdr w) 1 (cons (list (car w) n) ac)))
        ((ws (cdr w) (1+ n) ac))))

(defun first-most-boring-element (w &aux (v (ws w)))
  (car (find (loop for a in v maximize (cadr a)) v :key #'cadr)))

> (first-most-boring-element '(nil nil a b b b c c c d nil nil nil))
В
```

Задача 3.155 deepest.lisp

Определить функцию, возвращающую первый из самых глубоко расположенных атомов в произвольном списке.

Решение 3.155.1

```
(defun deepest (w)
  (cadr
   (labels
    ((deep (v n ac)
      (cond ((null v) ac)
            ((atom v) (if (> n (car ac)) `(,n ,v) ac))
            ((deep (cdr v) n (deep (car v) (1+ n) ac))))))
    (deep w 0))))
```

```
(deep w 0 `(0 nil))))))
> (deepest '((1) 5 ((8) 9 (4))))
8
```

Задача 3.156 *len.lisp*

Определите с помощью макроса `prog` аналог встроенной функции `length`, которая возвращает в качестве значения длину списка (количество элементов на верхнем уровне).

Решение 3.156.1

```
(defun len (w)
  (prog ((n 0))
    loop
    (unless w (return n))
    (pop w)
    (incf n)
    (go loop)))
> (len '(a b c d e))
5
```

Задача 3.157 *self-dual.lisp*

Напишите программу определения самодвойственности булевой функции, заданной списком ее значений.

Решение 3.157.1

```
(defun self-dual (w
  &aux
  (m (/ (length w) 2))
  (a (reverse (subseq w 0 m)))
  (b (nthcdr m w)))
  (notany #'= a b))
> (self-dual '(1 1 0 1 0 1 0 0))
T
```

Задача 3.158 *and-four.lisp*

Напишите с помощью условной формы `cond` функцию `and-four` для четырех аргументов.

Решение 3.158.1

```
(defun and-four (a b c d)
  (cond ((not a) nil)
        ((not b) nil)
```

```
((not c) nil)
((not d) nil)
(t))
```

```
> (and-four 1 2 3 4)
T
> (and-four 1 2 3 nil)
NIL
```

Решение 3.158.2

```
(defun and-four (a b c d)
  (cond (a (cond (b (cond (c (cond (d))))))))))

> (and-four 1 2 3 4)
4
> (and-four 1 2 nil 4)
NIL
```

Задача 3.159 evennn.lisp

I am always doing that which I can not do,
in order that I may learn how to do it.

Pablo Picasso

Написать предикат от аргумента-списка, определяющий наличие в нем четных элементов.

Решение 3.159.1

```
(defun evennn (w)
  (cond ((null w) nil)
        ((evenp (car w))
         ((evennn (cdr w)))))

> (evennn '(1 2))
T
```

Решение 3.159.2

```
(defun evennn (w)
  (cond ((null w) nil)
        ((-evenp (car w))
         ((evennn (cdr w)))))

(defun -evenp (n) (= (rem n 2) 0))

> (evennn '(1 2))
T

Решение 3.159.3
```

```
(defun evennn (w)
  (cond ((null w) nil)
        ((-evenp (car w))
         (evennn (cdr w)))))

(defun -evenp (n) (zerop (rem n 2)))
```

```
> (evennn '(1 2))
T
```

Решение 3.159.4

```
(defun evennn (w)
  (cond ((null w) nil)
        ((= (rem (car w) 2) 0)
         (evennn (cdr w)))))
```

```
> (evennn '(1 2))
T
```

Решение 3.159.5

```
(defun evennn (w)
  (cond ((null w) nil)
        ((= (rem (car w) 2) 0)
         (evennn (cdr w)))))
```

```
> (evennn '(1 2))
T
```

Решение 3.159.6

```
(defun evennn (w)
  (when w (if (evenp (car w)) t (evennn (cdr w)))))
```

```
> (evennn '(1 2))
T
```

Решение 3.159.7

```
(defun evennn (w) (loop for a in w thereis (evenp a)))
```

```
> (evennn '(1 2))
T
```

Решение 3.159.8

```
(defun evennn (w) (some #'evenp w))
```

```
> (evennn '(1 2))
T
```


[illegible]

Решение 3.161.4

```
(defun knock-into-one (w)
  (with-output-to-string (s)
    (knock s w)))

(defun knock (s w)
  (destructuring-bind (&optional a &rest b) w
    (when a (write-string a s)
      (when b (knock s b))))))

> (time (knock-into-one (loop for a repeat 1000 collect "a"))))
Real time: 0.0080028 sec.
Run time: 0.0 sec.
Space: 29340 Bytes
```

[illegible]

Решение 3.161.5

[illegible]


```
(if w (digits-number (cdr w) (+ (car w) (* ac 10))) ac))

> (digits-number '(1 2 3 4))
1234
> (digits-number '(0 2 3 4))
234
```

Задача 3.163 *twins.lisp*

Проверить, есть в заданном списке хотя бы два одинаковых элемента.

Решение 3.163.1

```
(defun twins (w)
  (cond ((null w) 'no)
        ((member (car w) (cdr w) :test #'equalp) 'yes)
        (t (twins (cdr w)))))

> (twins '((a) (a)))
YES
```

Задача 3.164 *sum-equals.lisp*

Дейкстра: А как вы запишите более сложное выражение? Например, сумму всех элементов матрицы, которые равны сумме индексов соответствующих строк и столбцов.

Айверсон: $++ / (M = ?1^{(o)} + ?1)M//$

Кеннет Айверсон, создатель языков APL и J

Найти сумму всех элементов матрицы, которые равны сумме индексов соответствующих строк и столбцов.

Решение 3.164.1

```
(defun sum-equals (w)
  (loop for a in w for b from 0
        sum (sum-equal a b)))

(defun sum-equal (w n)
  (loop for a in w for b from 0
        when (equal a (+ n b)) sum a))

> (sum-equals '((0 1 2 3) (1 2 3 4) (2 3 4 5) (3 4 5 6)))
48
```

Решение 3.164.2

```
(defun sum-equals (w &optional n)
  (loop for a in w for b from 0
```

```

      sum (if (and n (equal a (+ n b))) a
              (sum-equals a b))))
> (sum-equals '((0 1 2 3) (1 2 3 4) (2 3 4 5) (3 4 5 6)))
48

```

Задача 3.165 decreasing.lisp

Определить функцию, которая проверяет, образует ли список убывающую последовательность.

Решение 3.165.1

```

(defun decreasing (w) (eval (cons '> w)))

> (decreasing '(3 2 1))
T
> (decreasing '(3 2 2))
NIL

```

Решение 3.165.2

```

(defun decreasing (w) (when w (eval (cons '> w))))

> (decreasing '())
NIL

```

Решение 3.165.3

```

(defun decreasing (w) (if w (eval (cons '> w)) t))

> (decreasing '())
T

```

Решение 3.165.4

```

(defun decreasing (w) (when w (apply #'> w)))

> (decreasing '(3 2 1))
T
> (decreasing '(3 2 2))
NIL
> (decreasing '())
NIL

```

Решение 3.165.5

```

(defun decreasing (w)
  (cond ((null (cdr w))
        (t nil))
        ((> (car w) (cadr w)) (decreasing (cdr w)))
        (t nil)))

```

```
> (decreasing '(3 2 1))
T
> (decreasing '(3 2 2))
NIL
> (decreasing '(3))
T
```

Решение 3.165.6

```
(defun decreasing (w)
  (if (cdr w) (when (> (car w) (cadr w)) (decreasing (cdr w))) t))

> (decreasing '(3 2 1))
T
> (decreasing '(3 2 2))
NIL
> (decreasing '(3))
T
```

Решение 3.165.7

```
(defun decreasing (w)
  (when (some #'numberp w)
    (decr (remove-if-not #'numberp w))))

(defun decr (w)
  (cond ((null (cdr w))
        ((> (car w) (cadr w)) (decr (cdr w)))
        (t nil)))

> (decreasing '(a 3 b 2 c 1 d))
T
> (decreasing '(a 3 b 2 c 2 d))
NIL
> (decreasing '(3))
T
```

Задача 3.166 *eqq.lisp*

Определить функцию, которая проверяет, одинаковы ли элементы в списке.

Решение 3.166.1

```
(defun eqq (w)
  (loop for a in (cdr w) always (equal (car w) a)))

> (eqq '(a a))
T
```

Решение 3.166.2

```
(defun eqq (w)
```

```
(cond ((null (cdr w)))
      ((equal (car w) (cadr w)) (eqq (cdr w)))
      (nil)))
```

```
> (eqq '(a a))
T
```

Решение 3.166.3

```
(defun eqq (w) (not (remove (car w) w)))
```

```
> (eqq '(a a))
T
```

Решение 3.166.4

```
(defun eqq (w)
  (if (cdr w)
      (when (equal (car w) (cadr w)) (eqq (cdr w)))
      t))
```

```
> (eqq '(a a))
T
```

Задача 3.167 duplicatep.lisp

Определить функцию, которая проверяет повторное появление элементов в списке.

Решение 3.167.1

```
(defun duplicatep (w)
  (when w (if (member (car w) (cdr w)) t
              (duplicatep (cdr w)))))
```

```
> (duplicatep '(1 2 3 2))
T
```

Решение 3.167.2

```
(defun duplicatep (w) (loop for a in w thereis (> (count a w) 1)))
```

```
> (duplicatep '(1 2 3 2))
T
```

Задача 3.168 deep-sum.lisp

Определить функцию, возвращающую сумму числовых элементов списка, учитывая элементы подсписков.

Решение 3.168.1


```
(defun sum-deep (w &aux (a (car w)))
  (if w (+ (cond ((listp a) (sum-deep a))
                 ((numberp a) a)
                 (t 0))
    (sum-deep (cdr w)))
  0))

> (sum-deep '(1 2 3 a b ((c)) ((7 8))))
21
```

Решение 3.168.2

```
(defun sum-deep (w)
  (loop for a in w if (listp a) sum (sum-deep a)
        else if (numberp a) sum a))

> (sum-deep '(1 2 3 a b ((c)) ((7 8))))
21
```

Решение 3.168.3 (автор - transformator.t, www.cyberforum.ru)

```
(defun sum-deep (w)
  (reduce #'+ w :key
    #'(lambda (a)
      (typecase a
        (list (sum-deep a))
        (number a)
        (otherwise 0)))))

> (sum-deep '(1 2 3 a b ((c)) ((7 8))))
21
```

Задача 3.169 nearp.lisp

Определить функцию, проверяющую, есть ли в заданном списке хотя бы один элемент, который равен следующему за ним (по кругу) элементу.

Решение 3.169.1

```
(defun nearp (w &optional (a (car w)) f)
  (cond ((null (cdr w)) (when f (eq a (car w))))
        ((eq (car w) (cadr w))
         ((nearp (cdr w) a t)))

> (nearp '(1 2 1))
T
> (nearp '(1 2 2))
T
```

Задача 3.170 list-insidep.lisp

Определить функцию, проверяющую, есть ли в заданном списке хотя бы один подсписок.

Решение 3.170.1

```
(defun list-insidep (w)
  (if (null w) nil
      (if (atom (car w))
          (list-insidep (cdr w))
          t)))
```

```
> (list-insidep '(a (b)))
T
```

Решение 3.170.2

```
(defun list-insidep (w)
  (if (null w) nil
      (if (listp (car w))
          t
          (list-insidep (cdr w)))))
```

```
> (list-insidep '(a (b)))
T
```

Решение 3.170.3

```
(defun list-insidep (w)
  (cond ((null w) nil)
        ((atom (car w)) (list-insidep (cdr w)))
        (t)))
```

```
> (list-insidep '(a (b)))
T
```

Решение 3.170.4

```
(defun list-insidep (w)
  (when w (if (atom (car w)) (list-insidep (cdr w)) t)))
```

```
> (list-insidep '(a (b)))
T
```

Решение 3.170.5

```
(defun list-insidep (w) (loop for a in w thereis (listp a)))
```

```
> (list-insidep '(a (b)))
T
```

Решение 3.170.6

```
(defun list-insidep (w) (some #'listp w))

> (list-insidep '(a (b)))
T
```

Задача 3.171 string-sine-length.lisp

Дан многомерный массив. Массив может содержать элементы различных типов – целые, вещественные, комплексные числа, строки, символы, списки, другие массивы. Определить функцию, возвращающую сумму длин строк, не содержащих символов #\a, #\i и #\u.

Решение 3.171.1

```
(defun string-sine-length (v a)
  (loop for e across (make-array
                     (apply #'* (array-dimensions a))
                     :displaced-to a)
    when (and (stringp e) (loop for b in v never (find b e)))
    sum (length e)))

> (string-sine-length '(#\a #\i #\u) #3A(((1 "abcd") ("ijk" "klm")
("xvn" "uxy"))) ((1 2) ("zzzzzzzzzz" 4) (5 6)))
16
```

Задача 3.172 max-cnt.lisp

Определить функцию, возвращающую количество элементов, равных максимальному элементу числового списка.

Решение 3.172.1

```
(defun max-cnt (w) (cnt w (mx w)))

(defun cnt (w m)
  (cond ((null w) 0)
        ((= (car w) m) (1+ (cnt (cdr w) m)))
        (t (cnt (cdr w) m))))

(defun mx (w &optional (m (car w)))
  (cond ((null w) m)
        ((> (car w) m) (mx (cdr w) (car w)))
        (t (mx (cdr w) m))))

> (max-cnt '(1 0 1 0 1))
3
```

Решение 3.172.2

```
(defun max-cnt (w) (cnt w (mx w)))
```

```

(defun cnt (w m)
  (cond ((null w) 0)
        ((= (car w) m) (1+ (cnt (cdr w) m)))
        (t (cnt (cdr w) m))))

(defun mx (w &optional (m (car w)))
  (cond ((null w) m)
        (t (mx (cdr w) (max m (car w))))))

> (max-cnt '(1 0 1 0 1))
3

```

Решение 3.172.3

```

(defun max-cnt (w) (cnt w (mx w)))

(defun cnt (w m)
  (cond ((null w) 0)
        ((= (car w) m) (1+ (cnt (cdr w) m)))
        (t (cnt (cdr w) m))))

(defun mx (w &optional (m (car w)))
  (if w (mx (cdr w) (max m (car w))) m))

> (max-cnt '(1 0 1 0 1))
3

```

Решение 3.172.4

```

(defun max-cnt (w &optional (m (car w)) (n 0))
  (cond ((null w) n)
        ((> (car w) m) (max-cnt (cdr w) (car w) 1))
        ((= (car w) m) (max-cnt (cdr w) m (1+ n)))
        (t (max-cnt (cdr w) m n))))

> (max-cnt '(1 0 1 0 1))
3

```

Решение 3.172.5

```

(defun max-cnt (w
  &optional (n 0) (m (car w))
  &aux (a (car w)) (d (cdr w)))
  (cond ((null w) n)
        ((> a m) (max-cnt d 1))
        ((= a m) (max-cnt d (1+ n) m))
        ((max-cnt d n m))))

> (max-cnt '(1 0 1 0 1))
3

```

Решение 3.172.6

```
(defun max-cnt (w &aux (m (loop for a in w maximize a)))
  (loop for a in w when (= a m) count a))
```

```
> (max-cnt '(1 0 1 0 1))
3
```

Решение 3.172.7

```
(defun max-cnt (w)
  (count (reduce #'max w) w))
```

```
> (max-cnt '(1 0 1 0 1))
3
```

Задача 3.173 product-deep.lisp

Why restrict myself to a GUI? Emacs can work just fine without one. I can shell into a remote system and fire it up in seconds.

Ben Sinclair

Определить функцию, возвращающую произведение элементов списка.

Решение 3.173.1

```
(defun product-deep (w &aux (a (car w)))
  (if w (* (car a) (caadr a) (product-deep (cdr w))) 1))
```

```
> (product-deep '((1 (2)) (3 (4)) (5 (6))))
720
```

Решение 3.173.2

```
(defun product-deep (w &optional (p 1))
  (cond ((null w) p)
        ((atom w) (* w p))
        ((product-deep (car w) (product-deep (cdr w) p)))))
```

```
> (product-deep '((1 (2)) (3 (4)) (5 (6))))
720
```

Решение 3.173.3

```
(defun product-deep (w)
  (reduce #'* w :key #'(lambda (a) (* (car a) (caadr a)))))
```

```
> (product-deep '((1 (2)) (3 (4)) (5 (6))))
720
```

Решение 3.173.4

```
(defun product-deep (w &optional (p 1))
  (loop for a in w do
    (if (listp a) (setf p (* p (product-deep a)))
        (setf p (* p a))))
  p)

> (product-deep '((1 (2)) (3 (4)) (5 (6))))
720
```

Решение 3.173.5

```
(defun product-deep (w)
  (reduce #'*(flatten w)))

(defun flatten (w &optional acc)
  (cond ((null w) acc)
        ((atom w) (cons w acc))
        ((flatten (car w) (flatten (cdr w) acc)))))

> (product-deep '((1 (2)) (3 (4)) (5 (6))))
720
```

Решение 3.173.6

```
(defun product-deep (w)
  (reduce #'*(flatten w)))

(defun flatten (w)
  (loop for a in w
    if (listp a) append (flatten a)
    else collect a))

> (product-deep '((1 (2)) (3 (4)) (5 (6))))
720
```

Задача 3.174 sorted.lisp

Определить, упорядочены ли элементы в одноуровневом списке, содержащем латинские буквы, по алфавиту.

Решение 3.174.1

```
(defun sorted (w v)
  (cond ((null v)
        ((null w) (null v))
        ((eq (car w) (car v)) (sorted w (cdr v)))
        ((sorted (cdr w) v))))

> (sorted '(a b c d e f g h i j k l m n o p q r s t u v w x y z) '(g g
h h z z))
T
```

Задача 3.175 *count-triples.lisp*

It may be natural to ask: "if Scheme is less familiar, how can it be simpler than Python?". The answer is that the simplicity is measured by the number of rules that are needed to describe a language. Scheme consists of significantly less rules than Python, but it is very likely that you have already internalized some of the rules that govern Python at school. For example Python defines the priority of operators that is similar to the convention applied in mathematics, whereas Lisp requires you to be precise and doesn't rely on any implicit invisible rules. So while in Python you can write `1 + 2 * 3` to mean `1 + (2 * 3)` in Lisp you need to write the same thing as `(+ 1 (* 2 3))`, because it doesn't leave anything implicit. You can also observe that Python treats mathematical operators specially, and allows to infix them, whereas Lisp treats them like every other function. One could go on and ask: why should I change my habits about writing arithmetic expressions just to conform to some spartan set of rules? What does this striving for some abstract simplicity buy me? And there are some very specific answers to this question. First, this sort of simplicity makes your programs easy to process, so you can easier create better tools for processing and manipulating your programs. Parsing Lisp syntax is very easy and you really don't need any specialized "parser generators" to do that. Not to mention the fact, that most Lisp systems already provide the parser for you, so you can easily use the Lisp expressions as a data serialization format, much more succinct than XML and more powerful than JSON. Lastly, you can extend the syntax of your language with new forms using macros. Even though you don't need that very often, you do need it sometimes, and not having this feature just doesn't seem rational.

Panicz Godek

Определить функцию, возвращающую количество троек подряд идущих одинаковых элементов числового списка.

Решение 3.175.1

```
(defun count-triples (w)
  (cond ((null (cddr w)) 0)
        ((= (car w) (cadr w) (caddr w)) (1+ (count-triples (cdr w))))
        (t (count-triples (cdr w)))))
```

```
> (count-triples '(1 1 3 3 3 3 4 3 4 4 4 2 1 1 1))
4
```

Решение 3.175.2

```
(defun count-triples (w)
  (if (cddr w) (+ (if (= (car w) (cadr w) (caddr w)) 1 0)
                  (count-triples (cdr w)))
      0))
```

```
> (count-triples '(1 1 3 3 3 3 4 3 4 4 4 2 1 1 1))
4
```

Решение 3.175.3

```
(defun count-triples (w &optional (n 0))
  (if (cddr w)
      (count-triples (cdr w)
                      (+ n (if (= (car w) (cadr w) (caddr w)) 1 0)))
      n))
```

```
> (count-triples '(1 1 3 3 3 3 4 3 4 4 4 2 1 1 1))
4
```

Решение 3.175.4

```
(defun count-triples (w)
  (loop for a in w for b in (cdr w) for c in (cddr w)
        when (= a b c) count a))
```

```
> (count-triples '(1 1 3 3 3 3 4 3 4 4 4 2 1 1 1))
4
```

Задача 3.176 our-if.lisp

Определить макрос our-if.

Решение 3.176.1

```
(defmacro our-if (test success failure)
  (subst test 'test
    (subst success 'success
      (subst failure 'failure
        '(cond (test success) (failure)))))))
```

```
> (our-if (= 1 1) 2 3)
```


2

Решение 3.176.2

```
(defmacro our-if (p c &optional e) `(cond (,p ,c) (,e)))

> (our-if (= 1 1) 2 3)
2
```

Задача 3.177 sum-numbers.lisp

Определить функцию, которая находила бы сумму всех числовых атомов в списке, учитывая все атомы подсписков.

Решение 3.177.1

```
(defun sum-numbers (w &optional (ac 0) &aux (a (car w)))
  (if w (sum-numbers (cdr w)
                     (+ ac
                        (cond ((numberp a) a)
                              ((listp a) (sum-numbers a))
                              (0))))
    ac))

> (sum-numbers '(1 2 a (((3 x))) b 4))
10
```

Решение 3.177.2

```
(defun sum-numbers (w)
  (loop for a in w
        sum (cond ((numberp a) a)
                  ((listp a) (sum-numbers a))
                  (0))))

> (sum-numbers '(1 2 a (((3 x))) b 4))
10
```

Решение 3.177.3

```
(defun sum-num (w)
  (loop for a in w
        if (numberp a) sum a
        else if (listp a) sum (sum-num a)))

> (sum-numbers '(1 2 a (((3 x))) b 4))
10
```

Решение 3.177.4

```
(defun sum-numbers (w)
  (typecase w
```

```

      (cons (+ (sum-numbers (car w))
              (sum-numbers (cdr w))))
      (number w)
      (t 0)))

> (sum-numbers '(1 2 a ((3 x)) b 4))
10

```

Задача 3.178 cnt-smb.lisp

90% of the teaching on programming misses the fact that programming is NOT a study and research endeavor. You only really learn programming when you are writing code and solving real problems. Programming is a PRACTICAL subject.

Mike Anthony

Список может содержать символы строки и цифры. Определить функцию, возвращающую количество символов.

Решение 3.178.1

```

(defun cnt-smb (w)
  (cond ((null w) 0)
        ((symbolp (car w)) (1+ (cnt-smb (cdr w))))
        (t (cnt-smb (cdr w)))))

> (cnt-smb '(ab cd ef 1 "s"))
3

```

Решение 3.178.2

```

(defun cnt-smb (w)
  (if w (+ (if (symbolp (car w)) 1 0) (cnt-smb (cdr w))) 0))

> (cnt-smb '(ab cd ef 1 "s"))
3

```

Решение 3.178.3

```

(defun cnt-smb (w)
  (loop for a in w when (symbolp a) count a))

> (cnt-smb '(ab cd ef 1 "s"))
3

```

Решение 3.178.4

```

(defun cnt-smb (w) (count-if #'symbolp w))

```

```
> (cnt-smb '(ab cd ef 1 "s"))
3
```

Задача 3.179 cnt-char.lisp

When I was writing my thesis on a topic related to the Vehicle Routing Problem, I was forced to use Java. I was never given any choice — or the illusion of it. I spent 80% of my time on anything but developing my ideas into prototype software. I recently re-wrote the library in Common Lisp and released it open-source. I've been working on it just over a month, and I did so with a big smile. I couldn't bother counting the number of lines, but to give you an idea, my Java source code was 500.4 kb versus 78.8 kb in Common Lisp.

Marc Kuo

Список может содержать символы строки и цифры. Определить функцию, возвращающую количество значков (литер) в символах, строках и цифрах.

Решение 3.179.1

```
(defun cnt-char (w &aux (a (car w)))
  (cond ((null w) 0)
        ((symbolp a) (+ (length (string a)) (cnt-char (cdr w))))
        ((numberp a)
         (+ (length (write-to-string a)) (cnt-char (cdr w))))
        ((stringp a) (+ (length a) (cnt-char (cdr w))))
        (t (cnt-char (cdr w)))))
```

```
> (cnt-char '(ab cd ef 100 "string"))
15
```

Решение 3.179.2

```
(defun cnt-char (w &aux (a (car w)))
  (if w (+ (length (cond ((symbolp a) (string a))
                        ((numberp a) (write-to-string a))
                        ((stringp a) a)
                        (t "")))
          (cnt-char (cdr w)))
  0))
```

```
> (cnt-char '(ab cd ef 100 "string"))
15
```

Решение 3.179.3

```
(defun cnt-char (w)
  (loop for a in w sum
    (length (cond ((symbolp a) (string a))
                  ((numberp a) (write-to-string a))
                  ((stringp a) a)
                  ("")))))

> (cnt-char '(ab cd ef 100 "string"))
15
```

Решение 3.179.4

```
(defun cnt-char (w)
  (reduce #'+ w :key
    #'(lambda (a)
      (typecase a
        (symbol (length (string a)))
        (number (length (write-to-string a)))
        (string (length a))
        (otherwise 0)))))

> (cnt-char '(ab cd ef 100 "string"))
15
```

Решение 3.179.5

```
(defun cnt-char (w)
  (reduce #'+ w :key
    #'(lambda (a)
      (length (typecase a
        (symbol (string a))
        (number (write-to-string a))
        (string a)
        (otherwise ""))))))

> (cnt-char '(ab cd ef 100 "string"))
15
```

Решение 3.179.6

```
(defun cnt-char (w)
  (reduce #'+ w :key
    #'(lambda (a)
      (length (typecase a
        (symbol (string a))
        (number (write-to-string a))
        (string a)
        (t ""))))))

> (cnt-char '(ab cd ef 100 "string"))
15
```

Задача 3.180 count-string-with-zero.lisp

Personally, I feel inspired whenever I open Emacs. Like a craftsman entering his workshop, I feel a realm of possibility open before me. I feel the comfort of an environment that has evolved over time to fit me perfectly - an assortment of packages and keybindings which help me bring ideas to life day after day.

Daniel Higginbotham «Clojure for the Brave and True»

Дана прямоугольная матрица. Определить функцию, возвращающую количество строк матрицы, которые содержат хотя бы один 0.

Решение 3.180.1

```
(defun count-string-with-zero (w)
  (loop for a in w when (member 0 a) sum 1))

> (count-string-with-zero '((1 0 3 4) (2 3 0 5) (5 6 4 3)))
2
```

Решение 3.180.2

```
(defun count-string-with-zero (w)
  (cond ((null w) 0)
        ((member 0 (car w)) (1+ (count-string-with-zero (cdr w))))
        (t (count-string-with-zero (cdr w)))))

> (count-string-with-zero '((1 0 3 4) (2 3 0 5) (5 6 4 3)))
2
```

Решение 3.180.3

```
(defun count-string-with-zero (w)
  (if w (+ (if (member 0 (car w)) 1 0)
          (count-string-with-zero (cdr w)))
    0))

> (count-string-with-zero '((1 0 3 4) (2 3 0 5) (5 6 4 3)))
2
```

Решение 3.180.4

```
(defun count-string-with-zero (w &optional (n 0))
  (cond ((null w) n)
        ((member 0 (car w)) (count-string-with-zero (cdr w) (1+ n)))
        ((count-string-with-zero (cdr w) n))))
```

```
> (count-string-with-zero '((1 0 3 4) (2 3 0 5) (5 6 4 3)))
2
```

Решение 3.180.5

```
(defun count-string-with-zero (w &optional (n 0))
  (if w (count-string-with-zero (cdr w)
                                (if (member 0 (car w)) (1+ n) n))
      n))
```

```
> (count-string-with-zero '((1 0 3 4) (2 3 0 5) (5 6 4 3)))
2
```

Решение 3.180.6

```
(defun count-string-with-zero (w)
  (count-if #'(lambda (a) (member 0 a)) w))
```

```
> (count-string-with-zero '((1 0 3 4) (2 3 0 5) (5 6 4 3)))
2
```

Решение 3.180.7

```
(defun count-string-with-zero (w)
  (loop for a in w when (member 0 a) count a))
```

```
> (count-string-with-zero '((1 0 3 4) (2 3 0 5) (5 6 4 3)))
2
```

Решение 3.180.8

```
(defun count-string-with-zero (w)
  (loop for a in w count (member 0 a)))
```

```
> (count-string-with-zero '((1 0 3 4) (2 3 0 5) (5 6 4 3)))
2
```

Решение 3.180.9

```
(defun count-string-with-zero (w)
  (reduce #'+ w :key #'(lambda (a) (if (member 0 a) 1 0))))
```

```
> (count-string-with-zero '((1 0 3 4) (2 3 0 5) (5 6 4 3)))
2
```

Решение 3.180.10

```
(defun count-string-with-zero (w)
  (reduce #'+ w :key #'(lambda (a) (if (find 0 a) 1 0))))
```

```
> (count-string-with-zero '((1 0 3 4) (2 3 0 5) (5 6 4 3)))
2
```

Задача 3.181 *very-big-sum.lisp*

First, Lisp is the refinement of the writing systems that we have been developing throughout the millennia. The uniform syntax of Lisp is the only example of a notation that I know of which is self-contained (or, as — quite ironically — Phil Wadler would say it — it's the notation that is discovered rather than invented).

Panicz Godek

С клавиатуры последовательно вводятся: *n* (количество чисел) и сами числа. Определить функцию, выводящую на экран сумму введенных чисел.

Пример ввода:

```
5
20349850239
30945809484
67934859492
76599345204
56830490420
```

Образец вывода:

```
252660354839
```

Решение 3.181.1

```
(defun very-big-sum ()
  (write (reduce #'+ (loop repeat (read) collect (read))))))

> (very-big-sum)
5
20349850239
30945809484
67934859492
76599345204
56830490420
252660354839
252660354839
```

Решение 3.181.2

```
(defun very-big-sum ()
  (write (reduce #'+ (loop repeat (read) collect (read))))))

> (very-big-sum)
5
20349850239
30945809484
67934859492
```

```
76599345204
56830490420
252660354839
NIL
```

Задача 3.182 *simple-array-sum.lisp*

Every great developer you know got there by solving problems they were unqualified to solve until they actually did it.

Patrick McKenzie

С клавиатуры последовательно вводятся: в первой строке *n* (количество чисел) и во второй строке сами числа, разделенные пробелами. Определить функцию, выводящую на экран сумму введенных чисел.

Пример ввода:

```
5
20349850239 30945809484 67934859492 76599345204 56830490420
```

Образец вывода:

```
252660354839
```

Решение 3.182.1

```
(defun simple-array-sum ()
  (read)
  (format t "~a"
    (loop for a in (read-from-string
                    (concatenate 'string "(" (read-line) "))")
      sum a)))

> (simple-array-sum)
5
20349850239 30945809484 67934859492 76599345204 56830490420
252660354839
NIL
```

Решение 3.182.2

```
(defun simple-array-sum ()
  (read)
  (format t "~a"
    (reduce #'+
      (read-from-string
        (concatenate 'string "(" (read-line) ")")))))

> (simple-array-sum)
5
```



```
20349850239 30945809484 67934859492 76599345204 56830490420
252660354839
NIL
```

Решение 3.182.3

```
(defun simple-array-sum ()
  (format t "~a"
    (reduce #'+ (loop repeat (read) collect (read)))))

> (simple-array-sum)
5
20349850239 30945809484 67934859492 76599345204 56830490420
252660354839
NIL
```

Задача 3.183 staircase.lisp

Even if you don't intend anybody else to read your code, there's still a very good chance that somebody will have to stare at your code and figure out what it does: That person is probably going to be you, twelve months from now.

Raymond Chen

С клавиатуры вводится положительное целое число n . Определить функцию, выводящую на экран лестницу из n строк по n знаков #, посредством пробелов сдвинутых вправо.

Пример ввода:

6

Образец вывода:

```
  #
 ##
###
####
#####
#####
```

Решение 3.183.1

```
(defun sq (n s)
  (loop repeat n collect s))

(defun make (m s1 s2)
  (loop for a from 1 to m do
    (format t "~a~%" (coerce (append (sq (- m a) s1)
```

```

                                (sq a s2))
                                'string)))

(defun staircase ()
  (make (read) #\Space #\#))

> (staircase)
6
#
##
###
####
#####
#####
#####
NIL

```

Задача 3.184 *deep-twins.lisp*

A complex system that works is invariably found to have evolved from a simple system that worked. The inverse proposition also appears to be true: A complex system designed from scratch never works and cannot be made to work.

John Gall

So much complexity in software comes from trying to make one thing do two things.

Ryan Singer

Написать функцию, которая определяет, содержит ли список с подписками повторяющиеся элементы.

Решение 3.184.1

```

(defun deep-twins (w) (twins (flatten w)))

(defun flatten (w &optional ac)
  (cond ((null w) ac)
        ((atom w) (cons w ac))
        ((flatten (car w) (flatten (cdr w) ac)))))

(defun twins (w)
  (cond ((null w) nil)
        ((member (car w) (cdr w)) t)
        ((twins (cdr w)))))

> (deep-twins '(1 2 (3 (4 5) 6 1) 7))
T
> (deep-twins '(1 2 (3 (4 5) 6) 7))

```

NIL

Задача 3.185 *join-with-delimiter.lisp*

I could think of three languages off the top of my head that are easy to learn the syntax: Lisp (in all its flavors), Forth (in all its flavors), and Smalltalk (in all its flavors). They have very simple syntax, and they are also very easy to extend. However, to do useful things with them, they typically have extensive libraries. Consider Scheme, for example – a flavor of Lisp – which has a 50-page specification, but if you wish to have a version that has extensive libraries, you’re now thumbing through 800 more pages. Furthermore, because of their simplicity and extendability, they enable programmers to do very difficult things, things that other types of languages can only dream of doing. Because these things are difficult, however, these languages sometimes get the reputation of being difficult to learn. The hardest programming languages to learn? They are things like C++, Perl and PHP. They are difficult to extend in a meaningful way, they have tons of inconsistent features that you have to fully understand, because one little thing can completely change the meaning of your code, and they have a huge number of inconsistencies and pitfalls that need to be remembered, lest you constantly fall into ridiculous gotchas. Ironically enough, they are offered as “easy to learn” because they have C-like mathematical constructs, complete with precedence, but the precedence is one of the major reasons why these languages are difficult to extend. Whether you’re new to programming, or have merely been doing it for a living, do yourself a favor: learn Lisp, Smalltalk and Forth. You’ll discover things that you wouldn’t have ever dreamed were possible!

Alpheus Madsen

Определить функцию, которая для одноуровневого списка строк формирует итоговую строку сцеплением всех элементов списка. Элементы в итоговой строке разделены знаком двоеточие.

Решение 3.185.1

```
(defun join-with-delimiter (w)
  (if w (concatenate 'string
                     (car w)
                     (if (cdr w) ":" ""))
      (join-with-delimiter (cdr w)))
  ""))
```

```
> (join-with-delimiter '("aa" "bb" "cc"))
"aa:bb:cc"
```

Решение 3.185.2

```
(defparameter s '("Aa" "Ss" "Dd" " " "Ff" "Gg" "Hh" "" "D"))
```

```
(defun join-with-delimiter (w &optional (ac ""))
  (if w (join-with-delimiter
      (cdr w)
      (concatenate 'string ":" (car w) ac))
      (reverse (string-left-trim ":" ac))))
```

```
> (join-with-delimiter '("aa" "bb" "cc"))
"aa:bb:cc"
> (time (loop repeat 1000000 do (join-with-delimiter s)))
Real time: 33.624516 sec.
Run time: 33.609375 sec.
Space: 760003944 Bytes
GC: 1450, GC time: 8.03125 sec.
NIL
```

Решение 3.185.3 (автор - helter, www.cyberforum.ru)

```
(defparameter s '("Aa" "Ss" "Dd" " " "Ff" "Gg" "Hh" "" "D"))
```

```
(defun join-with-delimiter (list-of-strings)
  (format nil "~{~A~^::~~}" list-of-strings))
```

```
> (join-with-delimiter '("aa" "bb" "cc"))
"aa:bb:cc"
> (time (loop repeat 1000000 do (join-with-delimiter s)))
Real time: 69.32035 sec.
Run time: 69.203125 sec.
Space: 3268003944 Bytes
GC: 6234, GC time: 37.359375 sec.
NIL
```

Решение 3.185.4 (автор - transformator.t, www.cyberforum.ru)

```
(defparameter s '("Aa" "Ss" "Dd" " " "Ff" "Gg" "Hh" "" "D"))
```

```
(defun join-with-delimiter (list delimiter)
  (declare (list list) (character delimiter))
  (if (and (not (null list)) (every #'stringp list))
```

```

(loop
  with length-out of-type fixnum
  = (1- (reduce #' + list
               :key
               #' (lambda (x)
                     (let ((length (length
                                   (the simple-string x))))
                       (if (plusp length) (1+ length) 0))))))
  with out = (make-string (max length-out 0))
  with last of-type fixnum = -1
  for idx of-type fixnum from 0
  for string in list
  for string-length = (length (the simple-string string))
  do
    (loop
      for char across string
      do (setf (aref out (incf last)) char))
    (when (and (plusp string-length) (< (+ last 2) length-out))
      (setf (aref out (incf last)) delimiter))
    finally
      (return out))
  ""))

```

```

> (join-with-delimiter '("aa" "bb" "cc") #\:)
"aa:bb:cc"
> (time (loop repeat 1000000 do (join-with-delimiter s #\:)))
Real time: 30.259645 sec.
Run time: 30.25 sec.
Space: 92003944 Bytes
GC: 176, GC time: 0.8125 sec.
NIL

```

Решение 3.185.5 (автор - helter, www.cyberforum.ru)

```

(defparameter s '("Aa" "Ss" "Dd" " " " " "Ff" "Gg" "Hh" "" "D"))

(defun join-with-delimiter (strings delimiter)
  (check-type strings (satisfies list-of-strings-p))
  (check-type delimiter character)
  (if (endp strings)
      ""
      (let ((total-length (1- (loop for string in strings
                                   sum (1+ (length string))))))
        (check-type total-length fixnum)
        (locally
          (declare (optimize (speed 3) (safety 0)))
          (let ((result (make-string total-length))
                (first-string (the string (first strings))))
            (setf (subseq result 0) first-string)
            (loop for start fixnum = (length first-string)
                  then (+ start (length string) 1)
                  for string string in (rest strings)
                  do (setf (char result start) delimiter)))))))

```

```

      (loop for i fixnum from (1+ start)
            for c character across string
            do (setf (schar result i) c))
      finally (return result))))))

(defun list-of-strings-p (list)
  (check-type list list)
  (loop for x in list always (stringp x)))

> (join-with-delimiter '("aa" "bb" "cc") #\:)
"aa:bb:cc"
> (time (loop repeat 1000000 do (join-with-delimiter s #\:)))
Real time: 28.387346 sec.
Run time: 28.375 sec.
Space: 96003944 Bytes
GC: 183, GC time: 0.96875 sec.
NIL

```

Задача 3.186 ascendingp.lisp

When Amazon got its start, we had brilliant engineers... They wrote the Obidos webserver. Obidos made Amazon successful... Obidos was a key cornerstone of Amazon's initial success... They all used Emacs, of course. Hell, Eric Benson was one of the authors of XEmacs. All of the greatest engineers in the world use Emacs. The world-changer types... I'm talking about the greatest software developers of our profession, the ones who changed the face of the industry. The James Goslings, the Larry Walls, the Paul Grahams, the Jamie Zawinskis, the Eric Bensons. Real engineers use Emacs. You have to be way smart to use it well, and it makes you incredibly powerful if you can master it. Go look over Paul Nordstrom's shoulder while he works sometime, if you don't believe me. It's a real eye-opener for someone who's used Visual Blub .NET-like IDEs their whole career.

Steve Yegge

Определить функцию, которая проверяет, является ли отсортированным в порядке возрастания список, состоящий из чисел.

Решение 3.186.1

```

(defun ascendingp (w)
  (cond ((null (cdr w))
        ((< (car w) (cadr w)) (ascendingp (cdr w)))
        (t nil)))

```

```
> (ascendingp '(0 1 2 3))
T
```

Решение 3.186.2

```
(defun ascendingp (w)
  (cond ((null (cdr w))
        ((>= (car w) (cadr w)) nil)
        ((ascendingp (cdr w))))))
```

```
> (ascendingp '(0 1 2 3))
T
```

Решение 3.186.3

```
(defun ascendingp (w)
  (loop for a in w
        for b in (cdr w)
        always (< a b)))
```

```
> (ascendingp '(0 1 2 3))
T
```

Решение 3.186.4

```
(defun ascendingp (w)
  (eval (cons '< w)))
```

```
> (ascendingp '(0 1 2 3))
T
```

Решение 3.186.5

```
(defun ascendingp (w)
  (apply #'< w))
```

```
> (ascendingp '(0 1 2 3))
T
```

Задача 3.187 length4p.lisp

Определить функцию, которая проверяет, является ли её аргумент списком из четырёх элементов.

Решение 3.187.1

```
(defun length4p (w)
  (and (listp w) (= (length w) 4)))
```

```
> (length4p '(1 2 3 4))
T
```

```
> (length4p '(1 2 3))
NIL
> (length4p 4)
NIL
```

Задача 3.188 *length4-atomp.lisp*

Определить функцию, которая проверяет, является ли её аргумент списком из четырёх атомов.

Решение 3.188.1

```
(defun length4-atomp (w)
  (and (listp w)
       (= (length w) 4)
       (every (function atom) w)))

> (length4-atomp '(1 2 3 4))
T
> (length4-atomp '(1 2 3 (4)))
NIL
> (length4-atomp 4)
NIL
```

Решение 3.188.2

```
(defun length4-atomp (w)
  (and (listp w) (= (length w) 4) (every #'atom w)))

> (length4-atomp '(1 2 3 4))
T
> (length4-atomp '(1 2 3 (4)))
NIL
> (length4-atomp 4)
NIL
```


Структура 4 (функция (СПИСОК)) > (СПИСОК)

The newbie realizes that the difference between code and data is trivial. The expert realizes that all code is data. And the true master realizes that all data is code.

Sriram Krishnan «Lisp is sin»

Your age is the number of times you've orbited around the sun.

~

Задача 4.1 *our-last.lisp*

Определить функцию, возвращающую список из последнего элемента заданного списка.

Решение 4.1.1

```
(defun our-last (w)
  (cond ((null (cdr w)) w)
        ((our-last (cdr w)))))
```

```
> (our-last '(a b))
(B)
> (our-last '())
NIL
```

Решение 4.1.2

```
(defun our-last (w)
  (if (cdr w) (our-last (cdr w)) w))
```

```
> (our-last '(a b))
(B)
> (our-last '())
NIL
```

Задача 4.2 *odd-even.lisp*

Определить функцию перестановки местами соседних чётных и нечётных элементов в заданном списке.

Решение 4.2.1

```
(defun odd-even (w)
  (if (cdr w) (cons (cadr w) (cons (car w) (odd-even (cddr w)))) w))
```

```
> (odd-even '(a 1 a 1))
(1 A 1 A)
> (odd-even '(a 1 a 1 a))
(1 A 1 A A)
```

До тех пор – *if*, пока список *w* состоит хотя бы из двух элементов (*cadr w*) включаем – *cons* второй элемент (*cadr w*) списка *w* в начало списка, полученного в результате включения – *cons* первого элемента (*car w*) списка *w* в начало списка, полученного в результате вызова определяемой функции *odd-even* с хвостом хвоста (*cddr w*) списка *w*. Если условие *if* не выполнено, то есть в списке *w* меньше двух элементов, то возвращаем сам список *w* (или что от него осталось).

Задача 4.3 *age.lisp*

Написать функцию, которая принимает в качестве параметров: фамильное имя, год рождения и год смерти исторического деятеля и возвращает список из фамильного имени исторического деятеля и прожитых им лет.

Решение 4.3.1 *

```
(defun age (n b d)
  (list n (- d b)))

> (age 'bismarck 1815 1898)
(BISMARCK 83)
```

* Поскольку в условии не заданы даты рождения и смерти, то в отношении количества полных прожитых лет решение может содержать ошибку в один год.

Задача 4.4 *ages.lisp*

Написать функцию, которая принимает в качестве параметра список, состоящий из подсписков, содержащих фамильное имя, год рождения и год смерти исторического деятеля и возвращает список подсписков с фамильными именами исторических деятелей и числом прожитых ими лет.

Решение 4.4.1

```
(defun age (n b d)
  (list n (- d b)))

(defun ages (w)
  (mapcar #'(lambda (a) (apply #'age a)) w))

> (ages '((lincoln 1809 1865) (bismark 1815 1898)))
((LINCOLN 56) (BISMARCK 83))

> (ages '((cromwell 1599 1658) (napoleon 1769 1821)))
```

```
((CROMWELL 59) (NAPOLEON 52))
```

Задача 4.5 *w021.lisp*

В числовом списке каждый элемент равен 0, 1 или 2. Определите функцию, переставляющую элементы списка так, чтобы сначала располагались все нули, затем все двойки и, наконец, все единицы.

Решение 4.5.1

```
(defun w021 (w &optional a0 a2 a1 &aux (a (car w)) (d (cdr w)))
  (cond ((null w) (nconc a0 a2 a1))
        ((zerop a) (w021 d (cons a a0) a2 a1))
        ((= a 1) (w021 d a0 a2 (cons a a1)))
        ((w021 d a0 (cons a a2) a1))))

> (w021 '(1 0 2 1 2 0))
(0 0 2 2 1 1)
```

Комментарии (построчно):

1. Определяем функцию (**defun** от **define function**) с именем **w021**, в которую передается список **w** с необязательными (**&optional**) параметрами: переменными-аккумуляторами **a0**, **a2**, **a1** и вспомогательными (**&aux** - от **auxiliary** - вспомогательный) параметрами: голова **a** - значение вызова (**car w**) и хвост **d** - значение вызова (**cdr w**);

2. При условии **cond** (от слова **condition** - условие) если список **w** пустой, то возвращаем объединение **nconc** (можно вместо **nconc** применить **append**/дополнить) списков **a0 a2 a1**;

3. Функция **zerop** (от **zero** - ноль) проверяет, если первый элемент **a** списка **w** равен нулю, то вызываем функцию **w021** с хвостом **d**, списком из первого элемента **a**, присоединенного к списку-аккумулятору **a0** (функция **cons** включает новый элемент **a** в начало списка **a0**), и аккумуляторами **a2**, **a1**;

4. Функция **=** проверяет если первый элемент **a** списка **w** равен единице, то вызываем функцию **w021** с хвостом **d**, аккумуляторами **a0 a2** и списка из первого элемента **a**, присоединенного к списку-аккумулятору **a1** (функция **cons** включает новый элемент **a** в начало списка **a1**);

5. Если не выполнилось ни одно из трех вышеназванных условий, то вызывается функция **w021** с хвостом **d**, аккумулятором **a0**, списком из первого элемента **a**, присоединенного к списку-аккумулятору **a2** (функция **cons** включает новый элемент **a** в начало списка **a2**), и аккумулятором **a1**.

Задача 4.6 *list-min.lisp*

Написать функцию, ищущую минимум в заданном списке списков. Например: вход ((34 73 9 6) (5) (67 8 4 0)), выход (6 5 0). Использовать **mapcar** и **lambda**.

Решение 4.6.1

```
(defun list-min (w)
```

```
(mapcar #'(lambda (a) (reduce #'min a)) w))

> (list-min '((34 73 9 6) (5) (67 8 4 0)))
(6 5 0)
```

Задача 4.7 *del-third.lisp*

Another possibility I see is it might help in learning how to do data-driven programming, since it works by the principle of binding symbols to functions, and then evaluating lists of symbols, which means mapping some symbol references to those functions, and then executing that code.

Mark Miller

Написать функцию, которая удаляет из списка *w* каждый третий элемент (3,6,9,12 и так далее).

Решение 4.7.1

```
(defun del-third (w &aux (n 0))
  (mapcar #'(lambda (a) (incf n) (cond ((= n 3) (setq n 0) nil)
                                         ((list a))))
    w))

> (del-third '(1 2 3 4 5 6))
(1 2 4 5)
```

Решение 4.7.2

```
(defun del-third (w)
  (cond ((< (length w) 3) w)
        ((nconc (list (car w) (cadr w)) (del-third (nthcdr 3 w))))))

> (del-third '(1 2 3 4 5 6))
(1 2 4 5)
```

Решение 4.7.3

```
(defun del-third (w)
  (if (cddr w)
      (nconc (list (car w) (cadr w)) (del-third (nthcdr 3 w)))
      w))

> (del-third '(1 2 3 4 5 6))
(1 2 4 5)
```

Решение 4.7.4

```
(defun del-third (w)
```

```
(cond ((< (length w) 3) w)
      ((list* (car w) (cadr w) (del-third (nthcdr 3 w))))))

> (del-third '(1 2 3 4 5 6))
(1 2 4 5)
```

Решение 4.7.5

```
(defun del-third (w)
  (cond ((< (length w) 3) w)
        ((nconc (subseq w 0 2) (del-third (subseq w 3))))))

> (del-third '(1 2 3 4 5 6))
(1 2 4 5)
```

Решение 4.7.6

```
(defun del-third (w &aux (n 0))
  (delete-if #'(lambda (a) (incf n) (zerop (mod n 3))) w))

> (del-third '(1 2 3 4 5 6))
(1 2 4 5)
```

Решение 4.7.7

```
(defun del-third (w &aux (n 0))
  (delete 0 w :key #'(lambda (a) (incf n) (mod n 3))))

> (del-third '(1 2 3 4 5 6))
(1 2 4 5)
```

Решение 4.7.8

```
(defun del-third (w)
  (cond ((null w) nil)
        ((null (cdr w)) (cons (car w) nil))
        ((cons (car w) (cons (cadr w) (del-third (cdddr w)))))))

> (del-third '(1 2 3 4 5 6 7))
(1 2 4 5 7)
```

Решение 4.7.9

```
(defun del-third (w)
  (if (cddr w)
      (nconc (list (car w) (cadr w)) (del-third (nthcdr 3 w)))
      w))

> (del-third '(1 2 3 4 5 6 7))
(1 2 4 5 7)
```

Решение 4.7.10

```
(defun del-third (w)
  (if (cddr w) (list* (car w) (cadr w) (del-third (nthcdr 3 w))) w))

> (del-third '(1 2 3 4 5 6 7))
(1 2 4 5 7)
```

Решение 4.7.11

```
(defun del-third (w)
  (if (cddr w) (nconc (subseq w 0 2) (del-third (subseq w 3))) w))

> (del-third '(1 2 3 4 5 6 7))
(1 2 4 5 7)
```

Решение 4.7.12

```
(defun del-third (w)
  (reduce #'nconc
    (loop for a on w by #'cdddr
      if (cdr a) collect (list (car a) (cadr a))
      else collect (list (car a))))

> (del-third '(1 2 3 4 5 6 7))
(1 2 4 5 7)
```

Решение 4.7.13

```
(defun del-third (w)
  (reduce #'nconc
    (loop for a on w by #'cdddr
      if (cdr a) collect `',(car w) ,(cadr a))
      else collect `',(car a))))

> (del-third '(1 2 3 4 5 6 7))
(1 2 4 5 7)
```

Решение 4.7.14

```
(defun del-third (w)
  (loop for a on w by #'cdddr
    if (cdr a) nconc (list (car a) (cadr a))
    else nconc (list (car a))))

> (del-third '(1 2 3 4 5 6 7))
(1 2 4 5 7)
```

Решение 4.7.15

```
(defun del-third (w)
  (loop for a on w by #'cdddr
    if (cdr a) nconc `',(car w) ,(cadr a))
    else nconc `',(car a))))
```

```
> (del-third '(1 2 3 4 5 6 7))
(1 2 4 5 7)
```

Решение 4.7.16

```
(defun del-third (w)
  (loop for a in w for b from 1
        unless (zerop (rem b 3)) collect a))

> (del-third '(1 2 3 4 5 6 7))
(1 2 4 5 7)
```

Задача 4.8 order-elms.lisp

Проверить список, если в нем есть одинаковые элементы, то выстроить их подряд.

Решение 4.8.1

```
(defun order-elms (w)
  (cond ((= (length (delete-duplicates w)) (length w)) w)
        ((numberp (car w)) (sort w #'<))
        ((sort w #'string<))))

> (order-elms '(1 2 1))
(1 1 2)
> (order-elms '(a b a))
(A A B)
```

Задача 4.9 del-last3.lisp

Удалить из списка три последних элемента. Опишите функцию на примере списка (1 2 3 4 5).

Решение 4.9.1

```
(defun del-last3 (w)
  (delete-if #'atom w :count 3 :from-end t))

> (del-last3 '(0 1 2 3))
(0)
```

Решение 4.9.2

```
(defun del-last3 (w)
  (delete-if (constantly t) w :count 3 :from-end t))

> (del-last3 '(0 1 2 3))
(0)
```

Решение 4.9.3

```
(defun del-last3 (w)
  (delete-if #'identity w :count 3 :from-end t))

> (del-last3 '(0 1 2 3))
(0)
```

Решение 4.9.4

```
(defun del-last3 (w)
  (ldiff w (last w 3)))

> (del-last3 '(0 1 2 3))
(0)
```

Решение 4.9.5

```
(defun del-last3 (w)
  (nbutlast w 3))

> (del '(0 1 2 3))
(0)
```

Задача 4.10 reverse-els.lisp

Написать функцию, примером работы которой будет:

```
> (f '(a b c))
(c b a)
```

Решение 4.10.1

```
(defun reverse-els (w &optional acc)
  (cond ((null w) acc)
        ((reverse-els (cdr w) (cons (car w) acc)))))

> (reverse-els '(a b c))
(C B A)
```

`&optional` - указывает на то, что далее следует необязательный (опциональный) параметр;
`acc` - переменная, накапливает результат (аккумулятор) и возвращает результат из второй строки.

P.S. (без спецификатора `&optional` пришлось бы явно задавать второй параметр - `nil`)

Решение 4.10.2

```
(defun reverse-els (w acc)
  (cond ((null w) acc)
        ((reverse-els (cdr w) (cons (car w) acc)))))
```



```
> (reverse-els '(a b c) nil)
(C B A)
```

Решение 4.10.3

```
(defun reverse-els (w &optional acc)
  (if w (reverse-els (cdr w) (cons (car w) acc)) acc))
```

```
> (reverse-els '(a b c) nil)
(C B A)
```

Решение 4.10.4

```
(defun reverse-els (w)
  (cond ((null w) w)
        ((null (cdr w)) w)
        ((cons (car (reverse-els (cdr w)))
                 (reverse-els
                  (cons (car w)
                        (reverse-els
                         (cdr (reverse-els (cdr w))))))))))
```

```
> (reverse-els '(a b c) nil)
(C B A)
```

Решение 4.10.5

```
(defun reverse-els (w)
  (when w
    (if (cdr w)
        (cons (car (reverse-els (cdr w)))
              (reverse-els
               (cons (car w)
                     (reverse-els
                      (cdr (reverse-els (cdr w)))))))
        w)))
```

```
> (reverse-els '(a b c) nil)
(C B A)
```

Задача 4.11 maxub.lisp

Построить функцию, которая заданному списку выдает подсписок, в котором больше всего элементов: (d '(a (b) (c (d)))) > (c (d)).

Решение 4.11.1

```
(defun maxub (w &optional m)
  (cond ((null w) m)
        ((and (listp (car w))
              (> (length (car w))
```

```

      (length m))
      (maxub (cdr w) (car w))))
  ((maxub (cdr w) m)))

> (maxub '(a (b) (c (d))))
(C (D))

```

Решение 4.11.2

```

(defun maxub (w &optional m)
  (if w ((lambda (a)
            (if (and (listp a) (> (length a) (length m)))
                (maxub (cdr w) a)
                (maxub (cdr w) m)))
        (car w)
        m))

> (maxub '(a (b) (c (d))))
(C (D))

```

Задача 4.12 group3.lisp

Написать функцию преобразования списка: (a b c d e f) -> ((a b c) (d e f)) с учетом того что количество троек элементов может быть любым. Элементы списков могут быть любыми объектами.

Решение 4.12.1

```

(defun group3 (w)
  (cond ((caddr w) (cons (list (car w) (cadr w) (caddr w))
                          (group3 (cddddr w))))
        (w (cons w nil))))

> (group3 '(a b c d e f))
((A B C) (D E F))

```

Решение 4.12.2

```

(defun group3 (w)
  (loop for v on w by #'cddddr collect
        (if (caddr v) (subseq v 0 3) (subseq v 0))))

> (group3 '(a b c d e f))
((A B C) (D E F))

```

Решение 4.12.3*

* Нет ни у кого мысли как сделать более высокоскоростной вариант без рекурсии?

```

(defun group3 (w &aux acv acw)
  (dolist (a w)

```

```

(push a acv)
(when (caddr acv)
  (push (nreverse acv) acw)
  (setq acv nil)))
(when (> (length acv) 0) (push (nreverse acv) acw))
(nreverse acw))

> (group3 '(a b c d e f))
((A B C) (D E F))

```

Решение 4.12.4*

```

(defparameter *a* nil)

(defun pack ()
  (dotimes (i 20000)
    (push "a" *a*)))

(defmacro while (test &rest body)
  `(do ()
    ((not ,test))
    ,@body))

(defun group3 (w acc)
  (while (caddr w)
    (setq acc (append acc (list (list (car w) (cadr w) (caddr w)))))
    (setq w (cdddr w)))
  (if (car w) (setq acc (append acc (list w))))
  acc)

> (pack)
NIL
> (time (group3 *a* nil))
Real time: 2.7221558 sec.
Run time: 2.6988173 sec.
Space: 177982208 Bytes
GC: 339, GC time: 1.8876121 sec.
(("a" "a" "a") ("a" "a" "a") ("a" "a" "a") ...)

```

Решение 4.12.5*

```

(defparameter *a* nil)

(defun pack ()
  (dotimes (i 20000)
    (push "a" *a*)))

(defmacro while (test &rest body)
  `(do ()
    ((not ,test))
    ,@body))

(defun group3 (w acc)

```

```

(while (caddr w)
  (setq acc (nconc acc (list (list (car w) (cadr w) (caddr w)))))
  (setq w (cddddr w)))
(if (car w) (setq acc (nconc acc (list w))))
acc)

> (pack)
NIL
> (time (group3 *a* nil))
Real time: 0.157009 sec.
Run time: 0.156001 sec.
Space: 213320 Bytes
(("a" "a" "a") ("a" "a" "a") ("a" "a" "a") ...)

```

Решение 4.12.6

```

(defun group3 (w)
  (cond ((caddr w) (cons `((, (car w) , (cadr w) , (caddr w))
                          (group3 (cddddr w))))
        (w (cons w nil)))))

> (group3 '(a b c d e f))
((A B C) (D E F))

```

Задача 4.13 list-company.lisp

Прайс компьютерного магазина описывается списком (наименование фирма параметры цена): ((Терминал Acer 15 4500) (Терминал Sumsung 17 7000) (Матплата Asus 238 2500) (Процессор Intel 2.5 3000)). Определить функцию возвращающую список фирм.

Решение 4.13.1

```

(defun list-company (w)
  (remove-duplicates (mapcar #'cadr w)))

> (list-company '((terminal acer 15 4500) (terminal samsung 17 7000)
(motherboard intel 238 2500) (processor intel 2.5 3000)))
(ACER SAMSUNG INTEL)

```

Решение 4.13.2

```

(defun list-company (w)
  (remove-duplicates
   (loop for (name firm parameters price) in w
         collect firm)))

> (list-company '((terminal acer 15 4500) (terminal samsung 17 7000)
(motherboard intel 238 2500) (processor intel 2.5 3000)))
(ACER SAMSUNG INTEL)

```

Решение 4.13.3

```
(defun list-company (w)
  (remove-duplicates
   (loop for a in w collect (cadr a))))

> (list-company '((terminal acer 15 4500) (terminal samsung 17 7000)
  (motherboard intel 238 2500) (processor intel 2.5 3000)))
(ACER SAMSUNG INTEL)
```

Задача 4.14 *twice-atom.lisp*

Поставлена задача удвоения всех атомов в подписках любого уровня, т.е. из такого (2(3 5(7 8(2 6)))) получить (22(33 55(77 88(22 66)))).

Решение 4.14.1

```
(defun twice-atom (w &aux (a (car w)) (d (cdr w)))
  (cond ((null w) nil)
        ((atom a) (cons a (cons a (twice-atom d))))
        ((cons (twice-atom a) (twice-atom d)))))

> (twice-atom '(2 (3 5 (7 8 (2 6)))))
(2 2 (3 3 5 5 (7 7 8 8 (2 2 6 6))))
```

Решение 4.14.2

```
(defun twice-atom (w &aux (a (car w)) (d (cdr w)))
  (when w (if (atom a)
              (cons a (cons a (twice-atom d)))
              (cons (twins a) (twice-atom d)))))

> (twice-atom '(2 (3 5 (7 8 (2 6)))))
(2 2 (3 3 5 5 (7 7 8 8 (2 2 6 6))))
```

Задача 4.15 *vowels-del.lisp*

Задан список символов произвольного уровня вложенности, следует сформировать из него список, состоящий только из гласных букв. Вычислить число удаленных символов.

Решение 4.15.1

```
(defun vp (s) (find s '(a e i o u)))

(defun vs (w &optional acc (n 0) &aux (a (car w)) (d (cdr w)))
  (cond ((null w) (list (reverse acc) n))
        ((vp a) (vs d (push a acc) n))
        ((vs d acc (1+ n)))))

(defun ft (w &optional acc)
  (cond ((null w) acc)
```

```
((atom w) (cons w acc))
((ft (car w) (ft (cdr w) acc))))))
```

```
(defun vowels-del (w) (vs (ft w)))
```

```
> (vowels-del '((a c e) u))
((A E U) 1)
```

* Что делает каждая функция?

vp - проверяет, является ли символ гласной (vp - vowel p).

vs - возвращает список гласных символов и число удаленных символов (vs - vowel symbol).

ft - выравнивает список в одноуровневый, плоский (ft - flat).

vowels-del - главная, вызывает ft и vs.

** Что означают слова &optional, асс, &aux, push, acc?

&optional - в списке параметров предваряет необязательный (опциональный) параметр.

&aux - в списке параметров предваряет дополнительные (вспомогательные) переменные.

push - функция, заталкивает значение переменной а в асс - (push а асс).

асс - переменная - список, накапливает результат (аккумулятор).

Решение 4.15.2

```
(defun vp (s) (find s '(a e i o u)))
```

```
(defun vs (w &optional acc (n 0) &aux (a (car w)) (d (cdr w)))
  (cond ((null w) (list (reverse acc) n))
        ((vp a) (vs d (push a acc) n))
        ((vs d acc (1+ n)))))
```

```
(defun ft (w)
  (loop for a in w
        if (and a (atom a)) collect a
        else nconc (ft a)))
```

```
(defun vowels-del (w) (vs (ft w)))
```

```
> (vowels-del '((a c e) u))
((A E U) 1)
```

Задача 4.16 *lets&nums.lisp*

Определить функцию, которая по списку из букв и цифр (a b 1 2 c) строит два списка, один из букв, другой из цифр, отсортированных по возрастанию: (A B C) (1 2).

Решение 4.16.1

```
(defun lets&nums (w &optional ac acn &aux (a (car w)))
  (cond ((null w) (list (sort ac #'string<) (sort acn #'<))))
        ((numberp a) (lets&nums (cdr w) ac (push a acn)))
        ((lets&nums (cdr w) (push a ac) acn))))
```

```
> (lets&nums '(a b 1 2 c))
((A B C) (1 2))
```

Решение 4.16.2*

* При попытке выполнить функцию с заданными параметрами, у меня XLISP ругается: "error: unbound function - PUSH".

```
(defun lets&nums (w &optional ac acn &aux (a (car w)))
  (cond ((null w) (list (sort ac #'string<) (sort acn #'<))))
        ((numberp a) (lets&nums (cdr w) ac (push a acn)))
        ((lets&nums (cdr w) (push a ac) acn))))
```

```
(defun push (a w)
  (setf w (cons a w)))
```

```
> (lets&nums '(a b 1 2 c))
((A B C) (1 2))
```

Задача 4.17 *nums-to-words.lisp*

Написать функцию, которая переводит список цифр от 0 до 9 в список соответствующих им названий (строк).

Решение 4.17.1

```
(defun nums-to-words (w)
  (mapcar
   #'(lambda (a)
       (case a
         (0 "zero")
         (1 "one")
         (2 "two")
         (3 "three")
         (4 "four")
         (5 "five")
         (6 "six")
         (7 "seven")
         (8 "eight")
         (9 "nine")
         (t unknown))))
   w))
```

```
> (nums-to-words '(1 2 3))
("one" "two" "three")
```

Решение 4.17.2

```
(defun nums-to-words (w)
  (mapcar #'(lambda (a) (format nil "~r" a))
    w))

> (nums-to-words '(1 2 3))
("one" "two" "three")
```

Решение 4.17.3

```
(defun nums-to-words (w)
  (loop for a in w
        collect (format nil "~r" a)))

> (nums-to-words '(1 2 3))
("one" "two" "three")
```

Задача 4.18 sec-blast-num.lisp

Если второй и предпоследний элементы списка - числовые атомы, то функция возвращает список, состоящий из этих элементов.

Решение 4.18.1

```
(defun sec-blast-num (w)
  (when (every #'numberp (list (cadr w) (car (last w 2))))
    (list (cadr w) (car (last w 2)))))

> (sec-blast-num '(a 1 b 2 c))
(1 2)
> (sec-blast-num '(a 1 b 2))
NIL
```

Задача 4.19 car-encl-list.lisp

Если первый элемент списка содержит списки, то функция возвращает этот элемент.

Решение 4.19.1

```
(defun car-encl-list (w)
  (when (and (listp (car w))
    (some #'listp (car w))) (car w)))

> (car-encl-list '(((a) (b)) c))
((A) (B))
> (car-encl-list '((a) (b) c))
NIL
```


Задача 4.20 arithmetic-to-text.lisp

I will tell you a dark, little known secret about Common Lisp (my favourite language): you can use it productively without becoming "a better programmer". That's what I do. I caught on to Common Lisp because of Peter Seibel's book Practical Common Lisp. Then I studied Peter Norvig's AI book that uses Lisp. I use Lisp and C almost exclusively (along with R and octave in their respective domains). As much as I love to program, I must admit that I am no superstar. I would even say that I am a rather mediocre Lisp programmer. For example, I don't understand Lisp macros very well. My haskell experience taught me to use functional methods when I can. But I have no qualms about using CLOS (Common Lisp Object System), mutable state, and any other quick and dirty trick hodgepodge that would probably be scorned by more talented programmers. Why is this? I can do prototyping, thought experiments, and proof of concept stuff in Common Lisp way quicker than I can in other languages. I truly don't know why this is, though. And then, when I go back to make production quality stuff, that seems to work out too. I use Qt Creator and Xcode for C and C++ development, and I think both are great. But my normal way of exploring and finding solutions to a software problem is with emacs, slime, and Common Lisp. I have heard that I might get the same bang for the buck from Python. But, I don't see the point. Sometimes I think it's like taste in music, or food. Maybe a particular language just resonates with a person more than other languages.

Thomas Penton

Преобразовать математические термины, содержащиеся в произвольном тексте в символьное выражение. В качестве результата выдать весь текст с формулами. Текст может содержать следующие мат. термины: "существует", "для любого", "сумма", "разность", "произведение", латинские буквы X, Y, Z. Заменить все эти слова на соответствующие знаки + - *.

Решение 4.20.1

```
(defun arithmetic-to-text (w)
  (mapcar
    #'(lambda (a)
```

```

      (case a
        (sum '+)
        (product '*)
        (x 'x)
        (z 'z)
        (t "unknown"))))
  w))

> (tr '(product x z)
  (* X Z))

```

Решение 4.20.2*

* Возникло пару вопросов... что означает #' и если встречается слово не входящее в "словарь" как сделать чтобы оно оставалось на месте? например:

```

(arithmetic-to-text `(x sum z ravno z product x))
(x + z ravno z * x)

```

#' является сокращенной записью function (как ' - сокращенной записью quote)

HyperSpec: 2.4.8.2 Sharpsign Single-Quote

Any expression preceded by #' (sharpsign followed by single-quote), as in #'expression, is treated by the Lisp reader as an abbreviation for and parsed identically to the expression (function expression). See function. For example,

```

(apply #'+ 1) == (apply (function +) 1)

```

```

(defun arithmetic-to-text (w)
  (mapcar #'(lambda (a) (case a (sum '+) (product '*) (t a))) w))

> (arithmetic-to-text `(x sum z ravno z product x))
(X + Z RAVNO Z * X)

```

Задача 4.21 parse-pascal.lisp

Написать программу, которая преобразует цикл for произвольной программы на языке Паскаль в соответствие инструкции до языка Лисп (считать что в теле цикла один оператор, выполняющий арифметическое действие, сложение или вычитание).

Решение 4.21.1

```

(defun parse-pascal
  (w &aux (a (cadr w)) (j (eq (nth 4 w) 'to)) (f (nth 7 w)))
  `(do ((,a , (nth 3 w) ), (if j '1+ '1-), a))
    ((, (if j '> '<) ,a , (nth 5 w)), f)
    (setf ,f ((nth 10 w) , (nth 9 w) , (nth 11 w)))))

> (parse-pascal '(for i = 1 to 10 do k := k + i))

```

```
(DO ((I 1 (1+ I))) (> I 10) K) (SETF K (+ K I)))
```

Функция `pars-pascal` заполняет `do`-форму из `for`-списка. Пример того, как работают в связке ``` - `backquote` и `,` - `comma`:

```
>`((+ 2 3) ,(+ 2 3))
((+ 2 3) 5)
```

1. Commas должны быть вложены в `backquoted`-список, поэтому в примере `>`((+ 2 3) ,(+ 2 3))` выдаст ошибку: `READ: comma is illegal outside of backquote [Condition of type SYSTEM::SIMPLE-READER-ERROR]`;

2. Например: ``(do ((,a , (nth 3 w) ... ,` где `do` - необрабатываемый символ, не имя функции `/` , `a` - код, переменная - второй элемент списка `'w'` , `(nth 3 w)` - код, функция - подставляет четвертый элемент списка `'w'`; для `backquote` и `comma` можно применить метафору обратного переключателя или кнопки: `backquote` - положение 'вверх' (выключено - кнопка отжата) - режим данных `*`, `comma` - положение 'вниз' - (включено - кнопка утоплена) - режим кода.

* Conrad Barski, M.D. - автор идеи 'data mode' и 'code mode': www.lisperati.com/syntax.html

Задача 4.22 *arabic-to-roman.lisp*

Написать простой транслятор способный переводить арабские цифры в римские в диапазоне от 0 до 100.

Решение 4.22.1

```
(defun arabic-to-roman (w)
  (mapcar #'(lambda (a) (format nil "~@r" a)) w))

(arabic-to-roman '(123 456 789))
("CXXIII" "CDLVI" "DCCLXXXIX")
```

Решение 4.22.2

```
(defun arabic-to-roman (w)
  (mapcar #'(lambda (a) (intern (format nil "~@r" a))) w))

(arabic-to-roman '(123 456 789))
(CXXIII CDLVI DCCLXXXIX)
```

Задача 4.23 *reverse-list.lisp*

Написать функцию, которая обращает список. В качестве последнего элемента списка аргумента допускается использовать NTL. Пример: `(a b c) > (c b a)`, `(a(b)c(d)) > ((d)c(b)a)`.

Решение 4.23.1

```
(defun reverse-list (w)
  (reverse w))

(reverse-list '(a(b)c(d)))
((D) C (B) A)
```

... представить аналогичную рекурсивную функцию.

Решение 4.23.2

```
(defun reverse-list (w &optional ac)
  (cond ((null w) ac)
        ((reverse-list (cdr w) (cons (car w) ac)))))

(reverse-list '(a b c))
(C B A)
```

Задача 4.24 *liat.lisp*

The key to performance is elegance, not bat-
talions of special cases.

Jon Bentley and Doug McIlroy

Определите функцию, которая для заданного списка *w* формирует список-результат путем объединения результата реверсирования *w*, результата реверсирования хвоста *w*, результата реверсирования хвоста хвоста *w* и так далее. Пример: для списка (1 2 3 4 5 6) результатом будет: (6 5 4 3 2 1 6 5 4 3 2 6 5 4 3 6 5 4 6 5 6).

Решение 4.24.1

```
(defun liat (w)
  (cond ((null w) nil)
        ((append (reverse w) (liat (cdr w))))))

> (liat '(1 2 3 4 5 6))
(6 5 4 3 2 1 6 5 4 3 2 6 5 4 3 6 5 4 6 5 6)
```

Решение 4.24.2

```
(defun liat (w)
  (cond ((null w) nil)
        ((nconc (reverse w) (liat (cdr w))))))

> (liat '(1 2 3 4 5 6))
(6 5 4 3 2 1 6 5 4 3 2 6 5 4 3 6 5 4 6 5 6)
```

Решение 4.24.3

```
(defun liat (w)
```

```
(when w (nconc (reverse w) (liat (cdr w))))

> (liat '(1 2 3 4 5 6))
(6 5 4 3 2 1 6 5 4 3 2 6 5 4 3 6 5 4 6 5 6)
```

Решение 4.24.4

```
(defun liat (w)
  (loop for d on w nconc (reverse d)))

> (liat '(1 2 3 4 5 6))
(6 5 4 3 2 1 6 5 4 3 2 6 5 4 3 6 5 4 6 5 6)
```

Решение 4.24.5

```
(defun liat (w)
  (mapcon #'reverse w))

> (liat '(1 2 3 4 5 6))
(6 5 4 3 2 1 6 5 4 3 2 6 5 4 3 6 5 4 6 5 6)
```

Задача 4.25 reverse-subst.lisp

Реализовать функцию, которая в начальном списке изменяет все элементы-списки результатами их реверсирования. Реверсирование делать на всех уровнях вложения. Пример: Для списка (1((2 3)4) 5 6) результатом будет (1(4(3 2))5 6).

Решение 4.25.1

```
(defun reverse-subst (w)
  (cond ((null w) nil)
        ((atom (car w)) (cons (car w) (reverse-subst (cdr w))))
        ((cons (reverse-subst (reverse (car w)))
                 (reverse-subst (cdr w))))))

> (reverse-subst '(1((2 3)4) 5 6))
(1 (4 (3 2)) 5 6)
```

Решение 4.25.2

```
(defun reverse-subst (w)
  (when w (if (atom (car w))
               (cons (car w) (reverse-subst (cdr w)))
               (cons (reverse-subst (reverse (car w)))
                       (reverse-subst (cdr w))))))

> (reverse-subst '(1 ((2 3) 4) 5 6))
(1 (4 (3 2)) 5 6)
```

Решение 4.25.3

```
(defun reverse-subs (w)
  (loop for a in w
        collect (if (atom a) a (reverse-subs (reverse a)))))
```

```
> (reverse-subs '(1((2 3)4) 5 6))
(1 (4 (3 2)) 5 6)
```

Решение 4.25.4

```
(defun reverse-subs (w)
  (mapcar #'(lambda (a)
              (if (atom a) a (reverse-subs (reverse a)))) w))
```

```
> (reverse-subs '(1((2 3)4) 5 6))
(1 (4 (3 2)) 5 6)
```

Задача 4.26 num-list-atom.lisp

На входе список. Функция сортирует список на три подсписка:

1. Числа.
2. Списки.
3. Атомы.

Решение 4.26.1.

```
(defun num-list-atom (w &optional n c s &aux (a (car w)))
  (cond ((null w) (list n c s))
        ((cons a) (num-list-atom (cdr w) n (push a c) s))
        ((numberp a) (num-list-atom (cdr w) (push a n) c s))
        ((num-list-atom (cdr w) n c (push a s)))))
```

```
> (num-list-atom '(a 1 (b c) 2 d))
((2 1) ((B C)) (D A))
```

Задача 4.27 del-multy.lisp

Создайте функцию, удаляющую в исходном списке комбинации из двух и более повторяющихся элементов.

Решение 4.27.1

```
(defun del-multy (w &optional b)
  (cond ((null w) nil)
        ((or (equal (car w) b) (equal (car w) (cadr w)))
         (del-multy (cdr w) (car w)))
        ((cons (car w) (del-multy (cdr w) (car w)))))
```

```
> (del-multy '(a b b c b b b c c d d))
(A C)
```

Задача 4.28 even-2-3.lisp

Дан список чисел. Написать функцию, возвращающую в случае первого четного элемента исходный список, в котором первые три числа возведены в квадрат, иначе – исходный список, в котором первые три числа возведены в куб.

Решение 4.28.1

```
(defun even-2-3 (w)
  (if (evenp (car w))
      (cons (expt (car w) 2)
            (cons (expt (cadr w) 2)
                  (cons (expt (caddr w) 2) (cddddr w))))
      (cons (expt (car w) 3)
            (cons (expt (cadr w) 3)
                  (cons (expt (caddr w) 3) (cddddr w))))))

> (even-2-3 '(4 5 6 7 8 9))
(16 25 36 7 8 9)
```

Задача 4.29 atbash.lisp

Имеется некоторая последовательность символов, зашифровать ее шифром atbash.

Решение 4.29.1

```
(defun atbash (w)
  (mapcar
   #'(lambda (az)
       (case az
         (a z)
         (b y)
         (c x)
         (d w)
         (e v)
         (f u)
         (g t)
         (h s)
         (i r)
         (j q)
         (k p)
         (l o)
         (m n)
         (n m)
         (o l)
         (p k)
         (q j)
         (r i)
         (s h)
         (t g))
      ))
   w))
```

```

      (u f)
      (v e)
      (w d)
      (x c)
      (y b)
      (z a)))
  w))

> (atbash '(e x x e g o e x s r g i))
(V C C V T L V C H I T R)

```

Решение 4.29.2

```

(defun atbash
  (w &aux
    (x '(a b c d e f g h i j k l m n o p q r s t u v w x y z))
    (z '(z y x w v u t s r q p o n m l k j i h g f e d c b a)))
  (when w (cons (nth (position (car w) x) z) (atbash (cdr w)))))

> (atbash '(e x x e g o e x s r g i))
(V C C V T L V C H I T R)

```

Решение 4.29.3

```

(defun atbash
  (w &aux
    (x '(a b c d e f g h i j k l m n o p q r s t u v w x y z))
    (z '(z y x w v u t s r q p o n m l k j i h g f e d c b a)))
  (loop for a in w collect (nth (position a x) z)))

> (atbash '(e x x e g o e x s r g i))
(V C C V T L V C H I T R)

```

Решение 4.29.4

```

(defun atbash
  (w &aux
    (x '(a b c d e f g h i j k l m n o p q r s t u v w x y z))
    (z '(z y x w v u t s r q p o n m l k j i h g f e d c b a)))
  (mapcar #'(lambda (a) (nth (position a x) z)) w))

> (atbash '(e x x e g o e x s r g i))
(V C C V T L V C H I T R)

```

Задача 4.30 sine-twins.lisp

Определить функцию, которая удаляет из многоуровневого списка повторяющиеся элементы на верхнем уровне.

Решение 4.30.1

```
;-member (повторяет функцию Лиспа member) проверяет наличие элемента в
```


;списке, но возвращает не сам элемент, не позицию, а ту часть списка, которая начинается с этого элемента, то есть положительное значение, поскольку все, что не nil (ложь, пустой список) - все t (истина).

```
(defun -member (a w)
  (cond ((null w) nil)
        ((equalp a (car w)) (cons (car w) (cdr w)))
        ((-member a (cdr w)))))

(defun sine-twins (w)
  (cond ((null w) nil)
        ((-member (car w) (cdr w)) (sine-twins (cdr w)))
        ((cons (car w) (sine-twins (cdr w)))))

> (sine-twins '((a b) c d (a b) c e))
(D (A B) C E)
```

Решение 4.30.2

```
(defun sine-twins (w)
  (when w (if (member (car w) (cdr w) :test #'equalp)
              (sine-twins (cdr w))
              (cons (car w) (sine-twins (cdr w))))))

> (sine-twins '((a b) c d (a b) c e))
(D (A B) C E)
```

Решение 4.30.3

```
(defun sine-twins (w &optional acc)
  (cond ((null w) (reverse acc))
        ((member (car w) acc :test #'equalp)
         (sine-twins (cdr w) acc))
        ((sine-twins (cdr w) (cons (car w) acc)))))

> (sine-twins '((a b) c d (a b) c e))
((A B) C D E)
```

Решение 4.30.4

```
(defun sine-twins (w)
  (remove-duplicates w :test #'equalp))

> (sine-twins '((a b) c d (a b) c e))
(D (A B) C E)
```

Задача 4.31 conc-asterisc.lisp

Дан числовой список. Определить функцию, присоединяющую к началу каждого отрицательного элемента символ *.

Решение 4.31.1

```
(defun conc (n)
  (intern (concatenate 'string "*" (write-to-string n))))

(defun conc-asterisc (w)
  (cond ((null w) nil)
        ((minusp (car w))
         (cons (conc (car w)) (conc-asterisc (cdr w))))
        ((cons (car w) (conc-asterisc (cdr w)))))

> (conc-asterisc '(-1 0 1))
(*-1 0 1)
```

Задача 4.32 wordy.lisp

Дана строка символов. Определить, есть ли повторение слов в этой строке. На экран вывести те слова, которые встречаются в строке больше одного раза.

Решение 4.32.1

```
(defun string-list (s)
  (read-from-string
   (concatenate
    'string "(" (delete-if-not #'(lambda (x)
                                   (or (alpha-char-p x)
                                       (equal x #\space)
                                       (equal x #\-))))
            s) ")"))))

(defun word-amount (w)
  (when w (cons (list (string-downcase (car w)) (count (car w) w))
                (word-amount (delete (car w) (cdr w))))))

(defun wordy (s)
  (mapcar #'car (remove-if
                #'(lambda (a) (= a 1))
                (word-amount (string-list s)) :key #'cadr)))

> (wordy "aaa aaa bbb")
("aaa")
```

Задача 4.33 sum-matrixx.lisp

Определите функцию, которая вычисляет сумму элементов каждой строки матрицы, где матрица представлена в виде списка, каждая строка матрицы – подсписок.

Решение 4.33.1

```
(defun sum-matrixx (w)
```

```

    (when w (cons (apply #'(lambda (a) (sum-matrixx (cdr w)))
                    (sum-matrixx (cdr w)))))

> (sum-matrixx '((1 3) (3 7)))
(4 10)

```

Решение 4.33.2

```

(defun sum-matrixx (w)
  (mapcar #'(lambda (a) (apply #'(lambda (w)
                                   (sum-matrixx (cdr w)))
                                   a))
          w))

> (sum-matrixx '((1 3) (3 7)))
(4 10)

```

Решение 4.33.3

```

(defun sum-matrix (w)
  (loop for a in w
        collect (apply #'(lambda (w)
                            (sum-matrixx (cdr w)))
                        a)))

> (sum-matrixx '((1 3) (3 7)))
(4 10)

```

Задача 4.34 area.lisp

Определите функцию вычисления площади геометрических фигур и поверхностей.

Решение 4.34.1

```

(defconstant square 1)
(defconstant hexagon (* (sqrt 3) 1.5))
(defconstant circle pi)
(defconstant sphere (* 4 pi))
(defconstant torus (* 4 pi pi))

(defun area (shape r)
  (* shape r r))

> (area square 5)
25
> (area hexagon 5)
64.951904
> (area circle 5)
78.53981633974483096L0
> (area sphere 5)
314.15926535897932384L0
> (area torus 5)
986.96044010893586196L0

```

Задача 4.35 *clear-twins.lisp*

Заданы три списка. Удалить из первого списка все элементы остальных списков.

Решение 4.35.1

```
(defun clear-twins (w x z &aux (a (car w)) (d (cdr w)))
  (cond ((null w) nil)
        ((or (member a x) (member a z)) (clear-twins d x z))
        ((cons a (clear-twins d x z)))))
```

```
> (clear-twins '(a b c) '(b) '(c))
(A)
```

Решение 4.35.2

```
(defun clear-twins (w x z) (set-difference (set-difference w x) z))
```

```
> (clear-twins '(a b c) '(b) '(c))
(A)
```

Решение 4.35.3

```
(defun clear-twins (w x z) (set-difference w (union x z)))
```

```
> (clear-twins '(a b c) '(b) '(c))
(A)
```

Задача 4.36 *fold-n-m.lisp*

Дан числовой одноуровневый список. Определить функцию, которая возвращает числа кратные двум заданным.

Решение 4.36.1

```
(defun fold-n-m (w n m)
  (cond ((null w) nil)
        ((= 0 (mod (car w) n) (mod (car w) m))
         (cons (car w) (fold-n-m (cdr w) n m)))
        ((fold-n-m (cdr w) n m))))
```

```
> (fold-n-m '(1 2 3 4 5 6 7 8 9 10 11 12) 2 3)
(6 12)
```

Задача 4.37 *swing.lisp*

Сгруппировать элементы списка в подсписки следуя закономерности по 1, 2, 3, 2 элемента.

Решение 4.37.1

```
(defun swing (w m &optional (n 1) (up t))
  (cond ((null (nthcdr n w)) (cons w nil))
        ((and (< n m) up) (cons (subseq w 0 n)
                                (swing (nthcdr n w) m (1+ n) t)))
        ((= n m) (cons (subseq w 0 n)
                        (swing (nthcdr n w) m (1- n) nil)))
        ((= n 1) (cons (subseq w 0 n)
                        (swing (nthcdr n w) m (1+ n) t)))
        ((cons (subseq w 0 n) (swing (nthcdr n w) m (1- n) nil)))))

> (swing '(1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16) 3)
((1) (2 3) (4 5 6) (7 8) (9) (10 11) (12 13 14) (15 16))
```

Задача 4.38 map-sum.lisp

Дан список многоуровневых подсписков. Необходимо построить список сумм каждого подсписка верхнего уровня.

Решение 4.38.1

```
(defun flat (w &optional acc)
  (cond ((null w) acc)
        ((atom w) (cons w acc))
        ((flat (car w) (flat (cdr w) acc)))))

(defun map-sum (w)
  (mapcar #'(lambda (a) (apply #'+ a)) (mapcar #'flat w)))

> (map-sum '(((1 2 3) (4 5 6)) ((7 8 9))))
(21 24)
```

Решение 4.38.2

```
(defun flat (w &optional acc)
  (cond ((null w) acc)
        ((atom w) (cons w acc))
        ((flat (car w) (flat (cdr w) acc)))))

(defun map-sum (w)
  (cond ((null w) nil)
        ((cons (apply #'+ (flat (car w)))
                 (map-sum (cdr w)))))

> (map-sum '(((1 2 3) (4 5 6)) ((7 8 9))))
(21 24)
```

Решение 4.38.3

```
(defun flat (w &optional acc)
  (cond ((null w) acc)
        ((atom w) (cons w acc))
```

```

      ((flat (car w) (flat (cdr w) acc))))))

(defun map-sum (w)
  (if (null w) nil (cons (apply #'+ (flat (car w)))
                        (map-sum (cdr w)))))

> (map-sum '(((1 2 3) (4 5 6)) ((7 8 9))))
(21 24)

```

Решение 4.38.4

```

(defun flat (w &optional acc)
  (cond ((null w) acc)
        ((atom w) (cons w acc))
        ((flat (car w) (flat (cdr w) acc)))))

(defun map-sum (w)
  (when w (cons (apply #'+ (flat (car w))) (map-sum (cdr w)))))

> (map-sum '(((1 2 3) (4 5 6)) ((7 8 9))))
(21 24)

```

Решение 4.38.5

```

(defun flat (w &optional acc)
  (cond ((null w) acc)
        ((atom w) (cons w acc))
        ((flat (car w) (flat (cdr w) acc)))))

(defun add-numbers (w)
  (cond ((null w) 0)
        ((+ (car w) (add-numbers (cdr w)))))

(defun map-sum (w)
  (cond ((null w) nil)
        ((cons (add-numbers (flat (car w)))
                (map-sum (cdr w)))))

> (map-sum '(((1 2 3) (4 5 6)) ((7 8 9))))
(21 24)

```

Решение 4.38.6

```

(defun flat (w)
  (loop for a in w
        if (listp a) append (flat a)
        else collect a))

(defun add-numbers (w)
  (if (null w) 0 (+ (car w) (add-numbers (cdr w)))))

(defun map-sum (w)
  (when w (cons (add-numbers (flat (car w)))
                (map-sum (cdr w)))))

```

```
(map-sum (cdr w))))))
```

```
> (map-sum '(((1 2 3) (4 5 6)) ((7 8 9))))
(21 24)
```

Решение 4.38.7

```
(defun flat (w)
  (mapcan #'(lambda (a) (if (listp a) (flat a) `(,a))) w))

(defun map-sum (w)
  (mapcar #'(lambda (a) (reduce #' + a)) (mapcar #'flat w)))

> (map-sum '(((1 2 3) (4 5 6)) ((7 8 9))))
(21 24)
```

Задача 4.39 *best-grades.lisp*

Имеется список студентов с информацией об итогах сессии, в котором указаны фамилия, номер группы, оценки по трем предметам. Определить название предмета, который был сдан лучше всего.

Решение 4.39.1

```
(defun sum-progres (w)
  (list (caar w) (apply #' + (mapcar #'cadr w))))

(defun best-course (w &aux (v (mapcar #'cddr w)))
  (list (mapcar #'car v)
        (mapcar #'cadr v)
        (mapcar #'caddr v)))

(defun best-grades (w)
  (caar (sort (mapcar #'sum-progres (best-course w))
              #'> :key #'cadr)))

> (best-grades '((cameron 111 (java 5) (ruby 4) (lisp 3)) (haberman
222 (java 3) (ruby 5) (lisp 3))))
RUBY
```

Задача 4.40 *unique-subsets.lisp*

Написать программу определения и вывода подписков, которые не входят ни в какие другие подписки. Вхождение определяется в теоретико-множественном смысле.

Решение 4.40.1

```
(defun check-sub (a w)
  (cond ((null w) nil)
        ((subsetp a (car w)) t)
```

```

      ((check-sub a (cdr w))))))

(defun unique-sub (w &aux (v (cdr w)))
  (cond ((null w) nil)
        ((check-sub (car w) v) (unique-sub (cdr w)))
        ((cons (car w) (unique-sub (cdr w))))))

> (unique-sub '((a b) (a b c) (d e) (f g h)))
((A B C) (D E) (F G H))

```

Решение 4.40.2

```

(defun check-sub (a w)
  (when w (if (subsetp a (car w)) t (check-sub a (cdr w)))))

(defun unique-sub (w &aux (v (cdr w)))
  (when w (if (check-sub (car w) v)
              (unique-sub (cdr w))
              (cons (car w) (unique-sub (cdr w))))))

> (unique-sub '((a b) (a b c) (d e) (f g h)))
((A B C) (D E) (F G H))

```

Задача 4.41 *permutate.lisp*

Для заданного списка получить множество всех возможных перестановок его элементов.

Решение 4.41.1

```

(defun permutate (w)
  (cond ((null w) nil)
        ((null (cdr w)) (list w))
        ((loop for a in w
                 nconc (mapcar #'(lambda (e) (cons a e))
                               (permutate (remove a w)))))))

> (permutate '(1 2 3))
((1 2 3) (1 3 2) (2 1 3) (2 3 1) (3 1 2) (3 2 1))

```

Решение 4.41.2

```

(defun permutate (w)
  (when w (if (cdr w)
              (loop for a in w
                     nconc (mapcar #'(lambda (e) (cons a e))
                                     (permutate (remove a w))))
              (list w))))

> (permutate '(1 2 3))
((1 2 3) (1 3 2) (2 1 3) (2 3 1) (3 1 2) (3 2 1))

```


Задача 4.42 *sine-penultimate.lisp*

Дан список. Удалить предпоследний элемент.

Решение 4.42.1

```
(defun sine-penultimate (w)
  (if (cddr w)
      (cons (car w) (sine-penultimate (cdr w)))
      (when (cdr w) (list (cadr w)))))

> (sine-penultimate '(1 2 3 4))
(1 2 4)
```

Решение 4.42.2

```
(defun sine-penultimate (w)
  (if (cddr w)
      (cons (car w) (sine-penultimate (cdr w)))
      (when (cdr w) `',(cadr w)))))

> (sine-penultimate '(1 2 3 4))
(1 2 4)
```

Решение 4.42.3

```
(defun sine-penultimate (w)
  (loop for a in w
        for b downfrom (length w)
        unless (= b 2) collect a))

> (sine-penultimate '(1 2 3 4))
(1 2 4)
```

Решение 4.42.4

```
(defun sine-penultimate (w)
  (append (butlast w 2) (last w)))

> (sine-penultimate '(1 2 3 4))
(1 2 4)
```

Задача 4.43 *check-prime-numbers.lisp*

Составить программу для проверки утверждения: «Результатами вычислений по формуле $x * x + x + 17$ при $0 \leq x \leq 15$ являются простые числа».

Решение 4.43.1

```
(defun primep (n)
  (loop for a from 2 to (isqrt n) never (zerop (mod n a))))
```

```
(defun prime-numbers (n m)
  (loop for a from n to m collect (+ (* a a) a 17)))

(defun check-prime-numbers (n m)
  (every #'primep (prime-numbers n m)))

> (prime-numbers 0 15)
(17 19 23 29 37 47 59 73 89 107 127 149 173 199 227 257)
> (check-prime-numbers 0 15)
T
> (prime-numbers 0 16)
(17 19 23 29 37 47 59 73 89 107 127 149 173 199 227 257 289)
> (check-prime-numbers 0 16)
NIL
```

Задача 4.44 *same-celsius.lisp*

Lisp basically teaches you to find the most straightforward solutions for a given problem, solutions you often don't find in e.g. C because it is very hard to differentiate between the structure of the problem and technical details (such as memory management).

Jens Oliver Gutsfeld

Составить программу, которая группирует дни месяца, когда температура воздуха была одинаковой.

Решение 4.44.1

```
(defun same-celsius (w)
  (mapcar
   #'(lambda (z) (mapcar #'car z))
   (remove-if-not
    #'cdr (mapcar
            #'(lambda (a) (remove-if-not
                          #'(lambda (e) (= a e))
                          w :key #'cadr))
            (remove-duplicates (mapcar #'cadr w)))))))

> (same-celsius '((1 15) (2 17) (3 15) (4 21) (5 19) (6 17) (7 17)))
((1 3) (2 6 7))
```

Задача 4.45 *sum-lists.lisp*

Дан список, который составлен из подсписков. Необходимо построить список сумм каждого подсписка.

Решение 4.45.1

```
(defun sum-lists (w)
  (when w (cons (apply #'+ (car w)) (sum-lists (cdr w)))))

> (sum-lists '((1 1) (1 1)))
(2 2)
```

Задача 4.46 minus>0.lisp

Дан числовой многоуровневый список. Определить функцию, возвращает список, состоящий из элементов исходного списка, отрицательные числа в котором заменены 0.

Решение 4.46.1

```
(defun minus>0 (w)
  (cond ((null w) nil)
        ((listp (car w)) (cons (minus>0 (car w)) (minus>0 (cdr w)))))
        ((minusp (car w)) (cons 0 (minus>0 (cdr w))))
        ((cons (car w) (minus>0 (cdr w))))))

> (minus>0 '(1 ((-2 3) -4) 5 (6)))
(1 ((0 3) 0) 5 (6))
```

Задача 4.47 unique-atoms.lisp

Дан многоуровневый список. Определить функцию, которая возвращает список атомов, встречающихся в исходном списке ровно один раз.

Решение 4.47.1

```
(defun flat (w acc)
  (cond ((null w) acc)
        ((atom w) (cons w acc))
        ((flat (car w) (flat (cdr w) acc)))))

(defun unique (w v)
  (cond ((null w) nil)
        ((> (count (car w) v) 1) (unique (cdr w) v))
        ((cons (car w) (unique (cdr w) v)))))

(defun unique-atoms (w)
  (let ((v (flat w nil)))
    (unique v v)))

> (unique-atoms '(1 2 (((2 3 4))) 7 (8 (9 1))))
(3 4 7 8 9)
```

Решение 4.47.2

```
(defun flat (w)
```

```

(mapcan #'(lambda (a) (if (listp a) (flat a) `(,a))) w))

(defun unique (w &optional (v w))
  (cond ((null w) nil)
        ((> (count (car w) v) 1) (unique (cdr w) v))
        ((cons (car w) (unique (cdr w) v)))))

(defun unique-atoms (w)
  (unique (flat w)))

> (unique-atoms '(1 2 (((2 3 4))) 7 (8 (9 1))))
(3 4 7 8 9)

```

Решение 4.47.3

```

(defun unique-atoms (w &optional acc ac
                     &aux (a (car w)) (d (cdr w)))
  (cond ((null w) (set-difference acc ac))
        ((listp a)
         (unique-atoms a (unique-atoms d acc ac) ac))
        ((member a acc) (unique-atoms d acc (cons a ac)))
        ((unique-atoms d (cons a acc) ac))))

> (unique-atoms '(1 2 (((2 3 4))) 7 (8 (9 1))))
(4 3 9 8 7)

```

Решение 4.47.4

```

(defun unique-atoms (w &optional acc ac
                     &aux (a (car w)) (d (cdr w)))
  (cond ((null w) (set-difference acc ac))
        ((listp a)
         (unique-atoms d (unique-atoms a acc ac) ac))
        ((member a acc) (unique-atoms d acc (cons a ac)))
        ((unique-atoms d (cons a acc) ac))))

> (unique-atoms '(1 2 (((2 3 4))) 7 (8 (9 1))))
(9 8 7 4 3)

```

Решение 4.47.5

```

(defun flat (w)
  (loop for a in w
        if (listp a) append (flat a)
        else collect a))

(defun unique (w &optional (v w))
  (cond ((null w) nil)
        ((> (count (car w) v) 1) (unique (cdr w) v))
        ((cons (car w) (unique (cdr w) v)))))

(defun unique-atoms (w)
  (unique (flat w)))

```

```
> (unique-atoms '(1 2 (((2 3 4))) 7 (8 (9 1))))
(9 8 7 4 3)
```

Задача 4.48 *intercom-section-marks.lisp*

Normally code is like an assembly line - data goes in, follows one of the predetermined paths, and comes out the other end transformed in some way. In Lisps, the assembly line can also assemble itself, according to rules you specify, without having to stop (run-time and compile-time can be interleaved arbitrarily). This allows you to easily build programs as layers of languages, which in turn gets you thinking about how languages work.

Vladislav Zorov

Дан список вещественных чисел $(a_1\ b_1\ a_2\ b_2\ldots a_n\ b_n)$, $a_i < b_i$. Рассматривая a_i и b_i , как левые и правые концы отрезков на одной и той же прямой, определить функцию, выдающую концы отрезка, являющегося пересечением всех этих отрезков. Если такого отрезка нет, то выдать nil.

Решение 4.48.1

```
(defun intercom (w)
  (when w (cons (loop for e from (car w) to (cadr w) collect e)
    (intercom (cddr w)))))

(defun intercom-section (w)
  (reduce #'intersection (intercom w)))

(defun intercom-section-marks (w &aux (v (intercom-section w)))
  (cons (car v) (last v)))

> (intercom-section-marks '(2 15 10 20 12 25))
(12 15)
```

Задача 4.49 *append-reversed.lisp*

Определить функцию, которая добавляет в конец списка все его элементы располагая их в обратном порядке.

Решение 4.49.1

```
(defun slow-reverse (w)
  (cond ((null w) nil)
        ((-append (slow-reverse (cdr w)) (list (car w))))))
```

```
(defun -append (w v)
  (cond ((null w) v)
        ((cons (car w) (-append (cdr w) v))))))
```

```
(defun append-reversed (w)
  (-append w (slow-reverse w)))
```

```
> (append-reversed '(1 2 3))
(1 2 3 3 2 1)
```

Решение 4.49.2

```
(defun fast-reverse (w &optional ac)
  (if (null w) ac (fast-reverse (cdr w) (cons (car w) ac))))
```

```
(defun -append (w v)
  (cond ((null w) v)
        ((cons (car w) (-append (cdr w) v))))))
```

```
(defun append-reversed (w)
  (-append w (fast-reverse w)))
```

```
> (append-reversed '(1 2 3))
(1 2 3 3 2 1)
```

Решение 4.49.3

```
(defun -reverse (w)
  (when w (-append (-reverse (cdr w)) (list (car w)))))
```

```
(defun -append (w v)
  (if (null w) v
      (cons (car w) (-append (cdr w) v))))
```

```
(defun append-reversed (w)
  (-append w (-reverse w)))
```

```
> (append-reversed '(1 2 3))
(1 2 3 3 2 1)
```

Решение 4.49.4

```
(defun -append (w v)
  (if (null w) v (cons (car w) (-append (cdr w) v))))
```

```
(defun append-reversed (w &optional (v w) acc)
  (cond ((null w) (-append v acc))
        ((append-reversed (cdr w) v (cons (car w) acc)))))
```

```
> (append-reversed '(1 2 3))
(1 2 3 3 2 1)
```

Решение 4.49.5

```
(defun -append (w v)
  (if (null w) v (cons (car w) (-append (cdr w) v))))

(defun append-reversed (w &optional (v w) acc)
  (if (null w)
      (-append v acc)
      (append-reversed (cdr w) v (cons (car w) acc))))

> (append-reversed '(1 2 3))
(1 2 3 3 2 1)
```

Задача 4.50 *cadrs.lisp*

Определить функцию, которая строит список элементов, стоящих в исходном списке на четных местах.

Решение 4.50.1

```
(defun cadrs (w)
  (cond ((null (cdr w)) nil)
        ((cons (cadr w) (cadrs (cddr w))))))

> (cadrs '(1 2 3 4 5 6 7))
(2 4 6)
```

Решение 4.50.2

```
(defun cadrs (w)
  (loop for a on w by #'cddr
        when (cadr a) collect (cadr a)))

> (cadrs '(1 2 3 4 5 6 7))
(2 4 6)
```

Задача 4.51 *rotate-last-two.lisp*

Определить функцию, которая меняет местами последний и предпоследний элементы списка, если они есть.

Решение 4.51.1 (автор – VH, www.cyberforum.ru)

```
(defun rotate-last-two (w)
  (nconc (butlast w 2) (reverse (last w 2))))

> (rotate-last-two '(1 2 3 4 5 6 7))
(1 2 3 4 5 7 6)
> (rotate-last-two '(1 2))
(2 1)
> (rotate-last-two '(1))
(1)
```

```
> (rotate-last-two '())
NIL
```

Задача 4.52 atom-level.lisp

Определить функцию, которая определяет уровень атомов для заданного списка. Для списка (a (b (c)) d) a, d – атомы нулевого уровня; b – атом первого уровня; c – атом второго уровня. В списке имена атомов не могут повторяться

Решение 4.52.1

```
(defun atom-level (w &optional (n 0))
  (cond ((null w) nil)
        ((atom (car w)) (cons (list (car w) n)
                               (atom-level (cdr w) n)))
        ((nconc (atom-level (car w) (+ n 1))
                  (atom-level (cdr w) n))))))

> (atom-level '(a b ((c d)) e f))
((A 0) (B 0) (C 3) (D 3) (E 1) (F 1))
```

Решение 4.52.2

```
(defun atom-level (w &optional (n 0) &aux (a (car w)) (d (cdr w)))
  (cond ((null w) nil)
        ((atom a) (cons (list a n) (atom-level d n)))
        ((nconc (atom-level a (1+ n)) (atom-level d n)))))

> (atom-level '(a b ((c d)) e f))
((A 0) (B 0) (C 3) (D 3) (E 1) (F 1))
```

Решение 4.52.3

```
(defun atom-level (w &optional (n 0) &aux (a (car w)) (d (cdr w)))
  (when w (nconc (if (atom a) (list (list a n)) (atom-level a (1+ n)))
                  (atom-level d n))))

> (atom-level '(a b ((c d)) e f))
((A 0) (B 0) (C 3) (D 3) (E 1) (F 1))
```

Решение 4.52.4

```
(defun atom-level (w &optional (n 0) &aux (a (car w)))
  (when w (nconc (if (atom a) `((,a ,n)) (atom-level a (1+ n)))
                  (atom-level (cdr w) n))))

> (atom-level '(a b ((c d)) e f))
((A 0) (B 0) (C 3) (D 3) (E 1) (F 1))
```


Задача 4.53 ascending-row.lisp

Дана целочисленная прямоугольная матрица размера $m \times n$. Определить в каких строках элементы расположены в возрастающем порядке.

Решение 4.53.1

```
(defun ascending-row (w)
  (remove-if-not #'(lambda (a) (apply #'< a)) w))

(ascending-row '((1 2 3) (2 3 1) (1 2 3)))
((1 2 3) (1 2 3))
```

Решение 4.53.2

```
(defun ascending-row (w)
  (mapcan #'(lambda (a) (when (apply #'< a) (list a))) w))

(ascending-row '((1 2 3) (2 3 1) (1 2 3)))
((1 2 3) (1 2 3))
```

Решение 4.53.3

```
(defun ascending-row (w)
  (mapcar #'(lambda (a) (when (apply #'< a) a)) w))

> (ascending-row '((1 2 3) (2 3 1) (1 2 3)))
((1 2 3) NIL (1 2 3))
```

Задача 4.54 cadr-car.lisp

Написать функцию перестановки местами соседних элементов, стоящих на четных и нечетных местах в исходном списке.

Решение 4.54.1

```
(defun cadr-car (w)
  (cond ((null w) nil)
        ((null (cdr w)) w)
        ((cons (cadr w) (cons (car w) (cadr-car (cddr w)))))))

> (cadr-car '(1 2 3 4 5 6))
(2 1 4 3 6 5)
> (cadr-car '(1 2 3 4 5))
(2 1 4 3 5)
```

Задача 4.55 transpose-matrix.lisp

Определить функцию транспонирования матрицы. Транспонированная матрица – матрица, полученная из исходной матрицы заменой строк на столбцы.

Решение 4.55.1

```
(defun transpose-matrix (m) (apply #'mapcar #'list m))

> (transpose-matrix '((1 2 3) (4 5 6) (7 8 9)))
((1 4 7) (2 5 8) (3 6 9))
```

Решение 4.55.2

```
(defun cr (v c)
  (when v (cons (funcall c v) (cr (cdr v) c))))

(defun transpose-matrix (v)
  (when (car v) (cons (cr v #'caar)
                      (transpose-matrix (cr v #'cdar)))))

> (transpose-matrix '((1 2 3) (4 5 6) (7 8 9)))
((1 4 7) (2 5 8) (3 6 9))
```

Задача 4.56 delete-twins.lisp

Дан список. Удалить из него списки размерностью 2.

Решение 4.56.1

```
(defun delete-twins (w)
  (remove-if #'(lambda (a) (and (listp a) (= (length a) 2))) w))

> (delete-twins '(0 (1) (2 2)))
(0 (1))
```

Решение 4.56.2

```
(defun delete-twins (w)
  (remove-if #'(lambda (a)
                  (and (listp a) (cadr a) (null (caddr a))))
            w))

> (delete-twins '(0 (1) (2 2)))
(0 (1))
```

Решение 4.56.3

```
(defun delete-twins (w)
  (cond ((null w) nil)
        ((and (listp (car w)) (= (length (car w)) 2))
         (delete-twins (cdr w)))
        ((cons (car w) (delete-twins (cdr w)))))

> (delete-twins '(0 (1) (2 2)))
(0 (1))
```

Задача 4.57 allocate-cards.lisp

Двн список из 36 игральных карт: ((6 x) (6 y) (6 u) (6 z) (7 x) (7 y) (7 u) (7 z) (8 x) (8 y) (8 u) (8 z) (9 x) (9 y) (9 u) (9 z) (10 x) (10 y) (10 u) (10 z) (11 x) (11 y) (11 u) (11 z) (12 x) (12 y) (12 u) (12 z) (13 x) (13 y) (13 u) (13 z) (14 x) (14 y) (14 u) (14 z)). Необходимо размешать и разделить карты между игроками. Количество игроков может быть от 2 до 4.

Решение 4.57.1

```
(defun oc (w n &optional (m 0) acc ac &aux (a (car w)) (d (cdr w)))
  (cond ((null w) (cons ac acc))
        ((= m n) (oc d n 1 (cons ac acc) (list a)))
        ((oc d n (1+ m) acc (cons a ac)))))

(defun drop (n w)
  (remove n w :test #'equalp))

(defun shuffle (w &aux (n (nth (random 35) w)))
  (when w (if n (cons n (shuffle (drop n w))) (shuffle (drop n w)))))

(defun allocate-cards (w n &aux (z (length w)))
  (oc (shuffle w) (floor z n)))

> (allocate-cards ' ((6 x) (6 y) (6 u) (6 z) (7 x) (7 y) (7 u) (7 z)
(8 x) (8 y) (8 u) (8 z) (9 x) (9 y) (9 u) (9 z) (10 x) (10 y) (10 u)
(10 z) (11 x) (11 y) (11 u) (11 z) (12 x) (12 y) (12 u) (12 z) (13 x)
(13 y) (13 u) (13 z) (14 x) (14 y) (14 u) (14 z)) 4)
(( (7 Z) (11 Y) (14 X) (10 Y) (8 U) (9 Y) (6 U) (12 X) (14 U))
 ( (7 Y) (8 Z) (8 X) (7 X) (6 X) (10 U) (10 X) (14 Z) (9 X))
 ( (9 Z) (11 X) (11 Z) (13 U) (7 U) (13 X) (9 U) (12 U) (13 Y))
 ( (6 Y) (10 Z) (6 Z) (12 Y) (8 Y) (12 Z) (11 U) (14 Y) (13 Z)))
```

Решение 4.57.2

```
(defun oc (w n m acc ac)
  (cond ((null w) (cons ac acc))
        ((= m n) (oc (cdr w) n 1 (cons ac acc) (list (car w))))
        ((oc (cdr w) n (1+ m) acc (cons (car w) ac)))))

(defun drop (n w)
  (remove n w :test #'equalp))

(defun shuffle (w)
  ((lambda (w n)
    (when w
      (if n (cons n (shuffle (drop n w))) (shuffle (drop n w)))))
   w (nth (random 35) w)))

(defun allocate-cards (w n)
  (oc (shuffle w) (floor (length w) n) 0 nil nil))
```

```
> (allocate-cards ' ((6 x) (6 y) (6 u) (6 z) (7 x) (7 y) (7 u) (7 z)
(8 x) (8 y) (8 u) (8 z) (9 x) (9 y) (9 u) (9 z) (10 x) (10 y) (10 u)
(10 z) (11 x) (11 y) (11 u) (11 z) (12 x) (12 y) (12 u) (12 z) (13 x)
(13 y) (13 u) (13 z) (14 x) (14 y) (14 u) (14 z)) 4)
((8 U) (6 Y) (10 U) (13 U) (14 U) (12 X) (9 U) (10 Y) (13 X))
((11 Z) (8 Z) (9 Y) (8 Y) (11 X) (11 Y) (7 X) (10 X) (13 Z))
((6 U) (6 X) (9 X) (7 Y) (10 Z) (14 Y) (12 U) (6 Z) (12 Y))
((8 X) (14 X) (11 U) (7 Z) (13 Y) (14 Z) (9 Z) (7 U) (12 Z)))
```

Решение 4.57.3

```
(defun drop (n w)
  (remove n w :test #'equalp))

(defun shuffle (w)
  ((lambda (w n)
    (when w
      (if n (cons n (shuffle (drop n w))) (shuffle (drop n w))))))
   w (nth (random 35) w)))

(defun 2-3-4 (w n)
  (cond ((= n 2) (list (subseq w 0 18) (subseq w 18)))
        ((= n 3) (list (subseq w 0 12)
                        (subseq w 12 24)
                        (subseq w 24)))
        ((list (subseq w 0 9)
                (subseq w 9 18)
                (subseq w 18 27)
                (subseq w 27)))))

(defun allocate-cards (w n)
  (2-3-4 (shuffle w) n))
```

```
> (allocate-cards ' ((6 x) (6 y) (6 u) (6 z) (7 x) (7 y) (7 u) (7 z)
(8 x) (8 y) (8 u) (8 z) (9 x) (9 y) (9 u) (9 z) (10 x) (10 y) (10 u)
(10 z) (11 x) (11 y) (11 u) (11 z) (12 x) (12 y) (12 u) (12 z) (13 x)
(13 y) (13 u) (13 z) (14 x) (14 y) (14 u) (14 z)) 4)
((7 Y) (12 X) (7 X) (12 Z) (8 X) (13 Z) (13 Y) (11 Y) (7 Z))
((9 X) (14 Z) (10 U) (7 U) (14 X) (6 U) (6 Y) (11 X) (11 U))
((10 Z) (6 X) (9 Y) (12 U) (10 Y) (14 Y) (8 U) (14 U) (9 Z))
((13 U) (11 Z) (9 U) (8 Y) (10 X) (6 Z) (12 Y) (8 Z) (13 X)))
```

Решение 4.57.4

```
(defun drop (n w) ; удаляет карту из списка
  (remove n w :test #'equalp))

(defun shuffle (w) ; тасует колоду
  (let ((n (nth (random 35) w)))
    (when w
      (if n (cons n (shuffle (drop n w))) (shuffle (drop n w))))))
```

```
(defun s (&rest m) (apply #'subseq m)) ; раздача одному игроку

(defun 2-3-4 (w n) ; раздает 2, 3 или 4 игрокам
  (cond ((= n 2) (list (s w 0 18) (s w 18)))
        ((= n 3) (list (s w 0 12) (s w 12 24) (s w 24)))
        ((list (s w 0 9) (s w 9 18) (s w 18 27) (s w 27))))))

(defun allocate-cards (w n) ; тасует и раздает 36 карт
  (2-3-4 (shuffle w) n))

> (allocate-cards ' ((6 x) (6 y) (6 u) (6 z) (7 x) (7 y) (7 u) (7 z)
(8 x) (8 y) (8 u) (8 z) (9 x) (9 y) (9 u) (9 z) (10 x) (10 y) (10 u)
(10 z) (11 x) (11 y) (11 u) (11 z) (12 x) (12 y) (12 u) (12 z) (13 x)
(13 y) (13 u) (13 z) (14 x) (14 y) (14 u) (14 z)) 4)
((12 Z) (7 U) (9 Z) (14 Z) (13 Y) (7 Z) (11 U) (14 X) (8 X))
((12 Y) (6 Z) (12 U) (14 Y) (10 Z) (7 Y) (9 X) (6 X) (6 U))
((13 Z) (10 X) (7 X) (11 Y) (11 X) (8 Y) (9 Y) (8 Z) (11 Z))
((13 X) (10 Y) (9 U) (12 X) (14 U) (13 U) (10 U) (6 Y) (8 U)))
```

Задача 4.58 average-sine.lisp

На вход подается список чисел A . Вычислить новый список чисел B , где B_i равно среднему арифметическому всего списка A , кроме A_i . То есть, если список $A = (1\ 2\ 3)$, то $B_1 = (2+3)/2$; $B_2 = (1+3)/2$; $B_3 = (1+2)/2$.

Решение 4.58.1

```
(defun average-sine (w &aux (n (1- (length w)))) (m (apply #' + w)))
  (when (> n 1) (mapcar #'(lambda (a) (/ (- m a) n)) w)))

> (average-sine '(2 3 5 6))
(14/3 13/3 11/3 10/3)
```

Решение 4.58.2

```
(defun average-sine (w &aux (n (1- (length w)))) (m (apply #' + w)))
  (when (> n 1) (mapcar #'(lambda (a) (float (/ (- m a) n))) w)))

> (average-sine '(2 3 5 6))
(4.6666665 4.3333335 3.6666667 3.3333333)
```

Решение 4.58.3

```
(defun average-sine (w)
  (loop for a in w collect (/ (- (apply #' + w) a) (1- (length w)))))

> (average-sine '(2 3 5 6))
(14/3 13/3 11/3 10/3)

> (average-sine '(2 3 5 6))
(14/3 13/3 11/3 10/3)
```

Решение 4.58.4

```
(defun average-sine (w &aux (n (1- (length w))) (m (apply #' + w)))
  (loop for a in w collect (/ (- m a) n)))

> (average-sine '(2 3 5 6))
(14/3 13/3 11/3 10/3)
> (average-sine '(2 3 5 6))
(14/3 13/3 11/3 10/3)
```

Решение 4.58.5

```
(defun average-sine (w)
  (let ((n (1- (length w))) (m (apply #' + w)))
    (loop for a in w collect (/ (- m a) n))))

> (average-sine '(2 3 5 6))
(14/3 13/3 11/3 10/3)
```

Решение 4.58.6 * при помощи labels - обернуть во вложенную функцию average с параметрами n = (1- (length w)) и m = (apply #' + w):

```
(defun average-sine (w)
  (labels ((average (n m)
            (loop for a in w collect (/ (- m a) n))))
    (average (1- (length w)) (apply #' + w))))

> (average-sine '(2 3 5 6))
(14/3 13/3 11/3 10/3)
```

Решение 4.58.7 * используя lambda - обернуть во вложенную безымянную функцию с параметрами n = (1- (length w)) и m = (apply #' + w):

```
(defun average-sine (w)
  ((lambda (n m)
    (loop for a in w collect (/ (- m a) n)))
   (1- (length w)) (apply #' + w)))

> (average-sine '(2 3 5 6))
(14/3 13/3 11/3 10/3)
```

Задача 4.59 not-zero.lisp

Определить в списке ненулевые элементы и создать из них список.

Решение 4.59.1

```
(defun not-zero (w)
  (cond ((null w) nil)
        ((zerop (car w)) (not-zero (cdr w)))
        (t (cons (car w) (not-zero (cdr w))))))
```

```
> (not-zero '(0 1))
(1)
```

Решение 4.59.2

```
(defun not-zero (w)
  (when w (if (zerop (car w))
              (not-zero (cdr w))
              (cons (car w) (not-zero (cdr w))))))

> (not-zero '(0 1))
(1)
```

Решение 4.59.3

```
(defun not-zero (w)
  (loop for a in w when (not (zerop a)) collect a))

> (not-zero '(0 1))
(1)
```

Решение 4.59.4

```
(defun not-zero (w)
  (remove-if #'zerop w))

> (not-zero '(0 1))
(1)
```

Решение 4.59.5

```
(defun not-zero (w)
  (remove 0 w))

> (not-zero '(0 1))
(1)
```

Задача 4.60 1..4>zero.lisp

Элементы списка *w*, значения которых лежат в интервале $[0,4]$, заменить на 0.

Решение 4.60.1

```
(defun 1..4>zero (w)
  (cond ((null w) nil)
        ((listp (car w)) (cons (1..4>zero (car w))
                                (1..4>zero (cdr w))))
        ((<= 1 (car w) 4) (cons 0 (1..4>zero (cdr w))))
        ((cons (car w) (1..4>zero (cdr w)))))

> (1..4>zero '((-2) -1) 0 1 (2) (3 (4)) 5 6 7))
```

```
(((-2) -1) 0 0 (0) (0 (0)) 5 6 7)
```

Решение 4.60.2

```
(defun 1..4>zero (w)
  (mapcar #'(lambda (a) (if (atom a)
                             (if (<= 1 a 4) 0 a)
                             (1..4>zero a))) w))

> (1..4>zero '((-2) -1) 0 1 (2) (3 (4)) 5 6 7))
((-2) -1) 0 0 (0) (0 (0)) 5 6 7)
```

Решение 4.60.3

```
(defun 1..4>zero (w)
  (loop for a in w collect (if (atom a)
                                (if (<= 1 a 4) 0 a)
                                (1..4>zero a))))

> (1..4>zero '((-2) -1) 0 1 (2) (3 (4)) 5 6 7))
((-2) -1) 0 0 (0) (0 (0)) 5 6 7)
```

Задача 4.61 odd/even.lisp

Дан список. Вывести в виде отдельных списков его нечетные и четные элементы.

Решение 4.61.1

```
(defun odd/even (w)
  (labels ((oddeven (v odds evens)
            (cond ((null v) (list (reverse odds) (reverse evens)))
                  ((oddp (car v)) (oddeven (cdr v)
                                              (cons (car v) odds)
                                              evens))
                  ((oddeven (cdr v) odds (cons (car v) evens))))))
    (oddeven w nil nil)))

> (odd/even '(1 2 3 4 5 6))
((1 3 5) (2 4 6))
```

Решение 4.61.2

```
(defun odd/even (w)
  (loop for i in w
        if (oddp i)
          collect i into odds
        else collect i into evens
        finally (return (values odds evens))))

> (odd/even '(1 2 3 4 5 6))
(1 3 5)
```



```
(2 4 6)
```

Решение 4.61.3

```
(defun odd/even (w &optional odd even &aux (a (car w)))
  (cond ((null w) (list (reverse odd) (reverse even)))
        ((oddp a) (odd/even (cdr w) (cons a odd) even))
        (t (odd/even (cdr w) odd (cons a even)))))
```

```
> (odd/even '(1 2 3 4 5 6))
((1 3 5) (2 4 6))
```

Решение 4.61.4

```
(defun odd/even (w
                 &optional odd even
                 &aux (a (car w)) (d (cdr w)))
  (if w
      (if (oddp a)
          (odd/even d (cons a odd) even)
          (odd/even d odd (cons a even)))
      (list (reverse odd) (reverse even))))
```

```
> (odd/even '(1 2 3 4 5 6))
((1 3 5) (2 4 6))
```

Решение 4.61.5

```
(defun odd/even (w)
  (list (remove-if #'evenp w) (remove-if #'oddp w)))

> (odd/even '(1 2 3 4 5 6))
((1 3 5) (2 4 6))
```

Задача 4.62 *remove-twins.lisp*

Создать функцию, которая должна выполнять преобразование сложного многоуровневого списка таким образом, чтобы в результате его элементы, которые в списке повторяются, возможно и неоднократно, остались в единственных экземплярах. Причем, так должно происходить и в первом уровне списка, и независимым образом, в подсписках любого уровня. То есть, после преобразования список может содержать по несколько одинаковых элементов, но в разных подсписках; в одном же подсписке повторы должны быть исключены.

Решение 4.62.1

```
(defun remove-twins (w)
  (cond ((null w) nil)
        ((listp (car w)) (cons (remove-twins (car w))
                                (remove-twins (cdr w))))
        ((member (car w) (cdr w)) (remove-twins (cdr w))))
```

```
((cons (car w) (remove-twins (cdr w)))))
```

```
> (remove-twins '(1 (1 1) a b (a a (b b))))
(1 (1) A B (A (B)))
```

Решение 4.62.2

```
(defun remove-twins (w &aux (a (car w)) (d (cdr w)))
  (cond ((null w) nil)
        ((listp a) (cons (remove-twins a) (remove-twins d)))
        ((member a d) (remove-twins d))
        ((cons a (remove-twins d)))))
```

```
> (remove-twins '(1 (1 1) a b (a a (b b))))
(1 (1) A B (A (B)))
```

Решение 4.62.3 * удалять последующие вхождения повторяющихся элементов

```
(defun remove-twins (w)
  (cond ((null w) nil)
        ((listp (car w)) (cons (remove-twins (car w))
                                (remove-twins (cdr w))))
        ((cons (car w)
                (remove-twins (remove (car w) (cdr w)))))))
```

```
> (remove-twins '(0 1 (2 3 4 (5 1 5) 5) 4 2))
(0 1 (2 3 4 (5 1) 5) 4 2)
```

Решение 4.62.4

```
(defun remove-twins (w &aux (a (car w)) (d (cdr w)))
  (cond ((null w) nil)
        ((listp a) (cons (remove-twins a) (remove-twins d)))
        ((member a d) (remove-twins d))
        ((cons a (remove-twins d)))))
```

```
> (remove-twins '(1 (1 1) a b (a a (b b))))
(1 (1) A B (A (B)))
```

Задача 4.63 *pair-elms.lisp*

Составить функцию, группирующую попарно смежные элементы списка в под-списки. Если выходной список состоит из нечетного числа элементов, то последний элемент отбрасывается.

Решение 4.63.1

```
(defun pair-elms (w)
  (when (cdr w) (cons (list (car w) (cadr w)) (pair-elms (cddr w)))))
```

```
> (pair-elms '(1 a 2 b))
((1 A) (2 B))
```

```
> (pair-elms '(1 a 2))
((1 A))
```

Решение 4.63.2

```
(defun pair-elms (w)
  (when (cdr w) (cons `,(car w) ,(cadr w) (pair-elms (cddr w)))))

> (pair-elms '(1 a 2 b))
((1 A) (2 B))
> (pair-elms '(1 a 2))
((1 A))
```

Решение 4.63.3

```
(defun pair-elms (w)
  (loop for (a b) on w by #'cddr
        when b collect (list a b)))

> (pair-elms '(1 a 2 b))
((1 A) (2 B))
> (pair-elms '(1 a 2))
((1 A))
```

Задача 4.64 reverse-mxx.lisp

Превратить матрицу, переписав в списках, являющихся элементами данной матрицы, элементе упомянутых списков в обратном порядке (только в верхнем уровне этих списков). Понятно, что списки, о которых идет речь, должны быть под списками уже третьего уровня в исходном списке которым представлена сама матрица.

Решение 4.64.1

```
(defun reverse-mx (w)
  (cond ((null w) nil)
        ((cons (reverse (car w)) (reverse-mx (cdr w))))))

(defun reverse-mxx (w)
  (cond ((null w) nil)
        ((cons (reverse-mx (car w)) (reverse-mxx (cdr w))))))

> (reverse-mxx '(((a b) (c d) (e f)) ((a b) (c d) (e f)))
  ((B A) (D C) (F E)) ((B A) (D C) (F E)))
```

Решение 4.64.2

```
(defun reverse-mx (w)
  (if (null w) nil
      (cons (reverse (car w)) (reverse-mx (cdr w)))))

(defun reverse-mxx (w)
```

```

    (if (null w) nil
        (cons (reverse-mx (car w)) (reverse-mxx (cdr w)))))

> (reverse-mxx '(((a b) (c d) (e f)) ((a b) (c d) (e f))
  ((B A) (D C) (F E)) ((B A) (D C) (F E)))

```

Решение 4.64.3

```

(defun reverse-mx (w)
  (when w (cons (reverse (car w)) (reverse-mx (cdr w)))))

(defun reverse-mxx (w)
  (when w (cons (reverse-mx (car w)) (reverse-mxx (cdr w)))))

> (reverse-mxx '(((a b) (c d) (e f)) ((a b) (c d) (e f))
  ((B A) (D C) (F E)) ((B A) (D C) (F E)))

```

Решение 4.64.4

```

(defun reverse-mxx (w)
  (loop for v in w collect
        (loop for a in v collect (reverse a))))

> (reverse-mxx '(((a b) (c d) (e f)) ((a b) (c d) (e f))
  ((B A) (D C) (F E)) ((B A) (D C) (F E)))

```

Решение 4.64.5

```

(defun reverse-mx (w)
  (mapcar #'reverse w))

(defun reverse-mxx (w)
  (mapcar #'reverse-mx w))

> (reverse-mxx '(((a b) (c d) (e f)) ((a b) (c d) (e f))
  ((B A) (D C) (F E)) ((B A) (D C) (F E)))

```

Решение 4.64.6

```

(defun reverse-mxx (w)
  (cond ((null w) nil)
        ((atom (car w)) (reverse w))
        ((cons (reverse-mxx (car w)) (reverse-mxx (cdr w)))))

> (reverse-mxx '(((a b) (c d) (e f)) ((a b) (c d) (e f))
  ((B A) (D C) (F E)) ((B A) (D C) (F E)))

```

Решение 4.64.7

```

(defun reverse-mx (w)
  (loop for (a b) in w collect (list b a)))

(defun reverse-mxx (w)

```

```
(mapcar #'reverse-mx w))
```

```
> (reverse-mxx '(((a b) (c d) (e f)) ((a b) (c d) (e f))))
(( (B A) (D C) (F E)) ((B A) (D C) (F E)))
```

Решение 4.64.8

```
(defun reverse-mx (w)
  (loop for (a b) in w collect `((,b ,a))))
```

```
(defun reverse-mxx (w)
  (mapcar #'reverse-mx w))
```

```
> (reverse-mxx '(((1 2) (3 4) (5 6)) ((1 2) (3 4) (5 6))))
(((2 1) (4 3) (6 5)) ((2 1) (4 3) (6 5)))
```

Решение 4.64.9

```
(defun reverse-mxx (w)
  (loop for ((a b) (c d) (e f)) in w
        collect `((,b ,a) (, d ,c) (,f ,e))))
```

```
> (reverse-mxx '(((1 2) (3 4) (5 6)) ((1 2) (3 4) (5 6))))
(((2 1) (4 3) (6 5)) ((2 1) (4 3) (6 5)))
```

Решение 4.64.10

```
(defun reverse-mxx (w)
  (mapcar #'(lambda (a) (mapcar #'reverse a)) w))
```

```
> (reverse-mxx '(((1 2) (3 4) (5 6)) ((1 2) (3 4) (5 6))))
(((2 1) (4 3) (6 5)) ((2 1) (4 3) (6 5)))
```

Задача 4.65 annex.lisp

Определить функцию для преобразования списка по принципу: (a b c) -> (((a) b) c).

Решение 4.65.1

```
(defun annex (w)
  (reduce #'list (cons (list (car w)) (cdr w))))
```

```
> (annex '(a b c d e))
((((A) B) C) D) E)
```

Решение 4.65.2

```
(defun annex (w)
  (reduce #'list (cdr w) :initial-value (list (car w))))
```

```
> (annex '(a b c d e))
```

```
(((((A) B) C) D) E)
```

Решение 4.65.3

```
(defun annex (w)
  (reduce #'list (cdr w) :initial-value `((, (car w)))))

> (annex '(a b c d e))
((((A) B) C) D) E)
```

Решение 4.65.4

```
(defun annex (w &optional acc)
  (if (null w)
      (car acc)
      (annex (cdr w) (list (reverse (cons (car w) acc))))))

> (annex '(a b c d e))
((((A) B) C) D) E)
```

Решение 4.65.5

```
(defun annex (w &optional acc)
  (if w
      (annex (cdr w) (list (reverse (cons (car w) acc))))
      (car acc)))

> (annex '(a b c d e))
((((A) B) C) D) E)
```

Решение 4.65.6

```
(defun annex (w &optional acc)
  (cond ((null w) (car acc))
        ((annex (cdr w) (list (reverse (cons (car w) acc)))))))

> (annex '(a b c d e))
((((A) B) C) D) E)
```

Решение 4.65.7

```
(defun annex (w)
  (let ((r (list (car w))))
    (dolist (i (cdr w) r)
      (setq r (cons r (list i))))))

> (annex '(a b c d e))
((((A) B) C) D) E)
```

Решение 4.65.8

```
(defun annex (w &aux (r `((, (car w)))))
  (dolist (i (cdr w) r)
```

```
(setf r (cons r `(,i))))

> (annex '(a b c d e))
((((A) B) C) D) E)
```

Задача 4.66 *remove-lists.lisp*

Определить макрос, который удаляет списки из списка.

Решение 4.66.1

```
(defmacro remove-lists (w)
  `(cond ((null ,w) nil)
        ((atom (car ,w)) (cons (car ,w) (remove-lists (cdr ,w))))
        ((remove-lists (cdr ,w)))))

> (remove-lists '(1 (2)))
(1)
```

Решение 4.66.2

```
(defmacro remove-lists (w)
  `(remove-if #'listp ,w))

> (remove-lists '(1 (2)))
(1)
```

Решение 4.66.3

```
(defun remove-lists (w)
  `(loop for a in ,w when (atom a) collect a))

> (remove-lists '(1 (2)))
(LOOP FOR A IN (1 (2)) WHEN (ATOM A) COLLECT A)

(defmacro remove-lists (w)
  `(loop for a in ,w when (atom a) collect a))

> (remove-lists '(1 (2)))
(1)
```

Задача 4.67 *read&change.lisp*

Есть список (x1 x2 x3 ... xn). Необходимо списку присвоить значения, которые вводятся с клавиатуры. Оканчивать ввод если либо список кончился, либо пользователь ввел nil.

Решение 4.67.1

```
(defun read&change (lst)
  (when lst
```

```

      (let ((e (read)))
        (if e (cons e (read&change (cdr lst))) lst))))

> (read&change '(a a a))
3
4
5
(3 4 5)

```

Задача 4.68 *chess-board.lisp*

What characterizes Lisp most of all is list processing. The code you work with is embedded inside a structure, and it offers a consistent way of accessing anything within that structure. It's not just lines of text. People could say that this is no different from any other language, since they also operate within structures, but what makes Lisp different is you get access to that structure inside the language. Certain languages give the ability to modify the structure of the code being executed, outside of the "normal rules," called meta-programming, and Lisp is one of those. Some modern languages have had meta-programming abilities, but from what I've seen they're clunky, as if they're tacked on as afterthoughts. We've got a ways to go before they match the elegance of some older languages like Lisp and Smalltalk.

Mark Miller

Имеется шахматная доска произвольного размера, клетки которой по горизонтали пронумерованы буквами, а по вертикали - числами. Постройте список пар координат всех клеток доски, в соответствии с двумя заданными списками букв и чисел нумерации доски.

Решение 4.68.1

```

(defun row (a w)
  (when w (cons (list a (car w))
                (row a (cdr w)))))

(defun chess-board (v w)
  (when v (nconc (row (car v) w) (chess-board (cdr v) w))))

> (chess-board '(a b c) '(1 2))
((A 1) (A 2) (B 1) (B 2) (C 1) (C 2))

```


Задача 4.69 *high-landers.lisp*

Из заданного списка получить список, элементами которого будут являться только атомы верхнего уровня.

Решение 4.69.1

```
(defun high-landers (w &aux (a (car w)) (d (cdr w)))
  (cond ((null w) nil)
        ((atom a) (cons a (high-landers d)))
        ((high-landers d))))
```

```
> (high-landers '(2 (a e) 4 ((5) y) c))
(2 4 C)
```

Задача 4.70 *matrix-column-average.lisp*

Дана вещественная матрица. Определить средние арифметические значения элементов столбцов.

Решение 4.70.1

```
(defun matrix-column-average (w)
  (mapcar #'(lambda (a) (/ (reduce #'+ a) (length a)))
          (apply #'mapcar #'list w)))

> (matrix-column-average '((1 2.5 3) (4.5 5 6) (7 8 9.5)))
(4.1666665 5.1666665 6.1666665)
```

Задача 4.71 *minus>a.lisp*

Определить функцию, которая заменяет все отрицательные элементы списка на *a*.

Решение 4.71.1

```
(defun minus>a (w &aux (a (car w)) (d (cdr w)))
  (cond ((null w) nil)
        ((minusp a) (cons 'a (minus>a d)))
        ((cons a (minus>a d)))))
```

```
> (minus>a '(-1 1))
(A 1)
```

Решение 4.71.2

```
(defun minus>a (x)
  (cond ((null x) nil)
        ((minusp (car x)) (cons 'a (minus>a (cdr x))))
        ((cons (car x) (minus>a (cdr x)))))
```

```
> (minus>a '(-1 1))
(A 1)
```

Решение 4.71.3

```
(defun minus>a (w a)
  (cond ((null w) nil)
        ((minusp (car w)) (cons a (minus>a (cdr w) a)))
        ((cons (car w) (minus>a (cdr w) a)))))
```

```
> (minus>a '(-1 1) 10)
(10 1)
```

Решение 4.71.4

```
(defun minus>a (w b &aux (a (car w)))
  (when w (cons (if (minusp a) b a) (minus>a (cdr w) b)))))
```

```
> (minus>a '(-1 2 -3 4 -5 7) 0)
(0 2 0 4 0 7)
```

Решение 4.71.5

```
(defun minus>a (w a)
  (loop for e in w if (minusp e) collect a else collect e))
```

```
> (minus>a '(-1 1) 10)
(10 1)
```

Решение 4.71.6

```
(defun minus>a (w a)
  (mapcar #'(lambda (e) (if (minusp e) a e)) w))
```

```
> (minus>a '(-1 1) 10)
(10 1)
```

Решение 4.71.7

```
(defun minus>a (w) (substitute-if 'a #'minusp w))
```

```
> (minus>a '(-1 1))
(A 1)
```

Задача 4.72 *xelms.lisp*

Описать функцию, которая берет в качестве аргумента список атомов и выдает список всех атомов, встречающихся в списке вместе с частотой их появления.

Решение 4.72.1

```
(defun xelms(w)
  (labels
    ((xlm (w ac)
      (cond ((null w) (reverse ac))
            ((find (car w) ac :key #'car) (xlm (cdr w) ac))
            ((xlm (cdr w) (cons (list (car w) (count (car w) w))
                                ac))))))
    (xlm w nil)))

> (xelms'(a b a b a c a))
((A 4) (B 2) (C 1))
> (xelms'(ni to ni to))
((NI 2) (TO 2))
```

Решение 225.2

```
(defun xlm (w ac)
  (cond ((null w) (reverse ac))
        ((find (car w) ac :key #'car) (xlm (cdr w) ac))
        ((xlm (cdr w) (cons (list (car w) (count (car w) w)) ac)))))

(defun xelms (w) (xlm w nil))

> (xelms '(a b a b a c a))
((A 4) (B 2) (C 1))
> (xelms '(ni to ni to))
((NI 2) (TO 2))
```

Задача 4.73 matrix.lisp

Найти сумму всех чисел, которые входят числовыми компонентами в те списки, которые, являясь элементами матрицы, лежат на ее главной диагонали. Упомянутые списки должны быть подписками уже третьего уровня в исходном списке, которым представлена сама матрица (не учитывать чисел, которые просто находятся на диагонали и являются обычными элементами матрицы):

исходная

```
(1 2 5) (1 2) 6
5 6 (1 2)
6 8 (3 1)
```

результат

```
(1 2 5 (8)) (1 2) 6
5 6 (1 2)
6 8 (3 1 (4))
```

Решение 4.73.1

```
(defun matrix (w &optional (n 0) &aux (a (car w)) (d (cdr w)))
  (cond ((null w) nil)
        ((and (listp a) (or (= n 0) (= n 4) (= n 8)))
```

```

      (cons (append a (list (list (reduce #' + a))))
            (matrix d (1+ n))))
    ((cons a (matrix d (1+ n)))))

> (matrix '((1 2 5) (1 2) 6 5 6 (1 2) 6 8 (3 1)))
((1 2 5 (8)) (1 2) 6 5 6 (1 2) 6 8 (3 1 (4)))

```

Задача 4.74 x-doubles.lisp

Построить рекурсивную функцию средствами элементарного Лиспа, которая принимает список и возвращает список входящих в него атомов без повторов.

Решение 4.74.1

```

(defun x-doubles (w)
  (cond ((null w) nil)
        ((member (car w) (cdr w)) (x-doubles (cdr w)))
        (t (cons (car w) (x-doubles (cdr w))))))

> (x-doubles '(1 2 3 1 2 3 1 2 3))
(1 2 3)

```

Решение 4.74.2 * сохраняя порядок элементов

```

(defun doubles (w acc)
  (cond ((null w) (reverse acc))
        ((member (car w) acc) (doubles (cdr w) acc))
        (t (doubles (cdr w) (cons (car w) acc)))))

(defun x-doubles (w)
  (doubles w nil))

> (x-doubles '(1 3 2 1 2))
(1 3 2)

```

Решение 4.74.3

```

(defun x-doubles (w &optional ac)
  (cond ((null w) (reverse ac))
        ((member (car w) ac) (x-doubles (cdr w) ac))
        ((x-doubles (cdr w) (cons (car w) ac)))))

> (x-doubles '(1 3 2 1 2))
(1 3 2)

```

Решение 4.74.4

```

(defun x-doubles (w &optional ac &aux (a (car w)) (d (cdr w)))
  (if w (x-doubles d (if (member a ac) ac (cons a ac))) (reverse ac)))

> (x-doubles '(1 3 2 1 2))

```

```
(1 3 2)
```

Решение 4.74.5

```
(defun x-doubles (w &optional ac)
  (cond ((null w) (reverse ac))
        ((x-doubles (cdr w) (adjoin (car w) ac)))))
```

```
> (x-doubles '(1 3 2 1 2))
(1 3 2)
```

Решение 4.74.6

```
(defun x-doubles (w &optional ac)
  (if w (x-doubles (cdr w) (adjoin (car w) ac)) (reverse ac)))
```

```
> (x-doubles '(1 3 2 1 2))
(1 3 2)
```

Решение 4.74.7

```
(defun x-doubles (w &optional ac)
  (if w (x-doubles (cdr w) (adjoin (car w) ac)) (reverse ac)))
```

```
> (x-doubles '(1 3 2 1 2))
(1 3 2)
```

Решение 4.74.8

```
(defun x-doubles (w &aux ac)
  (dolist (a w (nreverse ac))
    (unless (member a ac) (push a ac))))
```

```
> (x-doubles '(1 3 2 1 2))
(1 3 2)
```

Решение 4.74.9 * для многоуровневого списка

```
(defun x-doubles (w &optional ac)
  (cond ((null w) ac)
        ((atom w) (adjoin w ac))
        ((x-doubles (car w) (x-doubles (cdr w) ac)))))
```

```
> (x-doubles '(1 (3) (2 (1))))
(3 2 1)
```

Задача 4.75 zeros.lisp

Определить функцию, заменяющую все отрицательные элементы списка значением 0.

Решение 4.75.1

```
(defun zeros (w)
  (cond ((null w) nil)
        ((< (car w) 0) (cons 0 (zeros (cdr w)))))
  ((cons (car w) (zeros (cdr w))))))
```

```
> (zeros '(1 -5 6 -7 8 -9 -4))
(1 0 6 0 8 0 0)
```

Решение 4.75.2

```
(defun zeros (w) (substitute-if 0 #'minusp w))
```

```
> (zeros '(1 -5 6 -7 8 -9 -4))
(1 0 6 0 8 0 0)
```

Решение 4.75.3

```
(defun zeros (w)
  (mapcar #'(lambda (a) (if (minusp a) 0 a)) w))
```

```
> (zeros '(1 -5 6 -7 8 -9 -4))
(1 0 6 0 8 0 0)
```

Решение 4.75.4

```
(defun zeros (w)
  (loop for i in w
        if (minusp i) collect 0
        else collect i))
```

```
> (zeros '(1 -5 6 -7 8 -9 -4))
(1 0 6 0 8 0 0)
```

Задача 4.76 *grow-deep.lisp*

Дан список чисел произвольного уровня вложенности. Увеличить каждый элемент на единицу.

Решение 4.76.1

```
(defun grow-deep (w)
  (cond ((null w) nil)
        ((atom (car w)) (cons (1+ (car w)) (grow-deep (cdr w)))))
  ((cons (grow-deep (car w)) (grow-deep (cdr w))))))
```

```
> (grow-deep '((1 (2 3)) ((4 (5) 6)) ((7 8) 9)))
((2 (3 4)) ((5 (6) 7)) ((8 9) 10))
```

Решение 4.76.2

```
(defun grow-deep (w &aux (a (car w)) (d (cdr w)))
```

```

(cond ((null w) nil)
      ((atom a) (cons (1+ a) (grow-deep d)))
      ((cons (grow-deep a) (grow-deep d)))))

> (grow-deep '((1 (2 3)) ((4 (5) 6)) ((7 8) 9)))
((2 (3 4)) ((5 (6) 7)) ((8 9) 10))

```

Решение 4.76.3

```

(defun grow-deep (x &aux r)
  (reverse (dolist (i x r)
    (cond ((numberp i) (setq r (cons (+ i 1) r)))
          ((listp i) (setq r (cons (grow-deep i) r)))
          ((setq r (cons (list i) r)))))))

> (grow-deep '((1 2 (3 4) ((5 6))) -1)))
((2 3 (4 5) ((6 7))) 0)

```

Решение 4.76.4

```

(defun grow-deep (w &aux (a (car w)) (d (cdr w)))
  (when w (cons (if (atom a) (1+ a) (grow-deep a)) (grow-deep d))))

> (grow-deep '((1 2 (3 4) ((5 6))) -1)))
((2 3 (4 5) ((6 7))) 0)

```

Решение 4.76.5

```

(defun grow-deep (w &aux (a (car w)))
  (when w (cons (if (atom a) (1+ a) (grow-deep a))
    (grow-deep (cdr w)))))

> (grow-deep '((1 2 (3 4) ((5 6))) -1)))
((2 3 (4 5) ((6 7))) 0)

```

Решение 4.76.6

```

(defun grow-deep (w)
  (loop for a in w collect (if (atom a) (1+ a) (grow-deep a))))

> (grow-deep '(1 (2) ((3) 4) 5))
(2 (3) ((4) 5) 6)

```

Решение 4.76.7

```

(defun grow-deep (w)
  (loop for a in w collect
    (if (atom a) (if (numberp a) (1+ a) a) (grow-deep a))))

> (grow-deep '(1 (2 nil) ((3) nil 4) 5))
(2 (3 NIL) ((4) NIL 5) 6)

```

Задача 4.77 up-words.lisp

Дан текст. Сделать заглавной каждую букву каждого слова. Текст рекомендуется представлять списком списков: каждое предложение - список слов, весь текст - список предложений.

Решение 4.77.1

```
(defun up-word (w) (mapcar #'string-upcase w))

(defun up-words (w) (mapcar #'up-words w))

> (up-words '(("abc" "abc") ("abc" "abc")))
(("ABC" "ABC") ("ABC" "ABC"))
```

Задача 4.78 drop-dups.lisp

Удалить дубликаты из многоуровневого списка.

Решение 4.78.1

```
(defun drop-dups (w)
  (cond (w (let ((a (car w)) (d (cdr w)))
              (cond ((listp a) (cons (drop-dups a) (drop-dups d)))
                    ((member a d) (drop-dups d))
                    ((cons a (drop-dups d)))))))

> (drop-dups '(1 2 (2 2 3) 3))
(1 2 (2 3) 3)
```

Решение 4.78.2

```
(defun drop-dups (w)
  (cond (w ((lambda (a d)
              (cond ((listp a) (cons (drop-dups a) (drop-dups d)))
                    ((member a d) (drop-dups d))
                    ((cons a (drop-dups d))))
            (car w) (cdr w)))))

> (drop-dups '(1 2 (2 2 3) 3))
(1 2 (2 3) 3)
```

Решение 4.78.3

```
(defun drop-dups (w &aux (a (car w)) (d (cdr w)))
  (when w (cond ((listp a) (cons (drop-dups a) (drop-dups d)))
                ((member a d) (drop-dups d))
                ((cons a (drop-dups d))))))

> (drop-dups '(1 2 (2 2 3) 3))
(1 2 (2 3) 3)
```


Решение 4.78.4

```
(defun drop-dups (w &aux (a (car w)) (d (cdr w)))
  (cond ((null w) nil)
        ((listp a) (cons (drop-dups a) (drop-dups d)))
        ((member a d) (drop-dups d))
        ((cons a (drop-dups d)))))

> (drop-dups '(1 2 (2 2 3) 3))
(1 2 (2 3) 3)
```

Решение 4.78.5 * попутно превратить список в одноуровневый

```
(defun drop-dups (w &optional acc)
  (cond ((null w) acc)
        ((listp w) (drop-dups (car w) (drop-dups (cdr w) acc)))
        ((member w acc) acc)
        ((cons w acc))))

> (drop-dups '(1 2 (2 2 3) 3 (3 4 4) 5))
(1 2 3 4 5)
```

Решение 4.78.6

```
(defun drop-dups (w &optional acc)
  (cond ((null w) acc)
        ((listp w) (drop-dups (car w) (drop-dups (cdr w) acc)))
        ((adjoin w acc))))

> (drop-dups '(1 2 (2 2 3) 3 (3 4 4) 5))
(1 2 3 4 5)
```

Решение 4.78.7

```
(defun drop-dups (w &optional acc)
  (if w
      (if (listp w)
          (drop-dups (car w) (drop-dups (cdr w) acc))
          (adjoin w acc))
      acc))

> (drop-dups '(1 2 (2 2 3) 3 (3 4 4) 5))
(1 2 3 4 5)
```

Решение 4.78.8

```
(defun drop-dups (w &optional acc)
  (if w
      (if (atom w)
          (adjoin w acc)
          (drop-dups (car w) (drop-dups (cdr w) acc)))
      acc))
```

```
> (drop-dups '(1 2 (2 2 3) 3 (3 4 4) 5))
(1 2 3 4 5)
```

Решение 4.78.9

```
(defun drop-dups (w &optional acc)
  (cond ((null w) acc)
        ((atom w) (adjoin w acc))
        ((drop-dups (car w) (drop-dups (cdr w) acc)))))
```

```
> (drop-dups '(1 2 (2 2 3) 3 (3 4 4) 5))
(1 2 3 4 5)
```

Задача 4.79 *exponent.lisp*

По списку, состоящему из атомов-чисел, составить список значений $\exp(ai)$, где ai - i -й элемент списка. Значения $\exp(ai)$ большие $(ai+2)$ не учитывать.

Решение 4.79.1

```
(defun exponent (w)
  (when w ((lambda (a d)
             (if (numberp a)
                 (if (< (exp a) (+ a 2))
                     (cons (exp a) (exponent d))
                     (exponent d))
                 (cons nil (exponent d)))))
    (car w) (cdr w))))
```

```
> (exponent '(0.5 b 1 2 3 (10 20) 4))
(1.6487212 NIL 2.7182817 NIL)
```

Решение 4.79.2

```
(defun exponent (w)
  (cond (w ((lambda (a d)
              (cond ((numberp a)
                     (cond ((< (exp a) (+ a 2))
                           (cons (exp a) (exponent d)))
                           ((exponent d)))))
                ((cons nil (exponent d)))))
        (car w) (cdr w)))))
```

```
> (exponent '(0.5 b 1 2 3 (10 20) 4))
(1.6487212 NIL 2.7182817 NIL)
```

Задача 4.80 *cap-words.lisp*

Написать программу обработки текста естественного языка с использованием функционалов. Текст рекомендуется представлять списком списков:

каждое предложение - список слов, весь текст - список предложений. Задание - сделать так, чтобы каждое слово начиналось с заглавной буквы.

Решение 4.80.1

```
(defun cap-words (text)
  (mapcar #'(lambda (sentence)
              (cons (string-capitalize (car sentence))
                    (cdr sentence)))
    text))

> (cap-words '(("aa" "aa") ("aa" "aa")))
(("Aa" "aa") ("Aa" "aa"))
```

Решение 4.80.2

```
(defun cap-word (w)
  (mapcar #'string-capitalize w))

(defun cap-words (w)
  (mapcar #'cap-word w))

> (cap-words '(("abc" "abc") ("abc" "abc")))
(("Abc" "Abc") ("Abc" "Abc"))
```

Решение 4.80.3

```
(defun cap-words (w)
  (mapcar #'(lambda (v) (mapcar #'string-capitalize v)) w))

> (cap-sents '(("abc" "abc") ("abc" "abc")))
(("Abc" "Abc") ("Abc" "Abc"))
```

Задача 4.81 *sum-pair.lisp*

В числовом списке сложить каждые два соседних элемента, пример работы:
(f '(2 3 4 5 6 7)) > ((5) (9) (13))

Решение 4.81.1

```
(defun sum-pair (w)
  (cond ((null w) nil)
        ((cons (list (+ (car w) (cadr w))) (sum-pair (cddr w))))))

> (sum-pair '(2 3 4 5 6 7))
((5) (9) (13))
```

Решение 4.81.2

```
(defun sum-pair (w)
  (if (null w)
      nil
```

```
(cons (list (+ (car w) (cadr w))) (sum-pair (cddr w))))
```

```
> (sum-pair '(2 3 4 5 6 7))
((5) (9) (13))
```

Решение 4.81.3

```
(defun sum-pair (w)
  (when w (cons (list (+ (car w) (cadr w)))
                (sum-pair (cddr w)))))
```

```
> (sum-pair '(2 3 4 5 6 7))
((5) (9) (13))
```

Решение 4.81.4

```
(defun sum-pair (w)
  (loop for e on w by #'cddr collect (list (+ (car e) (cadr e)))))
```

```
> (sum-pair '(2 3 4 5 6 7))
((5) (9) (13))
```

Решение 4.81.5

```
(defun sum-pair (w)
  (loop for (a b) on w by #'cddr
        collect (list (+ a b))))
```

```
> (sum-pair '(2 3 4 5 6 7))
((5) (9) (13))
```

В том числе для списков с нечетным числом элементов:

Решение 4.81.6

```
(defun sum-pair2 (w)
  (if (cdr w)
      (cons (list (+ (car w) (cadr w)))
            (sum-pair2 (cddr w)))
      (when w (list w))))
```

```
> (sum-pair '(2 3 4 5 6 7 8))
((5) (9) (13) (8))
```

Решение 4.81.7

```
(defun sum-pair (w)
  (loop for (a b) on w by #'cddr
        collect (list (+ a (if b b 0)))))
```

```
> (sum-pair '(2 3 4 5 6 7 8))
((5) (9) (13) (8))
```

Задача 4.82 drop-pluspp.lisp

Определить функцию, которая из заданного списка удаляет все элементы являющиеся положительным числами. Список может быть многоуровневым и содержать не только числа.

Решение 4.82.1

```
(defun drop-pluspp (w)
  (cond ((null w) nil)
        ((consp (car w))
         (cons (drop-pluspp (car w)) (drop-pluspp (cdr w))))
        ((and (numberp (car w)) (plusp (car w)))
         (drop-pluspp (cdr w)))
        ((cons (car w) (drop-pluspp (cdr w))))))

> (drop-pluspp '(a 1 (-1 0) (0 (1))))
(A (-1 0) (0 NIL))
```

Решение 4.82.1

```
(defun drop-pluspp (w &aux (a (car w)) (d (cdr w)))
  (cond ((null w) nil)
        ((consp a) (cons (drop-pluspp a) (drop-pluspp d)))
        ((and (numberp a) (plusp a)) (drop-pluspp d))
        ((cons a (drop-pluspp d)))))

> (drop-pluspp '(a 1 (-1 0) (0 (1))))
(A (-1 0) (0 NIL))
```

Задача 4.83 sum-pairs.lisp

Написать функцию, которая принимает на вход последовательность, а возвращает последовательность с попарными суммами, т.е. (1 2 3 4 5) > (3 5 7 9).

Решение 4.83.1

```
(defun sum-pairs (w)
  (when (cdr w) (cons (+ (car w) (cadr w)) (sum-pairs (cdr w)))))

> (sum-pairs '(1 2 3 4 5))
(3 5 7 9)
> (sum-pairs '(1))
NIL
> (sum-pairs '())
NIL
```

Решение 4.83.2

```
(defun sum-pairs (w)
  (loop for a in w for b in (cdr w) collect (+ a b)))
```

```
> (sum-pairs '(1 2 3 4 5))
(3 5 7 9)
> (sum-pairs '(1))
NIL
> (sum-pairs '())
NIL
```

Задача 4.84 *remove-eventh.lisp*

Из заданного списка удалить каждый второй элемент, результирующий список переписать в обратном порядке.

Решение 4.84.1

```
(defun remove-eventh (w &optional ac)
  (if (null w) ac (remove-eventh (cddr w) (cons (car w) ac))))

> (remove-eventh '(1 2 3 4 5))
(5 3 1)
```

Задача 4.85 *elm-freq.lisp*

Для списка произвольного уровня вложенности выполнить анализ частоты появления каждого символа. Все встречающиеся символы и их число повторов объединить в списки, из которых сформировать список результатов.

Решение 4.85.1

```
(defun flat (w &optional acc)
  (cond ((null w) acc)
        ((atom w) (cons w acc))
        ((flat (car w) (flat (cdr w) acc)))))

(defun elm-freq (w)
  (mapcar #'(lambda (a) (list a (count a (flat w))))
          (delete-duplicates (flat w))))

> (elm-freq '(a ((a) (a (a)) b) (b (b) c) c))
((A 4) (B 3) (C 2))
```

Решение 4.85.2

```
(defun flat (w &optional acc)
  (cond ((null w) acc)
        ((atom w) (cons w acc))
        ((flat (car w) (flat (cdr w) acc)))))

(defun xelm (w &optional ac &aux (a (car w)))
  (cond ((null w) (reverse ac))
        ((find a ac :key #'car) (xelm (cdr w) ac))
```

```

      ((xelm (cdr w) (cons (list a (count a w)) ac))))))
(defun elm-freq (w)
  (xelm (flat w)))
> (elm-freq '(a ((a) (a (a)) b) (b (b) c) c))
((A 4) (B 3) (C 2))

```

Решение 4.85.3

```

(defun flat (w acc)
  (cond ((null w) acc)
        ((atom w) (cons w acc))
        ((flat (car w) (flat (cdr w) acc))))))
(defun xelm (w ac)
  (cond ((null w) (reverse ac))
        ((find (car w) ac :key #'car) (xelm (cdr w) ac))
        ((xelm (cdr w) (cons (list (car w) (count (car w) w)) ac))))))
(defun elm-freq (w)
  (xelm (flat w nil) nil))
> (elm-freq '(a ((a) (a (a)) b) (b (b) c) c))
((A 4) (B 3) (C 2))

```

Решение 4.85.4

```

(defun flat (w)
  (mapcan #'(lambda (a) (if (listp a) (flat a) `(,a))) w))
(defun xelm (w &aux (v (remove-duplicates w)))
  (loop for a in v collect (list a (count a w))))
(defun elm-freq (w)
  (xelm (flat w)))
> (elm-freq '(a ((a) (a (a)) b) (b (b) c) c))
((A 4) (B 3) (C 2))

```

Решение 4.85.5

```

(defun flat (w)
  (loop for a in w
        if (listp a) append (flat a)
        else collect a))
(defun xelm (w &aux (v (remove-duplicates w)))
  (loop for a in v
        collect (list a (count a w))))
(defun elm-freq (w)
  (xelm (flat w)))

```

```
> (elm-freq '(a ((a) (a (a)) b) (b (b) c) c))
((A 4) (B 3) (C 2))
```

Решение 4.85.6

```
(defun flat (w)
  (loop for a in w
        if (listp a) append (flat a)
        else collect a))

(defun xelm (w)
  (loop for a in (remove-duplicates w)
        collect (list a (count a w))))

(defun elm-freq (w)
  (xelm (flat w)))

> (elm-freq '(a ((a) (a (a)) b) (b (b) c) c))
((A 4) (B 3) (C 2))
```

Задача 4.86 *arrange-num-uscore-char.lisp*

Создать рекурсивную функцию для обработки списка, чтобы каждый атом списка состоял из упорядоченных цифр, знака подчеркивания и букв.

Решение 4.86.1

```
(defun arrange-num-uscore-char (w)
  (when w (cons (intern (sort (string (car w)) #'string<))
                (arrange-num-uscore-char (cdr w)))))

> (arrange-num-uscore-char '("b_a72" "dc5_1"))
(|27_ab| |15_cd|)
```

Задача 4.87 *double.lisp*

Дан список, элементы которого — натуральные числа. Умножить каждый элемент на 2.

Решение 4.87.1

```
(defun double (w)
  (when w (cons (* (car w) 2) (double (cdr w)))))

> (double '(1 2))
(2 4)
```

Решение 4.87.2

```
(defun double (w)
```



```
(loop for a in w collect (* a 2)))

> (double '(1 2))
(2 4)
```

Задача 4.88 rearrange.lisp

Задана последовательность из трех чисел. Если последовательность упорядочена по возрастанию, то упорядочить ее по убыванию и наоборот. разработать функцию двумя способами (if.....и cond.....). Входные значения передаются в качестве списка и выходные тоже.

Решение 4.88.1

```
(defun rearrange (w)
  (if (or (apply #'< w) (apply #'> w)) (reverse w) (w)))

> (rearrange '(1 2 3))
(3 2 1)
```

Решение 4.88.2

```
(defun rearrange (w)
  (cond ((or (apply #'< w) (apply #'> w)) (reverse w)) (w)))

> (rearrange '(1 2 3))
(3 2 1)
```

Задача 4.89 drop-plusp.lisp

Определите функцию (f w), результатом которой является список, получающийся после удаления на всех уровнях всех положительных элементов списка чисел w.

Решение 4.89.1

```
(defun drop-plusp (w)
  (cond ((null w) nil)
        ((consp (car w))
         (cons (drop-plusp (car w)) (drop-plusp (cdr w))))
        ((plusp (car w)) (drop-plusp (cdr w)))
        ((cons (car w) (drop-plusp (cdr w)))))

> (drop-plusp '(1 (-1)))
((-1))
```

Решение 4.89.2

```
(defun drop-plusp (w &aux (a (car w)))
  (cond ((null w) nil)
        ((consp a) (cons (drop-plusp a) (drop-plusp (cdr w)))))
```

```
((plusp a) (drop-plusp (cdr w)))
((cons a (drop-plusp (cdr w)))))
```

```
> (drop-plusp '(1 (-1)))
((-1))
```

Решение 4.89.3

```
(defun drop-plusp (w &aux (a (car w)) (d (cdr w)))
  (cond ((null w) nil)
        ((consp a) (cons (drop-plusp a) (drop-plusp d)))
        ((plusp a) (drop-plusp d))
        ((cons a (drop-plusp d)))))
```

```
> (drop-plusp '(1 (-1)))
((-1))
```

Решение 4.89.4

```
(defun drop-plusp (w)
  (lambda (a d)
    (cond ((null w) nil)
          ((consp a) (cons (drop-plusp a) (drop-plusp d)))
          ((plusp a) (drop-plusp d))
          ((cons a (drop-plusp d)))))
  (car w) (cdr w))
```

```
> (drop-plusp '(1 (-1)))
((-1))
```

Задача 4.90 rotate.lisp

Реализовать функцию, меняющую местами первый и последний элементы исходного списка

Решение 4.90.1

```
(defun rotate (w)
  (rotatef (car w) (nth (1- (length w)) w)) w)
```

```
> (rotate '(a b c d e f))
(F B C D E A)
```

Решение 4.90.2

```
(defun rotate (w)
  (nconc (last w) (butlast (cdr w)) (cons (car w) nil)))
```

```
> (rotate '(a b c d e f))
(F B C D E A)
```

Задача 4.91 wvd.lisp

Реализовать: нахождение разности списков $R=S1/S2$ (элементы всех уровней). Форма представления результата: результатом есть список, содержащий исходный и результирующие.

Решение 4.91.1

```
(defun flat (w &optional acc)
  (cond ((null w) acc)
        ((atom w) (cons w acc))
        ((flat (car w) (flat (cdr w) acc)))))

(defun wvd (w v)
  (list w v (set-difference (flat w) (flat v))))

> (wvd '(a b (c) d) '((c d) e))
((A B (C) D) ((C D) E) (A B))
```

Решение 4.91.2

```
(defun flat (w)
  (mapcan #'(lambda (a) (if (listp a) (flat a) `(,a))) w))

(defun wvd (w v)
  (list w v (set-difference (flat w) (flat v))))

> (wvd '(a b (c) d) '((c d) e))
((A B (C) D) ((C D) E) (A B))
```

Решение 4.91.3

```
(defun flat (w)
  (loop for a in w
        if (listp a) append (flat a)
        else collect a))

(defun wvd (w v)
  (list w v (set-difference (flat w) (flat v))))

> (wvd '(a b (c) d) '((c d) e))
((A B (C) D) ((C D) E) (A B))
```

Задача 4.92 cut-1.lisp

Дан список, элементы которого - натуральные числа. Вычесть из каждого элемента списка 1.

Решение 4.92.1

```
(defun cut-1 (w)
  (cond ((null w) nil)
```

```
((cons (1- (car w)) (cut-1 (cdr w)))))
```

```
> (cut-1 '(1 2))
(0 1)
```

Решение 4.92.2

```
(defun cut-1 (w)
  (if w (cons (1- (car w)) (cut-1 (cdr w))) nil))
```

```
> (cut-1 '(1 2))
(0 1)
```

Решение 4.92.3

```
(defun cut-1 (w)
  (when w (cons (1- (car w)) (cut-1 (cdr w)))))
```

```
> (cut-1 '(1 2))
(0 1)
```

Решение 4.92.4

```
(defun cut-1 (w)
  (mapcar #'1- w))
```

```
> (cut-1 '(1 2))
(0 1)
```

Задача 4.93 *plusp10.lisp*

Дан список, элементы которого натуральные числа. Заменить их на 1, если исходное число положительно, и на ноль в противном случае.

Решение 4.93.1

```
(defun plusp10 (w)
  (cond ((null w) nil)
        ((plusp (car w)) (cons 1 (plusp10 (cdr w))))
        ((cons 0 (plusp10 (cdr w)))))
```

```
> (plusp10 '(-1 1))
(0 1)
```

Задача 4.94 *list-t-nil.lisp*

Дан список с произвольными элементами. Заменить каждый элемент на Т, если элемент - список. И на NIL в противном случае.

Решение 4.94.1

```
(defun list-t-nil (w)
  (cond ((null w) nil)
        ((cons (listp (car w)) (list-t-nil (cdr w))))))

> (list-t-nil '(a (a b)))
(NIL T)
```

Решение 4.94.2

```
(defun list-t-nil (w)
  (when w (cons (listp (car w)) (list-t-nil (cdr w)))))

> (list-t-nil '(a (a b)))
(NIL T)
```

Задача 4.95 *even-plusp.lisp*

Дан список, описывающий вызов арифметической функции. Написать функцию, что в случае четности результата вычисления списка делает его проверку на позитивность, а в противном случае выдает сам список. Вычисление списка делать с помощью встроенной функции eval.

Решение 4.95.1

```
(defun even-plusp (w)
  (or (and (evenp (eval w)) (plusp (eval w))) w))

> (even-plusp '(+ 1 1))
T
> (even-plusp '(+ 1 2))
(+ 1 2)
```

Задача 4.96 *compress-word.lisp*

Построить программу «сжатия», назначение которой – преобразование английских слов в их "звуковой" код. Этот процесс предусматривает "сжатие" примерно одинаково звучащих слов в одинаковый их код – своего рода, аббревиатуру этих слов. Слова "сжимаются" в соответствии со следующими правилами:

- первая буква слова сохраняется;
- все последующие за ней гласные, а также буквы "h", "w" и "y" удаляются;
- двойные буквы заменяются одиночными;
- закодированное слово состоит не более чем из четырех букв, остальные буквы удаляются.

Напишите функцию (f w), которая выдает результат сжатия слова, представленного в виде списка букв w. Примеры: (f '(b a r r i n g t o n)) = (b r n g) и (f '(l l e w e l l y n)) =: (l n).

Решение 4.96.1

```

(defun compress-word (w &optional (ac `',(car w))) &aux (a (car w)))
  (cond ((or (null w) (= (length ac) 4)) (reverse ac))
        ((or (member a '(a e i o u y h w)) (eq a (car ac)))
         (compress-word (cdr w) ac))
        ((compress-word (cdr w) (cons a ac)))))

> (compress-word '(b a r r i n g t o n))
(B R N G)
> (compress-word '(a r r i n g t o n))
(A R N G)
> (compress-word '(1 l e w e 1 l y n))
(1 N)

```

Решение 4.96.2

```

(defun compress-word (w)
  (compress (cdr w) (list (car w))))

(defun compress (w ac &aux (a (car w)))
  (cond ((or (null w) (= (length ac) 4)) (reverse ac))
        ((or (member a '(a e i o u y h w)) (eq a (car ac)))
         (compress (cdr w) ac))
        ((compress (cdr w) (cons a ac)))))

> (compress-word '(b a r r i n g t o n))
(B R N G)
> (compress-word '(a r r i n g t o n))
(A R N G)
> (compress-word '(1 l e w e 1 l y n))
(1 N)

```

Решение 4.96.3

```

(defun compress-word (w)
  (compress (cdr w) `',(car w)))

(defun compress (w ac &aux (a (car w)))
  (if (or (null w) (= (length ac) 4))
      (reverse ac)
      (compress (cdr w)
                 (if (or (member a '(a e i o u y h w))
                         (eq a (car ac)))
                     ac
                     (cons a ac)))))

> (compress-word '(b a r r i n g t o n))
(B R N G)
> (compress-word '(a r r i n g t o n))
(A R N G)
> (compress-word '(1 l e w e 1 l y n))
(1 N)

```

Решение 4.96.4* (для XLISP)

```

(defun compress-word (w)
  (compress w nil))

(defun compress (w ac)
  (cond ((or (null w) (= (length ac) 4)) (reverse ac))
        ((or (mmb (car w) '("a" "e" "i" "o" "u" "y" "h" "w")) (eql (car w) (car ac)))
         (compress (cdr w) ac))
        ((compress (cdr w) (cons (car w) ac)))))

(defun mmb (a w)
  (cond ((null w) nil)
        ((eql a (car w)) (cons (car w) (cdr w)))
        ((mmb a (cdr w)))))

> (compress-word '("b" "a" "r" "r" "i" "n" "g" "t" "o" "n"))
("B" "R" "N" "G")
> (compress-word '("l" "l" "e" "w" "e" "l" "l" "y" "n"))
("l" "N")

```

Решение 4.96.5

```

(defun compress-word (w &optional (ac `(("a" "e" "i" "o" "u" "y" "h" "w")) &aux (a (car w)))
  (if (or (null w) (= (length ac) 4))
      (reverse ac)
      (compress-word (cdr w)
                      (if (or (member a '("a" "e" "i" "o" "u" "y" "h" "w"))
                              (eq a (car ac)))
                          ac
                          (cons a ac)))))

> (compress-word '("b" "a" "r" "r" "i" "n" "g" "t" "o" "n"))
("B" "R" "N" "G")
> (compress-word '("a" "r" "r" "i" "n" "g" "t" "o" "n"))
("A" "R" "N" "G")
> (compress-word '("l" "l" "e" "w" "e" "l" "l" "y" "n"))
("l" "N")

```

Решение 4.96.6

```

(defun compress-word (w &optional (ac `(("a" "e" "i" "o" "u" "y" "h" "w")) &aux (a (car w)))
  (if (and w (< (length ac) 4))
      (compress-word (cdr w)
                      (if (or (member a '("a" "e" "i" "o" "u" "y" "h" "w"))
                              (eq a (car ac)))
                          ac
                          (cons a ac)))
      (reverse ac)))

> (compress-word '("b" "a" "r" "r" "i" "n" "g" "t" "o" "n"))
("B" "R" "N" "G")
> (compress-word '("a" "r" "r" "i" "n" "g" "t" "o" "n"))
("A" "R" "N" "G")

```

```
> (compress-word '(1 1 e w e 1 1 y n))
(1 N)
```

Решение 4.96.7

```
(defun compress-word (w &optional (ac `(), (car w))) &aux (a (car w))
  (if (and w (null (cdddr ac)))
      (compress-word (cdr w)
                      (if (or (find a '(a e i o u y h w))
                              (eq a (car ac)))
                          ac
                          (cons a ac)))
      (reverse ac)))

> (compress-word '(b a r r i n g t o n))
(B R N G)
> (compress-word '(a r r i n g t o n))
(A R N G)
> (compress-word '(1 1 e w e 1 1 y n))
(1 N)
```

Задача 4.97 arrange.lisp

Отсортировать список вида ((A 3) (B 2) (C 5)) так чтобы получилось ((C 5) (A 3) (B 2)).

Решение 4.97.1

```
(defun arrange (w)
  (sort w #'(lambda (x y) (> (cadr x) (cadr y)))))

> (arrange '((A 3) (B 2) (C 5)))
((C 5) (A 3) (B 2))
```

Решение 4.97.2

```
(defun arrange (w)
  (sort w #'> :key #'second))

> (arrange '((A 3) (B 2) (C 5)))
((C 5) (A 3) (B 2))
```

Решение 4.97.3

```
(defun arrange (w)
  (sort w #'> :key #'cadr))

> (arrange '((A 3) (B 2) (C 5)))
((C 5) (A 3) (B 2))
```


Задача 4.98 arrange-vcnm.lisp

Заданный список произвольного уровня вложенности разбить на четыре списка: в первом – гласные, во втором – согласные, в третьем – цифры, в четвертом – все остальное. Первые три списка должны быть упорядочены и в них не должно быть повторов. Списки результатов объединить в один список списков.

Решение 4.98.1

```
(defun pose (w &optional v c n m &aux (a (car w)) (d (cdr w)))
  (cond ((null w) (list (sort-vcn v #'char<)
                        (sort-vcn c #'char<)
                        (sort-vcn n #'<) m))
        (t (case a
              ((#\a #\e #\i #\o #\y #\u)
               (pose d (push a v) c n m))
              ((#\b #\c #\d #\f #\g #\h #\j #\k #\l #\m #\n #\p
                #\q #\r #\s #\t #\v #\w #\x #\z)
               (pose d v (push a c) n m))
              ((0 1 2 3 4 5 6 7 8 9)
               (pose d v c (push a n) m))
              (otherwise (pose d v c n (push a m)))))))

(defun flat (w &optional acc)
  (cond ((null w) acc)
        ((atom w) (cons w acc))
        ((flat (car w) (flat (cdr w) acc)))))

(defun sort-vcn (w p)
  (sort (remove-duplicates w) p))

(defun arrange-vcnm (w)
  (pose (flat w)))

> (arrange-vcnm '(3 (2 #\a) #\b (3) #\b (#\u 1) #\c 1 #\c (!)))
((#\a #\u) (#\b #\c) (1 2 3) (!))
```

Решение 4.98.2

```
(defun pose (w v c n m)
  (cond ((null w) (list (sort-smbs v)
                        (sort-smbs c)
                        (sort-vcn n #'<)
                        m))
        ((member (car w) '(a e i o y u))
         (pose (cdr w) (cons (car w) v) c n m))
        ((member (car w) '(b c d f g h j k l m n p q r s t v w x z))
         (pose (cdr w) v (cons (car w) c) n m))
        ((member (car w) '(0 1 2 3 4 5 6 7 8 9))
         (pose (cdr w) v c (cons (car w) n) m))
        (t (pose (cdr w) v c n (cons (car w) m))))))
```

```

(defun sort-smbs (w)
  (sort (remove-dps w) 'string<))

(defun flat (w acc)
  (cond ((null w) acc)
        ((atom w) (cons w acc))
        ((flat (car w) (flat (cdr w) acc))))))

(defun sort-vcn (w p)
  (sort (remove-dps w) p))

(defun remove-dps (w)
  (cond ((null w) nil)
        ((member (car w) (cdr w)) (remove-dps (cdr w)))
        (t (cons (car w) (remove-dps (cdr w))))))

(defun arrange-vcnm (w)
  (pose (flat w nil) nil nil nil nil))

> (arrange-vcnm '(3 (2 a) b (3) b (u 1) c 1 c (!)))
((A U) (B C) (1 2 3) (!))

```

Решение 4.98.3

```

(defun pose (w &optional v c n m &aux (a (car w)) (d (cdr w)))
  (cond ((null w)
        (list (sort v #'string<) (sort c #'string<) (sort n #'<) m))
        ((member a '(a e i o y u))
         (pose d (pushnew a v) c n m))
        ((member a '(b c d f g h j k l m n p q r s t v w x z))
         (pose d v (pushnew a c) n m))
        ((member a '(0 1 2 3 4 5 6 7 8 9))
         (pose d v c (pushnew a n) m))
        (t (pose d v c n (push a m)))))

(defun flat (w &optional acc)
  (cond ((null w) acc)
        ((atom w) (cons w acc))
        ((flat (car w) (flat (cdr w) acc))))))

(defun arrange-vcnm (w)
  (pose (flat w)))

> (arrange-vcnm '(3 (2 a) b (3) b (u 1) c 1 c (!)))
((A U) (B C) (1 2 3) (!))

```

Решение 4.98.4

```

(defconstant vowels '(a e i o y u))
(defconstant consonants '(b c d f g h j k l m n p q r s t v w x z))
(defconstant digits '(0 1 2 3 4 5 6 7 8 9))

```

```
(defun flat (w)
  (mapcan #'(lambda (a) (if (listp a) (flat a) `(,a))) w))

(defun arrange-vcdm (w)
  (list (loop for v in vowels when (member v w) collect v)
        (loop for c in consonants when (member c w) collect c)
        (loop for d in digits when (member d w) collect d)))

(defun arrange-vcdm (w &aux (u (flat w)) (v (arrange-vcd u)))
  (nconc v (list (remove-if #'(lambda (a) (member a (flat v))) u))))

> (arrange-vcdm '(3 (2 a a a a) b (3) b (u 1) c 1 c (!)))
((A U) (B C) (1 2 3) (!))
```

Решение 4.98.5

```
(defconstant vowels '(a e i o y u))
(defconstant consonants '(b c d f g h j k l m n p q r s t v w x z))
(defconstant digits '(0 1 2 3 4 5 6 7 8 9))

(defun flat (w)
  (loop for a in w
    if (listp a) append (flat a)
    else collect a))

(defun arrange-vcd (w)
  (list (loop for v in vowels when (member v w) collect v)
        (loop for c in consonants when (member c w) collect c)
        (loop for d in digits when (member d w) collect d)))

(defun arrange-vcdm (w &aux (u (flat w)) (v (arrange-vcd u)))
  (nconc v (list (remove-if #'(lambda (a) (member a (flat v))) u))))

> (arrange-vcdm '(3 (2 a a a a) b (3) b (u 1) c 1 c (!)))
((A U) (B C) (1 2 3) (!))
```

Задача 4.99 *decompose.lisp*

Для списка произвольного уровня вложенности, состоящего из букв, цифр и строк, вычислить число объектов каждого вида и их процентное соотношение. Сам список превратить в список из объектов, в котором нет повторов.

Решение 4.99.1

```
(defun cns (w &optional ac (m 0) (c 0) (n 0) (s 0)
  &aux (a (car w)) (d (cdr w)))
  (cond ((null w) (values
    ac
    (format nil "chrs ~a ~a%, nums ~a ~a%, strs ~a ~a%"
      c (cnt c m)
      n (cnt n m)
      s (cnt s m))
    )
    (t (cns d))))
```

```

                                s (cnt s m)))
((numberp a) (if (member a d)
                  (cns d ac (1+ m) c (1+ n) c)
                  (cns d (push a ac) (1+ m) c (1+ n) s)))
((stringp a) (if (member a d :test #'equal)
                  (cns d ac (1+ m) c n (1+ s))
                  (cns d (push a ac) (1+ m) c n (1+ s))))
(t (if (member a d)
        (cns d ac (1+ m) (1+ c) n s)
        (cns d (push a ac) (1+ m) (1+ c) n s))))

```

```

(defun cnt (a m)
  (floor (* a 100) m))

```

```

(defun flat (w &optional acc)
  (cond ((null w) acc)
        ((atom w) (cons w acc))
        ((flat (car w) (flat (cdr w) acc)))))

```

```

(defun decompose (w)
  (cns (flat w)))

```

```

> (decompose '(1 (#\a) ("b" (1)) "b" (1)))
(1 "b" #\a)
"chrs 1 16%, nums 3 50%, strs 2 33%"

```

Решение 4.99.2

```

(defun cnt (a m)
  (floor (* a 100) m))

```

```

(defun flat (w)
  (mapcan #'(lambda (a) (if (listp a) (flat a) `(,a))) w))

```

```

(defun pose (w &aux
  (a (count-if #'symbolp w))
  (b (count-if #'numberp w))
  (c (count-if #'stringp w))
  (m (+ a b c)))
  (values (remove-duplicates w)
          (format nil "smbs ~a ~a%, nums ~a ~a%, strs ~a ~a%"
                  a (cnt a m)
                  b (cnt b m)
                  c (cnt c m)))))

```

```

(defun decompose (w)
  (pose (flat w)))

```

```

> (decompose '(A S D (E 4 R "mix") (Q W) ("string" M) 2 I M M 2 2 P))
(A S D E 4 R "mix" Q W "string" I M 2 P)
"smbs 12 66%, nums 4 22%, strs 2 11%"

```

Решение 4.99.3

```

(defun cnt (a m)
  (floor (* a 100) m))

(defun flat (w)
  (loop for a in w
        if (listp a) append (flat a)
        else collect a))

(defun pose (w &aux
             (a (count-if #'symbolp w))
             (b (count-if #'numberp w))
             (c (count-if #'stringp w))
             (m (+ a b c)))
  (values (remove-duplicates w)
          (format nil "smbs ~a ~a%, nums ~a ~a%, strs ~a ~a%"
                  a (cnt a m)
                  b (cnt b m)
                  c (cnt c m))))

(defun decompose (w)
  (pose (flat w)))

> (decompose '(A S D (E 4 R "mix") (Q W) ("string" M) 2 I M M 2 2 P))
(A S D E 4 R "mix" Q W "string" I M 2 P)
"smbs 12 66%, nums 4 22%, strs 2 11%"

```

Задача 4.100 *translate.lisp*

Создать программу "Переводчик", которая для заданного списка английских слов произвольного уровня вложенности будет формировать список соответствующих русских слов. Список результата не должен содержать вложений. Перевод осуществляется на основе заранее сформированного словаря (не менее 15 слов).

Решение 4.100.1

```

(defun flat (w &optional acc)
  (cond ((null w) acc)
        ((atom w) (cons w acc))
        ((flat (car w) (flat (cdr w) acc)))))

(defun late (a w)
  (cond ((null w) nil)
        ((equal a (caar w)) (cons (cadar w) (late a (cdr w))))
        ((late a (cdr w)))))

(defun tran (w v)
  (cond ((null w) nil)
        ((nconc (late (car w) v) (tran (cdr w) v)))))

(defun translate (w v)

```

```

(tran (flat w) v))

> (translate '(cakes (ale)) '((cakes pyrogy) (ale pyvo)))
(PYROGY PYVO)

```

Решение 4.100.2 (muLISP)

```

(setq slovar '((cakes pyrogy) (ale pivo) (phone telefon) (child rebe-
nok) (day den) (apple yabloko) (cat kot) (dog sobaka) (full polniy)))

(defun flatten (w)
  (typecase w
    (cons (mapcan #'flatten w)
      (t `(,w))))

(defun late (a w)
  (cond ((null w) nil)
        ((eql a (car (car w)))
         (cons (car (cdr (car w))) (late a (cdr w))))
        ((late a (cdr w)))))

(defun tran (w v)
  (cond ((null w) nil)
        ((append (late (car w) v) (tran (cdr w) v)))))

(defun translate (w v)
  (tran (flat w nil) v))

> (translate '(apple) (phone) (day)) slovar)
(YABLOKO TELEFON DEN)

```

Решение 4.100.3

```

(defun flat (w)
  (loop for a in w
        if (listp a) append (flat a)
        else collect a))

(defun late (a w)
  (cond ((null w) nil)
        ((equal a (caar w)) (cons (cadar w) (late a (cdr w))))
        ((late a (cdr w)))))

(defun tran (w v)
  (cond ((null w) nil)
        ((nconc (late (car w) v) (tran (cdr w) v)))))

(defun translate (w v)
  (tran (flat w) v))

> (translate '(cakes (ale)) '((cakes pyrogy) (ale pyvo)))
(PYROGY PYVO)

```

Решение 4.100.3

```
(defun flat (w)
  (if (atom w) `(),w) (mapcan #'flat w)))

(defun late (a w)
  (cond ((null w) nil)
        ((equal a (caar w)) (cons (cadar w) (late a (cdr w))))
        ((late a (cdr w)))))

(defun tran (w v)
  (cond ((null w) nil)
        ((nconc (late (car w) v) (tran (cdr w) v)))))

(defun translate (w v)
  (tran (flat w) v))

> (translate '(cakes (ale)) '((cakes pyrogy) (ale pyvo)))
(PYROGY PYVO)
```

Задача 4.101 delete-first-atom.lisp

Определить функцию удаления первого атома в списке.

Решение 4.101.1

```
(defun delete-first-atom (w)
  (cond ((null w) nil)
        ((atom (car w)) (cdr w))
        ((cons (car w) (delete-first-atom (cdr w)))))

> (delete-first-atom '((a b) c (d) e))
((A B) (D) E)
```

Решение 4.101.2

```
(defun delete-first-atom (w)
  (loop for a in w and d on w
        if (consp a) collect a into as
        else return (nconc as (cdr d))))

> (delete-first-atom '((a b) c (d) e))
((A B) (D) E)

> (time (delete-first-atom (nconc (loop for i from 1 to 1000000 col-
lect '(a)) '(x (z)))))
Real time: 7.8451204 sec.
Run time: 7.81565 sec.
Space: 16004224 Bytes
GC: 12, GC time: 0.780005 sec.
```

Решение 4.101.3

```
(defun del-lead-atom (w)
  (delete-if #'atom w :count 1))

> (delete-first-atom '((a b) c (d) e))
((A B) (D) E)

> (time (del-lead-atom (nconc (loop for i from 1 to 1000000 collect
  '(a)) '(x (z))))))
Real time: 7.7151184 sec.
Run time: 7.6908493 sec.
Space: 8129236 Bytes
GC: 8, GC time: 0.3276021 sec.
```

Задача 4.102 coin.lisp

Мы сами знаем, что она не имеет решения, — сказал Хунта, немедленно ошестиниваясь. — Мы хотим знать, как ее решать.

Аркадий Стругацкий, Борис Стругацкий
«Понедельник начинается в субботу»

Среди 12 монет есть одна фальшивая, отличающаяся от остальных по весу (неизвестно легче она или тяжелее). Необходимо тремя взвешиваниями определить фальшивую монету.

Решение 4.102.1 (алгоритм взвешиваний)

```
(defun coin (c1 c2 c3 c4 c5 c6 c7 c8 c9 c10 c11 c12)
  (cond ((= (+ c1 c2 c3 c4) (+ c5 c6 c7 c8))
    (cond ((= (+ c1 c2 c3) (+ c9 c10 c11))
      (if (< c1 c12) (rm "c12 H") (rm "c12 L"))))
    ((> (+ c1 c2 c3) (+ c9 c10 c11))
      (cond ((= c9 c10) (rm "c11 L"))
        ((< c9 c10) (rm "c9 L"))
        ((rm "c10 L"))))
    ((cond ((= c9 c10) (rm "c11 H"))
      ((< c9 c10) (rm "c9 H"))
      ((rm "c10 H")))))
    ((> (+ c1 c2 c3 c4) (+ c5 c6 c7 c8))
      (cond ((= (+ c1 c2 c5) (+ c3 c4 c12))
        (cond ((= c6 c7) (rm "c8 L"))
          ((< c6 c7) (rm "c6 L"))
          ((rm "c7 L"))))
        ((> (+ c1 c2 c5) (+ c3 c4 c12))
          (if (< c1 c2) (rm "c2 H") (rm "c1 H"))))
        ((cond ((= c3 c4) (rm "c5 L"))
          ((< c3 c4) (rm "c4 H"))
          ((rm "c3 H")))))
        ((cond ((= (+ c1 c2 c5) (+ c3 c4 c12))
          (cond ((= c6 c7) (rm "c8 H"))
            ((< c6 c7) (rm "c6 H"))
```



```

      ((rm "c7 H"))))
    (< (+ c1 c2 c5) (+ c3 c4 c12))
    (if (< c1 c2) (rm "c1 L") (rm "c2 L")))
  ((cond ((= c3 c4) (rm "c5 H"))
    (< c3 c4) (rm "c3 L"))
    ((rm "c4 L")))))

(defun rm (s)
  (format t "~a" s))

> (coin 2 2 1.5 2 2 2 2 2 2 2)
c3 L
NIL

```

Задача 4.103 intersection-nums.lisp

Написать функцию, которая из списка цифр и символов выбирает числа, принадлежащие другому списку.

Решение 4.103.1

```

(defun inter-nums (w v)
  (cond ((null w) nil)
        ((and (numberp (car w)) (member (car w) v))
         (cons (car w) (inter-nums (cdr w) v)))
        ((inter-nums (cdr w) v))))

> (inter-nums '(a b 1 2 3) '(b c 2 3 4))
(2 3)

```

Решение 4.103.2

```

(defun ns (w v &aux (a (car w)) (d (cdr w)))
  (cond ((null w) nil)
        ((and (realp a) (member a v)) (cons a (inter-nums d v)))
        ((inter-nums d v))))

> (inter-nums '(a 0 1) '(a 1 2))
(1)

```

Задача 4.104 remove-names.lisp

Написать программу, реализующую тел.справочник. в нем содержится информация о каждом абоненте: Ф,И,О, телефон, улица, номер дома, квартира. Есть список фактов:

```

("Ivanov" "Slava" 11111).
("Petrov" "Igor" 22222).
("Sokolov" "Fedor" 3333,).
("Sidorov" "Kostya" 4444).
("Vasiljev" "Dmitrij" 5555).
("Petrov" "Ivan" 6666).

```

```
("Ivanov" "Fedor" 7777).
```

Как реализовать функцию, которая бы убирала все одинаковые фамилии?

Решение 4.104.1

```
(defun remove-names (w)
  (remove-duplicates w :key #'car :test #'string=))

> (rd '(("Ivanov" "Slava" 11111)("Petrov" "Igor" 22222) ("Sokolov"
"Fedor" 3333) ("Sidorov" "Kostya" 4444) ("Vasiljev" "Dmitrij" 5555)
("Petrov" "Ivan" 6666) ("Ivanov" "Fedor" 7777)))
(("Sokolov" "Fedor" 3333) ("Sidorov" "Kostya" 4444) ("Vasiljev"
"Dmitrij" 5555) ("Petrov" "Ivan" 6666) ("Ivanov" "Fedor" 7777))
```

Решение 4.104.2

```
(defun remove-names (w)
  (cond ((null w) nil)
        ((member (caar w) (cdr w) :key #'car :test #'string=)
         (remove-names (cdr w)))
        ((cons (car w) (remove-names (cdr w))))))

> (remove-names '(("Ivanov" "Slava" 11111)("Petrov" "Igor" 22222)
("Sokolov" "Fedor" 3333) ("Sidorov" "Kostya" 4444) ("Vasiljev"
"Dmitrij" 5555) ("Petrov" "Ivan" 6666) ("Ivanov" "Fedor" 7777)))
(("Sokolov" "Fedor" 3333) ("Sidorov" "Kostya" 4444) ("Vasiljev"
"Dmitrij" 5555) ("Petrov" "Ivan" 6666) ("Ivanov" "Fedor" 7777))
```

Решение 4.104.3

```
(defun remove-names (w)
  (cond ((null w) nil)
        ((find (caar w) (cdr w) :key #'car :test #'string=)
         (remove-names (cdr w)))
        ((cons (car w) (remove-names (cdr w))))))

> (rd '(("Ivanov" "Slava" 11111)("Petrov" "Igor" 22222) ("Sokolov"
"Fedor" 3333) ("Sidorov" "Kostya" 4444) ("Vasiljev" "Dmitrij" 5555)
("Petrov" "Ivan" 6666) ("Ivanov" "Fedor" 7777)))
(("Sokolov" "Fedor" 3333) ("Sidorov" "Kostya" 4444) ("Vasiljev"
"Dmitrij" 5555) ("Petrov" "Ivan" 6666) ("Ivanov" "Fedor" 7777))
```

Задача 4.105 rooms&price.lisp

Есть следующий набор данных:

```
("Grand" "5" 9 420)
("Nobis" "5" 4 318)
("Berns" "4" 8 333)
("Diplomat" "4" 30, 320)
("Attache" "3" 8 164)
("General" "3" 7 207)
("Vanadis" "2" 5 99)
```

("Dialog" "2" 5 78),
 который представляет данные об отелях, где на третьем месте стоит количество свободных номеров, а на четвертом – цена за номер. Необходимо посчитать общее количество свободных номеров (сумма по третьей позиции), и среднюю цену за номер (среднее значение по четвертой).

Решение 4.105.1

```
(defun rooms&price (w)
  (list
    (reduce #'+ (mapcar #'caddr w))
    (float (/ (reduce #'+ (mapcar #'caddr w)) (length w)))))

> (rooms&price '(("Grand" "5" 9 420) ("Nobis" "5" 4 318) ("Berns" "4"
8 333) ("Diplomat" "4" 30 320) ("Attache" "3" 8 164) ("General" "3" 7
207) ("Vanadis" "2" 5 99) ("Dialog" "2" 5 78)))
(76 242.375)
```

Задача 4.106 edges-up.lisp

Написать функцию, которая для списка-аргумента формирует список-результат по правилу: если первый и последний элементы списка-аргумента – четные положительные целые числа, то включить в список-результат первым элементом – квадрат последнего элемента исходного списка, вторым – четвертую степень первого; в противном случае сформировать список из первого и последнего элементов.

Решение 4.106.1

```
(defun edges-up (a)
  (if (and (plus-even (car a))
    (plus-even (car (reverse a))))
    (list (expt (car (reverse a)) 2) (expt (car a) 4))
    (list (car a) (car (reverse a)))))

(defun plus-even (x)
  (and (realp x) (plusp x) (evenp x)))

> (edges-up '(28 4 5 6 7 82))
(6724 614656)
> (edges-up '(20 4 5 6 7 89))
(20 89)
```

Решение 4.106.2

```
(defun edges-up (list)
  ((lambda (edges)
    (if (equal (mapcar #'plus-even edges) '(T T))
      (reverse (mapcar #'expt edges '(4 2)))
      edges))
    (cons (car list) (last list)))))
```

```
(defun plus-even (x)
  (and (realp x) (plusp x) (evenp x)))

> (edges-up '(20 4 5 6 7 82))
(6724 160000)
> (edges-up '(20 4 5 6 7 81))
(20 81)
> (edges-up '(20 4 5 6 7 g))
(20 G)
```

Решение 4.106.3

```
(defun edges-up (w &aux (edges (cons (car w) (last w))))
  (if (every #'(lambda (x) (and (realp x) (plusp x) (evenp x)))
      edges)
      (nreverse (mapcar #'expt edges '(4 2)))
      edges))

> (edges-up '(20 4 5 6 7 82))
(6724 160000)
> (edges-up '(20 4 5 6 7 81))
(20 81)
> (edges-up '(20 4 5 6 7 g))
(20 G)
```

Задача 4.107 *sort-numbers.lisp*

Упорядочить цифры в каждом атоме списка a53r2 > a23r5.

Решение 4.107.1

```
(defun split-string (s)
  (map 'list #'string s))

(defun wsmb-to-wwstrs (w)
  (mapcar #'split-string (mapcar #'string w)))

(defun extract-nums (w)
  (delete-if-not
   #'(lambda (a) (parse-integer a :junk-allowed t)) w))

(defun sort-nums (w)
  (mapcar
   #'write-to-string(sort (mapcar
                          #'parse-integer w) #'<)))

(defun arrange (w)
  (mapcar
   #'(lambda (a)
      (xmint a (sort-nums (extract-nums (copy-list a))))
   w))
```

```

(defun xmint (w n)
  (cond ((null w) nil)
        ((parse-integer (car w) :junk-allowed t)
         (cons (car n) (xmint (cdr w) (cdr n))))
        ((cons (car w) (xmint (cdr w) n)))))

(defun join-strs (w)
  (cond ((null w) nil)
        ((concatenate 'string
                       (car w)
                       (join-strs (cdr w)))))

(defun sort-numbers (w)
  (mapcar
   #'intern
   (mapcar
    #'join-strs (arrange (wsmb-to-wwstrs w)))))

> (sort-numbers '(a53r2 v8n74))
(A23R5 V4N78)

```

Задача 4.108 compress-words.lisp

Дан текст. В каждом слове каждого предложения для повторяющихся литер произвести следующую замену: повторные вхождения литер удалить, к первому вхождению литеры приписать число вхождений литеры в слово. Пример: ((aaabb cccddd) (eeefggg hhkl)) преобразуется в ((a3b2 c4d3) (e3fg3 h2kl)).

Решение 4.108.1

```

(defun ss (w)
  (mapcar #'sw (mapcar #'string w)))

(defun sw (s)
  (map 'list #'string s))

(defun cn (w &optional (n 1) (acc ""))
  (cond ((null w) acc)
        ((equal (car w) (cadr w)) (cn (cdr w) (1+ n) acc))
        ((cn (cdr w) 1 (concatenate
                        'string acc (car w)
                        (if (> n 1) (write-to-string n) "")))))

(defun squeeze (w) (mapcar #'cn (ss w)))

(defun compress-words (w)
  (mapcar #'squeeze w))

> (compress-words '((aaabb cccddd) (eeefggg hhkl)))
(("A3B2" "C4D3") ("E3FG3" "H2KL"))

```

Решение 4.108.2

```

(defun ss (w)
  (mapcar #'sw (mapcar #'string w)))

(defun sw (s)
  (map 'list #'string s))

(defun cn (w &optional (n 1) (acc ""))
  (cond ((null w) acc)
        ((equal (car w) (cadr w)) (cn (cdr w) (1+ n) acc))
        ((cn (cdr w) 1 (concatenate
                  'string acc (car w)
                  (if (> n 1) (write-to-string n) ""))))))

(defun squeeze (w) (mapcar #'cn (ss w)))

(defun str-sym (w)
  (mapcar #'(lambda (a) (read-from-string a)) w))

(defun compress-words (w)
  (mapcar #'str-sym (mapcar #'squeeze w)))

> (compress-words '((aaabb cccddd) (eeefggg hhkl)))
((A3B2 C4D3) (E3FG3 H2KL))

```

Решение 4.108.3 (MuLisp)

```

(defun compress-words (w)
  (mapcar #'(lambda (x) (cn (unpack x) 1 "")) w))

(defun cn (w n ac)
  (cond ((null w) ac)
        ((equal (car w) (cadr w)) (cn (cdr w) (1+ n) ac))
        ((cn (cdr w)
              1
              (pack (list ac (car w) (if (> n 1) n "")))))))

> (compress-words '((aaabb cccddd) (eeefggg hhkl)))
((A3B2 C4D3) (E3FG3 H2KL))

```

Задача 4.109 odd>0-even>1.lisp

Дан список, элементы которого натуральные числа. Поменять их на число 1, если исходное число - четное, и на 0 в противном случае.

Решение 4.109.1

```

(defun odd>0-even>1 (w)
  (cond ((null w) nil)
        ((odd-0-even-1enp (car w))
         (cons 1 (odd>0-even>1 (cdr w)))))

```

```

      ((cons 0 (odd>0>even-1 (cdr w))))))
> (odd>0-even>1 '(2 3))
(1 0)

```

Задача 4.110 del-blast.lisp

Написать программу удаления предпоследнего элемента в списке.

Решение 4.110.1

```

(defun del-blast (w) (append (butlast w 2) (last w)))
> (del-blast '(a b c))
(A C)

```

Решение 4.110.2

```

(defun del-blast (w)
  (delete-if #'identity w :from-end t :start 1 :end 2))
> (del-blast '(a b c))
(A C)

```

Задача 4.111 sum-digits.lisp

Вычислить сумму цифр натуральных чисел. Вот пример ввода: (f 34 12 29 111 16 882), результат: (7 3 11 3 7 18).

Решение 4.111.1

```

(defun sn (n)
  (apply #'+
    (mapcar #'parse-integer
      (map 'list #'string (write-to-string n)))))
(defun sum-digits (&rest w)
  (mapcar #'sn w))
> (sum-digits 34 12 29 111 16 882)
(7 3 11 3 7 18)

```

Решение 4.111.2

```

(defun sn (n)
  (if (< n 10)
      n
      ((lambda (r) (+ r (sn (/ (- n r) 10)))) (mod n 10))))
(defun sum-digits (&rest w)
  (mapcar #'sn w))

```

```

> (sum-digits 34 12 29 111 16 882)
(7 3 11 3 7 18)

* Что делают mapcar, apply и reduce?

> (funcall #' + 1 1 1)
3

> (apply #' + '(1 1 1))
3

> (mapcar #'1+ '(1 2 3))
(2 3 4)

> (mapcar #' + '(1 2 3) '(1 2 3))
(2 4 6)

> (intersection '(a b) '(b c))
(B)

> (intersection '(a b c) '(b c d) '(c d a))
; Evaluation aborted

> (intersection (intersection '(a b c) '(b c d)) '(c d a))
(C)

> (reduce #'intersection '((a b c) (b c d) (c d a)))
(C)

```

Задача 4.112 *sort-words.lisp*

Записать предложение в порядке возрастания количества гласных букв в слове. Предложение - список слов.

Решение 4.112.1

```

(defun ws (w)
  (mapcar #'(lambda (a) (coerce (string a) 'list)) w))

(defun cn (w &optional (n 0))
  (cond ((null w) n)
        ((member (car w) '(#\A #\E #\I #\O #\U))
         (cn (cdr w) (1+ n)))
        ((cn (cdr w) n))))

(defun ez (w)
  (mapcar #'(lambda (e) (list e (cn e))) w))

(defun ts (w)
  (mapcar
   #'(lambda (a) (intern (coerce (car a) 'string)))

```



```
(sort w #'< :key #'cadr))

(defun sort-words (w)
  (ts (ez (ws w))))

> (sort-words '(green red))
(RED GREEN)
```

Задача 4.113 *list-caars.lisp*

Построить список, состоящий из первых элементов вложенных списков. Например ((A B) (C D) (A D)) -> (A C A).

Решение 4.113.1

```
(defun list-caars (w)
  (cond ((null w) nil)
        ((cons (caar w) (list-caars (cdr w))))))

> (list-caars '((A B) (C D) (A D)))
(A C A)
```

Решение 4.113.2

```
(defun list-caars (w)
  (if (null w) nil (cons (caar w) (list-caars (cdr w)))))

> (list-caars '((A B) (C D) (A D)))
(A C A)
```

Решение 4.113.3

```
(defun list-caars (w)
  (mapcar #'car w))

> (list-caars '((A B) (C D) (A D)))
(A C A)
```

Задача 4.114 *remove-odds.lisp*

Написать функцию, удаляющую из списка члены с нечетными номерами.

Решение 4.114.1

```
(defun remove-odds (w)
  (cond ((null (cdr w)) nil)
        ((cons (cadr w) (remove-odds (cddr w))))))

> (remove-odds '(1 2 3 4 5))
(2 4)
```

Задача 4.115 smbs-only.lisp

Опишите функцию, которая из исходного списка формирует список, содержащий только символьные атомы.

Решение 4.115.1

```
(defun smbs-only (w)
  (cond ((null w) nil)
        ((symbolp (car w)) (cons (car w) (smbs-only (cdr w))))
        ((smbs-only (cdr w)))))

> (smbs-only '(1 a))
(A)
```

Задача 4.116 list-products.lisp

Сформировать список из элементов: произведение всех элементов исходного списка, произведение $n-1$ последних элементов, произведение $n-2$ последних элементов и т. д. Пример: для списка `'(1 2 3 4 5 6)` результатом будет: `'(720 720 360 120 30 6)`.

Решение 4.116.1

```
(defun list-products (w)
  (cond ((null w) nil)
        ((cons (apply #'* w) (list-products (cdr w)))))

> (list-products '(1 2 3 4 5 6))
(720 720 360 120 30 6)
```

Решение 4.116.2

```
(defun list-products (w)
  (when w (cons (apply #'* w) (list-products (cdr w)))))

> (list-products '(1 2 3 4 5 6))
(720 720 360 120 30 6)
```

Решение 4.116.3

```
(defun list-products (w)
  (maplist #'(lambda (a) (reduce #'* a)) w))

> (list-products '(1 2 3 4 5 6))
(720 720 360 120 30 6)
```

Решение 4.116.4

```
(defun list-products (w)
  (maplist #'ma w))
```

```
(defun ma (v)
  (reduce #'* v))

> (list-products '(1 2 3 4 5 6))
(720 720 360 120 30 6)
```

Задача 4.117 select-sort.lisp

Написать функцию сортировки списка методом прямого выбора.

Решение 4.117.1

```
(defun select-sort (w)
  (when w (let ((m (reduce #'min w)))
    (cons m (select-sort (remove m w :count 1))))))

> (select-sort '(1 2 3 4 1 2 3 4))
(1 1 2 2 3 3 4 4)
```

Задача 4.118 selection-sort.lisp

Написать функцию сортировки списка методом прямого выбора. Встроенные функции max и min не использовать. Можно использовать только средства строго функционального языка программирования (без использования функций присваивания).

Решение 4.118.1

```
(defun selection-sort (w)
  (when w (cons (mini w) (selection-sort (remove (mini w) w)))))

(defun mini (w)
  (mi (cdr w) (car w)))

(defun mi (w m)
  (cond ((null w) m)
        ((< (car w) m) (mi (cdr w) (car w)))
        ((mi (cdr w) m))))

> (selection-sort '(5 4 7 1 3))
(1 3 4 5 7)
```

Решение 4.118.2

```
(defun selection-sort (w)
  (when w (cons (mini w) (selection-sort (remove (mini w) w)))))

(defun mini (w)
  (mi (cdr w) (car w)))

(defun mi (w m)
```

```
(if w (mi (cdr w) (if (< (car w) m) (car w) m)) m))

> (selection-sort '(5 4 7 1 3))
(1 3 4 5 7)
```

Решение 4.118.3

```
(defun selection-sort (w)
  (when w (cons (mi (cdr w) (car w))
                 (selection-sort (remove (mi (cdr w) (car w)) w))))))

(defun mi (w m)
  (if w (mi (cdr w) (if (< (car w) m) (car w) m)) m))

> (selection-sort '(5 4 7 1 3))
(1 3 4 5 7)
```

Задача 4.119 sign-to-word.lisp

В списке заменить все атомы-числа: положительные на слово "положительное", отрицательные на слово "отрицательное", нулевые на "нуль".

Решение 4.119.1

```
(defun sign-to-word (w)
  (mapcar #'(lambda (a)
              (if (numberp a)
                  (cond ((zerop a) "zero")
                        ((plusp a) "positive")
                        (t "negative"))
                a))
          w))

> (sign-to-word '(a -1 0 1))
(A "negative" "zero" "positive")
```

Задача 4.120 del-dkrange.lisp

Из произвольного списка удалить латинские буквы расположенные в алфавите между d и k.

Решение 4.120.1

```
(defun del-dkrange (w &aux (a (car w)) (d (cdr w)))
  (cond ((null w) nil)
        ((and (stringp a) (< 100 (char-code (char a 0)) 107))
         (del-dkrange d))
        ((cons a (del-dkrange d)))))

> (del-dkrange '("a" "e" "j" "z"))
("a" "z")
```

Решение 4.120.2

```
(defun del-dkrange (w)
  (delete-if
   #'(lambda (a)
       (and (stringp a) (< 100 (char-code (char a 0)) 107)))
   w))

> (del-dkrange '("a" "e" "j" "z"))
("a" "z")
```

Задача 4.121 rev&del-pairs.lisp

Дан список из десяти элементов, его надо инвертировать задом наперед и удалить из него равные элементы, которые стоят рядом. Т.е.:

2 3 3 4 4 5 6 3 4 4 - исходный список
 4 4 3 6 5 4 4 3 3 2 - инвертированный
 3 6 5 2 - полученный.

Решение 4.121.1

```
(defun rev&del-pairs (w &optional ac &aux (a (car w)))
  (cond ((null w) ac)
        ((eq a (cadr w)) (rev&del-pairs (cddr w) ac))
        ((rev&del-pairs (cdr w) (push a ac)))))

> (rev&del-pairs '(2 3 3 4 4 5 6 3 4 4))
(3 6 5 2)
```

Задача 4.122 even-odd.lisp

Из исходного списка получить два списка, содержащие четные и нечетные элементы исходного списка.

Решение 4.122.1

```
(defun even-odd (w &optional e o &aux (a (car w)))
  (cond ((null w) (list e o))
        ((evenp a) (even-odd (cdr w) (cons a e) o))
        ((even-odd (cdr w) e (cons a o)))))

> (even-odd '(3 45 458 3 4 3 434))
((434 4 458) (3 3 45 3))
```

Задача 4.123 minus-plus.lisp

Дан список из атомов (-7 1 8 -2 -9 4 9 -1). Нужно отсортировать элементы так, чтобы сначала в списке были отрицательные, а потом положительные элементы. Причем они должны быть в той же последовательности, что и в изначальном списке, т.е. должен получиться такой список (-7 -2

```
-9 -1 1 8 4 9).
```

Решение 4.123.1

```
(defun minus-plus (w &optional m p &aux (a (car w)))
  (cond ((null w) (nconc (nreverse m) (nreverse p)))
        ((plusp a) (minus-plus (cdr w) m (cons a p)))
        ((minus-plus (cdr w) (cons a m) p))))

> (minus-plus '(-7 1 8 -2 -9 4 9 -1))
(-7 -2 -9 -1 1 8 4 9)
```

Задача 4.124 *xlambda.lisp*

Расставить скобки, чтобы функция вернула список (g66 fc l):

```
((lambda (x y z) (cons (car (cdr x)) (cons (car (cdr y)) (cons (car
(cdr (cdr z))) nil))))) '(g55 g66 g777) '(9 fc i) 'n i l t d j (ii
jj))
```

Решение 4.124.1

```
> ((lambda (a b c) (cons (car (cdr a)) (cons (car (cdr b)) (cons (car
(cdr (cdr c))) nil))))) '(g55 g66 g777) '(9 fc i) '(n i l t d j (ii
jj))
(G66 FC L)
```

Решение 4.124.2

```
> ((lambda (a b c) (cons (cadr a) (cons (cadr b) (cons (caddr c)
nil))))) '(g55 g66 g777) '(9 fc i) '(n i l t d j (ii jj))
(G66 FC L)
```

Задача 4.125 *append-reverse.lisp*

Напишите функцию, которая соединяет два подсписка, причем оба подсписка должны быть обращены, например:

```
> (f '((a b) (c d)))
(B A D C)
```

Решение 4.125.1.

```
(defun append-reverse (w)
  (nconc (rv (car w)) (rv (cadr w))))

(defun rv (v &optional ac)
  (cond ((null v) ac)
        ((rv (cdr v) (push (car v) ac)))))

> (append-reverse '((a b) (c d)))
```

(B A D C)

Задача 4.126 *grow-numbers.lisp*

Дан список чисел ((x x x x...) (x x x x...)). Увеличить каждый элемент на единицу.

Решение 4.126.1

```
(defun grow-numbers (w)
  (mapcar #'(lambda (a) (loop for x in a collect (1+ x))) w))

> (grow-numbers '((1 2 3) (4 5 6) (7 8 9)))
((2 3 4) (5 6 7) (8 9 10))
```

Решение 4.126.2

```
(defun grow-numbers (w)
  (when w (cons (mapcar #'1+ (car w)) (grow-numbers (cdr w)))))

> (grow-numbers '((1 2 3) (4 5 6) (7 8 9)))
((2 3 4) (5 6 7) (8 9 10))
```

Решение 4.126.3

```
(defun grow-numbers (w)
  (mapcar #'(lambda (a) (mapcar #'1+ a)) w))

> (grow-numbers '((1 2 3) (4 5 6) (7 8 9)))
((2 3 4) (5 6 7) (8 9 10))
```

Задача 4.127 *quicksort.lisp*

Emacs has a very intuitive way of working with text files and was incredibly efficient at dealing with multiple files which when combined with the power of a full programming language of the calibre of lisp gave it almost a perfect collection of features that allowed it to be anything to anyone.

Anurag Mendhekar

Реализовать сортировку Хоара (англ. – Hoare) для списка.

Решение 4.127.1

```
(defun quicksort (w &aux (pivot (car w)))
  (if (> (length w) 1)
      (nconc (quicksort (remove-if-not #'(lambda (a) (< a pivot)) w))
              (remove-if-not #'(lambda (a) (= a pivot)) w))
      w))
```

```

        (quicksort (remove-if-not #'(lambda (a) (> a pivot)) w)))
    w))

> (quicksort '(7 4 9 2 4 3 8 1 0 6 9 3 4 2 5))
(0 1 2 2 3 3 4 4 4 5 6 7 8 9 9)

```

Решение 4.127.2

```

(defun quicksort (w &aux (pivot (car w)))
  (if (> (length w) 1)
      (nconc (quicksort (remove-if #'(lambda (a) (>= a pivot)) w))
              (remove-if #'(lambda (a) (/= a pivot)) w)
              (quicksort (remove-if #'(lambda (a) (<= a pivot)) w)))
      w))

> (quicksort '(7 4 9 2 4 3 8 1 0 6 9 3 4 2 5))
(0 1 2 2 3 3 4 4 4 5 6 7 8 9 9)

```

Решение 4.127.3

```

(defun quicksort (w &aux (pivot (car w)))
  (if (cdr w)
      (nconc (quicksort (remove-if #'(lambda (a) (>= a pivot)) w))
              (remove-if #'(lambda (a) (/= a pivot)) w)
              (quicksort (remove-if #'(lambda (a) (<= a pivot)) w)))
      w))

> (quicksort '(7 4 9 2 4 3 8 1 0 6 9 3 4 2 5))
(0 1 2 2 3 3 4 4 4 5 6 7 8 9 9)

```

Решение 4.127.4

```

(defun quicksort (w &aux (pivot (car w)))
  (if (cdr w)
      (nconc (quicksort (remove-if-not #'(lambda (a) (< a pivot)) w))
              (remove-if-not #'(lambda (a) (= a pivot)) w)
              (quicksort (remove-if-not #'(lambda (a) (> a pivot)) w)))
      w))

> (quicksort '(7 4 9 2 4 3 8 1 0 6 9 3 4 2 5))
(0 1 2 2 3 3 4 4 4 5 6 7 8 9 9)

```

Решение 4.127.5

```

(defun quicksort (w)
  (if (cdr w)
      (macrolet ((pivot (test) `(remove (car w) w :test-not #',test)))
        (nconc (quicksort (pivot >)) (pivot =) (quicksort (pivot <)))))
      w))

> (quicksort '(8 4 5 0 2 3 9 5 2 3 9 5 4 2 8 3 7))
(0 2 2 2 3 3 3 4 4 5 5 5 7 8 8 9 9)

```


Задача 4.128 swap-elms.lisp

Определить функцию, которая меняет местами соседние элементы списка.

Решение 4.128.1

```
(defun swap-elms (w)
  (if (cdr w)
      (cons (cadr w)
            (cons (car w)
                  (swap-elms (cddr w)))))
      w))

> (swap-elms '(1 2 3 4 5))
(2 1 4 3 5)
```

Задача 4.129 sin>cos-sqrt>abs.lisp

Заменить в заданном математическом предложении все вхождения sin на cos и sqrt на abs.

Решение 4.129.1

```
(defun sin>cos-sqrt>abs (w &aux (a (car w)) (d (cdr w)))
  (cond ((null w) nil)
        ((listp a) (cons (sin>cos-sqrt>abs a) (sin>cos-sqrt>abs d)))
        ((eql a 'sin) (cons 'cos (sin>cos-sqrt>abs d)))
        ((eql a 'sqrt) (cons 'abs (sin>cos-sqrt>abs d)))
        ((cons a (sin>cos-sqrt>abs d)))))

> (sin>cos-sqrt>abs '((sin (sqrt 90)) + (sin (sqrt 90))))
((COS (ABS 90)) + (COS (ABS 90)))
```

Задача 4.130 popular-names.lisp

Имеется список студентов с информацией: фамилия, имя, отчество, пол, возраст и курс. Определить самое распространенное мужское и женское имя..

Решение 4.130.1

```
(defparameter w '((ivanov ivan ivanovich m 20 2)
                  (petrov ivan ivanovich m 20 2)
                  (petrova anna ivanovna f 20 2)
                  (petrova inna ivanovna f 20 2)
                  (ivanova anna ivanovna f 20 2)))

(defun popular-names (w &optional m f)
  (cond ((null w) (list (caar (sort m #'> :key #'cadr))
                        (caar (sort f #'> :key #'cadr))))
        ((member (cadar w) m :key #'car) (popular-names (cdr w) m f))
```

```

((member (cadar w) f :key #'car) (popular-names (cdr w) m f))
((eq1 (caddr (car w)) 'm)
 (popular-names (cdr w)
  (cons (list (cadar w)
              (count (cadar w) w :key #'cadr))
        m)
  f))
((popular-names (cdr w)
 m
 (cons (list (cadar w)
             (count (cadar w) w :key #'cadr))
       f))))))

```

```

> (time (popular-names w))
Real time: 0.0 sec.
Run time: 0.0 sec.
Space: 96 Bytes
(IVAN ANNA)

```

Решение 4.130.2

```

(defparameter w '((ivanov ivan ivanovich m 20 2)
                  (petrov ivan ivanovich m 20 2)
                  (petrova anna ivanovna f 20 2)
                  (petrova inna ivanovna f 20 2)
                  (ivanova anna ivanovna f 20 2)))

(defun drop-one (w &optional acc ac)
  (cond ((null w) acc)
        ((member (car w) ac) (drop-one (cdr w) (cons (car w) acc) ac))
        ((drop-one (cdr w) acc (cons (car w) ac)))))

(defun max-occur (w)
  (if (cdr w)
      (max-occur (drop-one w))
      (car w)))

(defun names-s (w p)
  (cond ((null w) nil)
        ((eq (cadar w) p) (cons (caar w) (names-s (cdr w) p)))
        ((names-s (cdr w) p))))

(defun names (w)
  (mapcar #'(lambda (a) (list (cadr a) (caddr a))) w))

(defun popular-names (w)
  (mapcar #'max-occur (list (names-s (names w) 'm)
                            (names-s (names w) 'f)))))

> (popular-names w)
(IVAN ANNA)

```

Задача 4.131 list-data.lisp

Написать функцию, которая из списка (Ann 20 160cm) строит список ((name Anna) (age 20) (height 151cm))

Решение 4.131.1

```
(defun list-data (w)
  (list (list 'name (car w))
        (list 'age (cadr w))
        (list 'height (caddr w))))

> (list-data '(Ann 20 160cm))
((NAME ANN) (AGE 20) (HEIGHT 160CM))
```

Решение 4.131.2

```
(defun list-data (w &optional (v '(name age height)))
  (loop for a in w
        for b in v
        collect (list b a)))

> (list-data '(Ann 20 160cm))
((NAME ANN) (AGE 20) (HEIGHT 160CM))
```

Задача 4.132 read-numbers.lisp

Прочитать файл целых чисел и вывести числа на экран.

Решение 4.132.1

```
(defun read-numbers (path)
  (with-open-file (s path :direction :input)
    (do ((line (read-line s nil :eof) (read-line s nil :eof)))
        ((eql line :eof))
        (format t "~a~%" line))))

> (read-numbers "c:/numbers.txt")
0 1 2 3 4
5 6 7 8 9
NIL
```

Задача 4.133 cons-n.lisp

Сформировать из простого списка ассоциативный, поставив ключи в виде последовательных чисел.

Решение 4.133.1

```
(defun cons-n (w &optional (n 1))
  (when w (cons (cons n (car w)) (cons-n (cdr w) (1+ n)))))
```

```
> (cons-n '(11 12 13 14))
((1 . 11) (2 . 12) (3 . 13) (4 . 14))
```

Решение 4.133.2

```
(defun cons-n (w)
  (loop for a in w
        for b upfrom 1
        collect (cons a b)))

> (cons-n '(11 12 13 14))
((1 . 11) (2 . 12) (3 . 13) (4 . 14))
```

Решение 4.133.3

```
(defun cons-n (w)
  (loop for a in w
        for b from 1
        collect (cons a b)))

> (cons-n '(11 12 13 14))
((1 . 11) (2 . 12) (3 . 13) (4 . 14))
```

Задача 4.134 *drop-deep-duplicates.lisp*

Определить функцию, которая удаляет дубликаты атомов в многоуровневом списке.

Решение 4.134.1

```
(defun flat (w &optional acc)
  (cond ((null w) acc)
        ((atom w) (cons w acc))
        ((flat (car w) (flat (cdr w) acc)))))

(defun drop-duplicates (w)
  (cond ((null w) nil)
        ((member (car w) (cdr w)) (drop-duplicates (cdr w)))
        ((cons (car w) (drop-duplicates (cdr w)))))

(defun drop-deep-duplicates (w)
  (drop-duplicates (flat w)))

> (drop-deep-duplicates '((1) 2 (3 (4 1) 2) ((3)) 4 ((1) 2)))
(3 4 1 2)
```

Решение 4.134.2

```
(defun flat (w)
  (mapcan #'(lambda (a) (if (listp a) (flat a) `(,a))) w))
```

```
(defun drop-duplicates (w)
  (cond ((null w) nil)
        ((member (car w) (cdr w)) (drop-duplicates (cdr w)))
        ((cons (car w) (drop-duplicates (cdr w))))))

(defun drop-deep-duplicates (w)
  (drop-duplicates (flat w)))

> (drop-deep-duplicates '((1) 2 (3 (4 1) 2) ((3)) 4 ((1) 2)))
(3 4 1 2)
```

Решение 4.134.3

```
(defun flat (w)
  (loop for a in w
        if (listp a) append (flat a)
        else collect a))

(defun drop-deep-duplicates (w)
  (remove-duplicates (flat w)))

> (drop-deep-duplicates '((1) 2 (3 (4 1) 2) ((3)) 4 ((1) 2)))
(3 4 1 2)
```

Задача 4.135 *meet.lisp*

Определить функцию, которая преобразует список (a (b (c (d (e))))) к виду (a b c d e).

Решение 4.135.1

```
(defun meet (w)
  (when w (cons (car w) (meet (cadr w)))))

> (meet '(a (b (c (d (e)))))
(A B C D E)
```

Задача 4.136 *show-ironian-names.lisp*

Задать структуру: фамилия, город, улица, дом, возраст. Определить функцию, которая возвращает фамилии студентов, живущих по одному адресу в разных городах.

Решение 4.136.1

```
(defstruct student name city str num apt age)

(defun sp (n)
  (cond ((= n 0) nil)
        (t(cons (make-student
                    :name (read)
                    :city (read)
                    :str (read)
                    :num (read)
                    :apt (read)
                    :age (read))
                  (sp (1- n)))))
```

```

:city (read)
:str (read)
:num (read)
:apt (read)) (sp (- n 1))))))

(defun check-address (a b)
  (and (eq (student-str a) (student-str b))
        (eq (student-num a) (student-num b))
        (eq (student-apt a) (student-apt b))))

(defun ironian (a w)
  (cond ((null w) nil)
        ((check-address a (car w)) t)
        ((ironian a (cdr w)))))

(defun show-ironian-names (w)
  (loop for a in w when (ironian a (remove a w))
        collect (student-name a)))

> (setq w (sp 3))
ivanov
rostov
lenina
10
11
petrov
noginsk
lenina
100
111
petrova
podolsk
lenina
10
11
(#S(STUDENT :NAME IVANOV :CITY ROSTOV :STR LENINA :NUM 10 :APT 11)
 #S(STUDENT :NAME PETROV :CITY NOGINSK :STR LENINA :NUM 100 :APT 111)
 #S(STUDENT :NAME PETROVA :CITY PODOLSK :STR LENINA :NUM 10 :APT 11))
> (show-ironian-names w)
(IVANOV PETROVA)

```

Решение 4.136.2

```

(defstruct student name city str num apt age)

(defun sp (n)
  (cond ((= n 0) nil)
        (t (cons (make-student
                     :name (read)
                     :city (read)
                     :str (read)
                     :num (read)
                     :apt (read)

```

```

      :age (read)) (sp (- n 1))))))

(defun check-address (a b)
  (and (eq (student-str a) (student-str b))
        (eq (student-num a) (student-num b))
        (eq (student-apt a) (student-apt b))))

(defun ironian (a w)
  (cond ((null w) nil)
        ((check-address a (car w)) t)
        ((ironian a (cdr w)))))

(defun remove-elm (a w)
  (cond ((null w) nil)
        ((equalp (car w) a) (remove-elm a (cdr w)))
        ((cons (car w) (remove-elm a (cdr w)))))

(defun show-ironian (w v)
  (cond ((null w) nil)
        ((ironian (car w) (remove-elm (car w) v))
         (cons (student-name (car w)) (show-ironian (cdr w) v)))
        ((show-ironian (cdr w) v))))

(defun show-ironian-names (w)
  (show-ironian w w))

> (setq w (sp 3))
ivanov
rostov
lenina
10
11
20
petrov
noginsk
lenina
100
111
20
petrova
podolsk
lenina
10
11
20
(#S(STUDENT :NAME IVANOV :CITY ROSTOV :STR LENINA :NUM 10 :APT 11 :AGE
20)
 #S(STUDENT :NAME PETROV :CITY NOGINSK :STR LENINA :NUM 100 :APT 111
:AGE 20)
 #S(STUDENT :NAME PETROVA :CITY PODOLSK :STR LENINA :NUM 10 :APT 11
:AGE 20))
> (show-ironian-names w)
(IVANOV PETROVA)

```

Задача 4.137 massp.lisp

Определите функцию, которая выдает список всех элементов, удовлетворяющих некоторому предикату и встречающихся в исходном списке более n раз.

Решение 4.137.1

```
(defun massp (n w p)
  (labels ((mass (n v p)
            (cond ((null v) nil)
                  ((> (count (car v) v) n)
                   (cons (car v) (mass n (delete (car v) v) p)))
                  ((mass n (cdr v) p))))))
    (mass n (delete-if-not p w) p)))

> (massp 2 '(1 2 3 1 3 2 3 2 2) #'evenp)
(2)
> (massp 2 '(1 2 5 3 1 3 5 2 3 2 5 2) #'oddp)
(5 3)
```

Решение 4.137.2

```
(defun massp (n w p)
  (labels ((mass (n v p)
            (when v (if (> (count (car v) v) n)
                        (cons (car v) (mass n (delete (car v) v) p))
                        (mass n (cdr v) p))))))
    (mass n (remove-if-not p w) p)))

> (massp 2 '(1 2 3 1 3 2 3 2 2) #'evenp)
(2)
> (massp 2 '(1 2 5 3 1 3 5 2 3 2 5 2) #'oddp)
(5 3)
```

Решение 4.137.2* (для XLISP)

```
(defun massp (n w p) (mass n w p nil))

(defun mass (n w p v)
  (cond ((null w) (reverse v))
        ((and (funcall p (car w))
               (not (mmb (car w) v))
               (> (cnt (car w) w) n))
         (mass n (cdr w) p (cons (car w) v)))
        ((mass n (cdr w) p v))))

(defun mmb (a w)
  (cond ((null w) nil)
        ((eql a (car w)) (cons (car w) (cdr w)))
        ((mmb a (cdr w)))))
```



```
(defun cnt (a w)
  (cond ((null w) 0)
        ((eq1 a (car w)) (+ 1 (cnt a (cdr w))))
        ((cnt a (cdr w)))))

> (massp 2 '(1 2 3 1 3 2 3 2 2) #'evenp)
(2)
> (massp 2 '(1 2 5 3 1 3 5 2 3 2 5 2) #'oddp)
(5 3)
```

Задача 4.138 word-right-p.lisp

Даны список строк и буква. Определить функцию, которая возвращает список строк оканчивающихся на эту букву.

Решение 4.138.1

```
(defun word-right-p (r w)
  (loop for a in w when (eq (car (last (coerce a 'list))) r)
        collect a))

> (word-right-p #\z '("aaab" "aaac" "aaaz" "bbbc" "bbbz"))
("aaaz" "bbbz")
```

Решение 4.138.2 * дополнительно вывести количество слов

```
(defun word-right (r w)
  (loop for a in w when (eq (car (last (coerce a 'list))) r)
        collect a into ws
        finally (return (values ws (length ws)))))

> (word-right #\z '("aaab" "aaac" "aaaz" "bbbc" "bbbz"))
("aaaz" "bbbz")
2
```

Решение 4.138.3 * дополнительно вывести количество слов

```
(defun word-right (r w)
  (loop for a in w when (eq (car (last (coerce a 'list))) r)
        collect a into ws
        finally (return (list ws (length ws)))))

> (word-right #\z '("aaab" "aaac" "aaaz" "bbbc" "bbbz"))
(("aaaz" "bbbz") 2)
```

Задача 4.139 ordered-chaos.lisp

Список целых чисел вернуть без изменений, если он упорядочен по возрастанию или убыванию. В противном случае: каждый четный по счету элемент списка утроить, а каждый элемент, стоящий на нечетном месте и кратный четырем, удалить.

Решение 4.139.1

```
(defun chaos (w &aux (a (car w)) (d (cdr w)))
  (cond ((null w) nil)
        ((zerop (mod a 4)) (cons (* (car d) 3) (chaos (cdr d))))
        ((cons a (cons (* (car d) 3) (chaos (cdr d)))))))

(defun ordered (w) (or (null w) (apply #'<= w) (apply #'>= w)))

(defun ordered-chaos (w) (if (ordered w) w (chaos w)))

> (ordered-chaos '(1 7 3 4 8 6 12 8))
(1 21 3 12 18 24)
```

Задача 4.140 *un-even.lisp*

All things considered, Emacs is better simply because it is older at being extensible than any other editor on the planet.

Anurag Mendhekar

Проверить, является ли первый элемент списка четным числом. Если является, то вернуть исходный список, выполнив замену первого элемента на ближайшее большее нечетное число, иначе удалить первый элемент.

Решение 4.140.1

```
(defun un-even (w)
  (if (oddp (car w)) (cdr w) (cons (1+ (car w)) (cdr w))))

> (un-even '(2 1))
(3 1)
```

Задача 4.141 *minus-odd.lisp*

Проверить, является ли второй элемент списка отрицательным нечетным числом. Если является, то вернуть исходный список без последнего элемента, иначе поменять местами первый и второй элементы исходного списка.

Решение 4.141.1

```
(defun minus-odd (w)
  (if (and (minusp (cadr w)) (oddp (cadr w)))
      (butlast w)
      (cons (cadr w) (cons (car w) (cddr w)))))

> (minus-odd '(1 2))
(2 1)
```

```
> (minus-odd '(2 -1))
(2)
```

Задача 4.142 snow-flake.lisp

Создать функцию, которая сортирует список. Например, переделывает из списка (3 2 1 5 6 0 4) список (5 3 1 0 2 4 6). То есть, находит самое малое значение, помещает его в середину списка, а другие элементы списка раскладывает по убыванию и возрастанию вокруг наименьшего значения.

Решение 4.142.1

```
(defun snow-flake (w &optional ac &aux (v (sort w #'<)))
  (if w (snow-flake (cdr v) (cons (car v) (reverse ac)))
      (reverse ac)))

> (time (snow-flake '(3 2 1 5 6 0 4)))
Real time: 0.0 sec.
Run time: 0.0 sec.
Space: 376 Bytes
(5 3 1 0 2 4 6)
```

Решение 4.142.2

```
(defun snow (w &optional ac)
  (if w (snow (cdr w) (cons (car w) (reverse ac)))
      (reverse ac)))

(defun snow-flake (w)
  (snow (sort w #'<)))

> (time (snow-flake '(3 2 1 5 6 0 4)))
Real time: 0.0 sec.
Run time: 0.0 sec.
Space: 304 Bytes
(5 3 1 0 2 4 6)
```

Решение 4.142.3

```
(defun snow (w &optional ac)
  (if w (snow (cdr w) (cons (car w) (nreverse ac)))
      (nreverse ac)))

(defun snow-flake (w)
  (snow (sort w #'<)))

> (time (snow-flake '(3 2 1 5 6 0 4)))
Real time: 0.0 sec.
Run time: 0.0 sec.
Space: 80 Bytes
(5 3 1 0 2 4 6)
```

Задача 4.143 max>min-substitute.lisp

Emacs had Lisp long before Vim did. That's how Emacs lisp got its reputation as a powerful tool. It was the first editor to have a fully-powered programming language inside it, allowing writing modes. That expanded to include shell interaction, a primitive IDE, mail, a browser... in fact, an early windowed operating system, long before there were windows and mice and GUIs.

Joshua Engel

Дан массив целых чисел, если минимальный элемент меньше максимального в два раза, то все элементы, предшествующие максимальному элементу установить равными минимальному элементу.

Решение 4.143.1

```
(defun max>min (w n x)
  (if (= (car w) x) w
      (cons n (max>min (cdr w) n x))))

(defun max>min-check (w n x)
  (if (= (* n 2) x) (max>min w n x) w))

(defun max>min-substitute (w)
  (max>min-check w (apply #'min w) (apply #'max w)))

> (max>min-substitute '(1 5 3 2 1 2 3 5 6 12))
(1 5 3 2 1 2 3 5 6 12)
> (max>min-substitute '(5 5 3 4 4 3 5 6 3 4 5))
(3 3 3 3 3 3 6 3 4 5)
```

Задача 4.144 insert-asterisc.lisp

Определите функцию, вставляющую перед каждым элементом списка, обладающим определенным свойством, символ *

Решение 4.144.1

```
(defun insert-asterisc (p w &aux (a (car w)) (d (cdr w)))
  (cond ((null w) nil)
        ((funcall p a) (cons '* (cons a (insert-asterisc p d))))
        ((cons a (insert-asterisc p d)))))

> (insert-asterisc #'plusp '(-1 2 -3 -4 -5 -6 7 8 9))
(-1 * 2 -3 -4 -5 -6 * 7 * 8 * 9)
> (insert-asterisc #'(lambda (a) (not (minusp a))) '(0 -1 2 -3 -4))
(* 0 -1 * 2 -3 -4)
> (insert-asterisc #'evenp '(-1 2 -3 -4 -5 -6 7 8 9))
```

```
(-1 * 2 -3 * -4 -5 * -6 7 * 8 9)
```

Задача 4.145 *pair-unpaired.lisp*

Составить функцию, группирующую смежные элементы списка в подсписки, которые должны содержать по два элемента (последний подсписок может содержать один элемент).

Решение 4.145.1

```
(defun pair-unpaired (w)
  (if (cdr w)
      (cons (list (car w) (cadr w))
            (pair-unpaired (cddr w)))
      (when w (list w))))

> (pair-unpaired '(1 2 3 4 5))
((1 2) (3 4) (5))
> (pair-unpaired '(1 2 3 4))
((1 2) (3 4))
> (pair-unpaired '())
NIL
```

Решение 4.145.2

```
(defun pair-unpaired (w)
  (loop for (a b) on w by #'cddr
        if b collect `(,a ,b)
        else collect `( ,a)))

> (pair-unpaired '(1 2 3 4 5))
((1 2) (3 4) (5))
> (pair-unpaired '(1 2 3 4))
((1 2) (3 4))
> (pair-unpaired '())
NIL
```

Задача 4.146 *map-repeat.lisp*

На входе список и число n , создать новый список, в котором каждый элемент из исходного повторяется n раз

Решение 4.146.1

```
(defun repeat (a n)
  (loop repeat n collect a))

(defun map-repeat (w n)
  (mapcan #'(lambda (a) (repeat a n)) w))

> (map-repeat '(1 2 3) 2)
```

```
(1 1 2 2 3 3)
```

Решение 4.146.2

```
(defun map-repeat (w n)
  (mapcan #'(lambda (a) (loop for i from 1 to n collect a))
    w))

> (map-repeat '(1 2 3) 2)
(1 1 2 2 3 3)
```

Задача 4.147 surnames-common-ages.lisp

Имеется список студентов с информацией: фамилия, имя, отчество, пол, возраст, и курс. Определить фамилии тех студенток, возраст которых является самым распространённым.

Решение 4.147.1

```
(defstruct student surname name1 name2 sex age year)

(defun res (n)
  (when (> n 0)
    (cons (make-student :surname (read)
                        :name1 (read)
                        :name2 (read)
                        :sex (read)
                        :age (read)
                        :year (read))
      (res (1- n)))))

(defun surname-age (w)
  (mapcan #'(lambda (a)
    (when (eq (student-sex a) 'f)
      (list (list (student-surname a)
                  (student-age a)))))
    w))

(defun ages (w &optional ac)
  (if w (ages (cdr w) (pushnew (caddr w) ac)) ac))

(defun common-age (w v)
  (sort (mapcar
    #'(lambda (a) (list a (count a v :key #'cadr)))
    w) #'> :key #'cadr))

(defun age (w)
  (caar (common-age (ages (surname-age w)) (surname-age w))))

(defun surnames (w n)
  (cond ((null w) nil)
```

```

      ((eq (cadar w) n)
       (cons (caar w)
              (surnames (cdr w) n)))
      ((surnames (cdr w) n))))

(defun surnames-common-ages (w)
  (surnames (surname-age w) (age w)))

> (setq w (res 3))
ivanova
anna
savishna
f
20
3
petrova
olga
andreevna
f
21
3
semenova
alina
sergeevna
f
20
2
(#S(STUDENT :SURNAME IVANOVA :NAME1 ANNA :NAME2 SAVISHNA :SEX F :AGE
20 :YEAR 3)
 #S(STUDENT :SURNAME PETROVA :NAME1 OLGA :NAME2 ANDREEVNA :SEX F :AGE
21 :YEAR 3)
 #S(STUDENT :SURNAME SEMENOVA :NAME1 ALINA :NAME2 SERGEEVNA :SEX F
:AGE 20 :YEAR 2))
> (surnames-common-ages w)
(IVANOVA SEMENOVA)

```

Задача 4.148 *xine-twins.lisp*

Определить функцию, которая полностью удаляет из списка повторяющиеся элементы.

Решение 4.148.1

```

(defun xine-twins (w &optional ac &aux (a (car w)) (d (cdr w)))
  (cond ((null w) nil)
        ((member a ac) (xine-twins d ac))
        ((member a d) (xine-twins d (cons a ac)))
        ((cons a (xine-twins d ac)))))

> (xine-twins '(a b a a c))
(B C)

```

Задача 4.149 count-atoms.lisp

Определить количество каждого атома в многоуровневом списке.

Решение 4.149.1

```
(defun flat (w &optional acc)
  (cond ((null w) acc)
        ((atom w) (cons w acc))
        ((flat (car w) (flat (cdr w) acc)))))

(defun count-atoms (w &aux (h (make-hash-table)) ac)
  (dolist (a (flat w)) (incf (gethash a h 0)))
  (maphash #'(lambda (e n) (push `(,e . ,n) ac)) h) ac)

> (count-atoms '(a b c b (a ((k)) ())) (3 8 k))
((A . 2) (B . 2) (C . 1) (K . 2) (3 . 1) (8 . 1))
```

Решение 4.149.2

```
(defun flat (w)
  (mapcan #'(lambda (a) (if (listp a) (flat a) `(,a))) w))

(defun count-atoms (w &aux (h (make-hash-table)) ac)
  (dolist (a (flat w)) (incf (gethash a h 0)))
  (maphash #'(lambda (e n) (push `(,e . ,n) ac)) h) ac)

> (count-atoms '(a b c b (a ((k)) ())) (3 8 k))
((A . 2) (B . 2) (C . 1) (K . 2) (3 . 1) (8 . 1))
```

Решение 4.149.3

```
(defun flat (w)
  (loop for a in w
        if (listp a) append (flat a)
        else collect a))

(defun count-atoms (w &aux (h (make-hash-table)) ac)
  (dolist (a (flat w)) (incf (gethash a h 0)))
  (maphash #'(lambda (e n) (push `(,e . ,n) ac)) h) ac)

> (count-atoms '(a b c b (a ((k)) ())) (3 8 k))
((A . 2) (B . 2) (C . 1) (K . 2) (3 . 1) (8 . 1))
```

Задача 4.150 list-cadr.lisp

Дан список (a b c d). Необходимо преобразовать его в список (a (b) c (d)).

Решение 4.150.1

```
(defun list-cadr (w)
```



```

(if (cdr w)
    (cons (car w) (cons (list (cadr w)) (list-cadr (cddr w))))
    w))

> (list-cadr '(a b c d))
(A (B) C (D))
> (list-cadr '(a b c d e))
(A (B) C (D) E)

```

Задача 4.151 delete-on-list.lisp

Разработать функцию, удаляющую из исходного списка элементы, порядковые номера которых заданы во втором списке.

Решение 4.151.1

```

(defun delete-on-list (w v &optional (n 1))
  (cond ((null v) w)
        ((= (car v) n) (delete-on-list (cdr w) (cdr v) (1+ n)))
        ((cons (car w) (delete-on-list (cdr w) v (1+ n))))))

> (delete-on-list '(a s d f g h j) '(3 4 5))
(A S H J)

```

Решение 4.151.2 * для несортированного списка

```

(defun delete-on-list (w v &optional (n 1))
  (cond ((null v) w)
        ((= (car v) n) (delete-on-list (cdr w) (cdr v) (1+ n)))
        ((cons (car w) (delete-on-list (cdr w) v (1+ n))))))

(defun delete-sort (w v)
  (delete-on-list w (sort v #'<)))

> (time (delete-sort '(a b c d e f g h j k l m) '(10 9 8 7 5 4 3 2)))
Real time: 0.0 sec.
Run time: 0.0 sec.
Space: 48 Bytes
(A F L M)

```

Решение 4.151.3 * для несортированного списка

```

(defun delete-sort (w v)
  (labels ((on-list (w v n)
            (cond ((null v) w)
                  ((= (car v) n) (on-list (cdr w) (cdr v) (1+ n)))
                  ((cons (car w) (on-list (cdr w) v (1+ n))))))
    (on-list w (sort v #'<) 1)))

> (time (delete-sort '(a b c d e f g h j k l m) '(10 9 8 7 5 4 3 2)))
Real time: 0.0 sec.
Run time: 0.0 sec.

```

Space: 48 Bytes
(A F L M)

Только удаление:

```
(defun delete-sort (w v)
  (labels ((on-list (w v n)
            (cond ((null w) w)
                  ((= (car w) n) (on-list (cdr w) (cdr v) (1+ n)))
                  ((cons (car w) (on-list (cdr w) v (1+ n)))))))
    (on-list w v 1)))

> (time (delete-sort '(a b c d e f g h j k l m) '(2 3 4 5 7 8 9 10)))
Real time: 0.0 sec.
Run time: 0.0 sec.
Space: 16 Bytes
(A F L M)
```

Отдельно сортировка:

```
(defun delete-sort (v) (sort v #'<))

> (time (delete-sort '(10 9 8 7 5 4 3 2)))
Real time: 0.0 sec.
Run time: 0.0 sec.
Space: 32 Bytes
(2 3 4 5 7 8 9 10)
```

Задача 4.152 insection.lisp

Определить функцию (пересечение x y), результатом которой является список атомов (без повторения) одновременно входящих в x и y .

Решение 4.152.1

```
(defun flat (w &optional acc)
  (cond ((null w) acc)
        ((atom w) (cons w acc))
        ((flat (car w) (flat (cdr w) acc)))))

(defun inter (w v)
  (cond ((null w) nil)
        ((or (not (member (car w) v)) (member (car w) (cdr w)))
         (inter (cdr w) v))
        ((cons (car w) (inter (cdr w) v)))))

(defun insection (w v)
  (inter (flat w) (flat v)))

> (insection '((1) 3 4 (3)) '((3 4) 5))
(4 3)
```

Решение 4.152.2

```
(defun flat (w &optional acc)
  (cond ((null w) acc)
        ((atom w) (cons w acc))
        ((flat (car w) (flat (cdr w) acc)))))

(defun inter (w v &aux (a (car w)) (d (cdr w)))
  (cond ((null w) nil)
        ((or (not (member a v)) (member a d)) (inter d v))
        ((cons a (inter d v)))))

(defun insection (w v)
  (inter (flat w) (flat v)))

> (insection '((1) 3 4 (3)) '((3 4) 5))
(4 3)
```

Решение 4.152.3

```
(defun unique (w &aux (a (car w)) (d (cdr w)))
  (cond ((null w) nil)
        ((member a d) (unique d))
        ((cons a (unique d)))))

(defun flat (w)
  (mapcan #'(lambda (a) (if (listp a) (flat a) `(,a))) w))

(defun inter (w v &aux (a (car w)) (d (cdr w)))
  (cond ((null w) nil)
        ((member a v) (cons a (inter d v)))
        ((inter d v)))

(defun insection (w v)
  (inter (unique (flat w)) (unique (flat v))))

> (insection '((1) 3 4 (3)) '((3 4) 5))
(4 3)
```

Решение 4.152.4

```
(defun flat (w)
  (loop for a in w
        if (listp a) append (flat a)
        else collect a))

(defun inter (w v &aux (a (car w)) (d (cdr w)))
  (cond ((null w) nil)
        ((or (not (member a v)) (member a d)) (inter d v))
        ((cons a (inter d v)))))

(defun insection (w v)
  (inter (flat w) (flat v)))
```

```
> (insection '((1) 3 4 (3)) '((3 4) 5))
(4 3)
```

Задача 4.153 *show-names.lisp*

Имеется список студентов с информацией об итогах сессии, в котором указаны фамилия, номер группы, оценки по трем предметам. Определить фамилии студентов, имеющих задолженность хотя бы по одному предмету.

Решение 4.153.1

```
(defstruct student name group lisp smalltalk haskell)

(defun res (n)
  (when (> n 0)
    (cons (make-student
            :name (read)
            :group (read)
            :lisp (read)
            :smalltalk (read)
            :haskell (read)) (res (- n 1)))))

(defun check-2 (a)
  (or (= (student-lisp a) 2)
      (= (student-smalltalk a) 2)
      (= (student-haskell a) 2)))

(defun show-names (w)
  (cond ((null w) nil)
        ((check-2 (car w))
         (cons (student-name (car w)) (show-names (cdr w))))
        ((show-names (cdr w)))))

> (setf w (res 3))
ivanov
2
4
5
2
petrov
4
4
4
4
sidorov
3
2
2
2
(#S(STUDENT :NAME IVANOV :GROUP 2 :LISP 4 :SMALLTALK 5 :HASKELL 2)
 #S(STUDENT :NAME PETROV :GROUP 4 :LISP 4 :SMALLTALK 4 :HASKELL 4))
```

```
#S(STUDENT :NAME SIDOROV :GROUP 3 :LISP 2 :SMALLTALK 2 :HASKELL 2))
> (show-names w)
(IVANOV SIDOROV)
```

Задача 4.154 *product-tail.lisp*

Описать функцию, которая на основе списка чисел формирует список-результат следующим образом : первый элемент есть произведение элементов списка, второй – произведение элементов хвоста, третий – произведение элементов хвоста хвоста и так далее. Пример : для списка '(1 2 3 4 5 6) результатом будет : '(720 720 360 120 30 6).

Решение 4.154.1

```
(defun product-tail (w)
  (cond ((null w) nil)
        ((cons (apply '* w) (product-tail (cdr w))))))

> (product-tail '(1 2 3 4 5 6))
(720 720 360 120 30 6)
```

Решение 4.154.2

```
(defun product-tail (w)
  (when w (cons (apply '* w) (product-tail (cdr w)))))

> (product-tail '(1 2 3 4 5 6))
(720 720 360 120 30 6)
```

Решение 4.154.3

```
(defun product-tail (w) (maplist #'(lambda (a) (apply #'* a)) w))

> (product-tail '(1 2 3 4 5 6))
(720 720 360 120 30 6)
```

Решение 4.154.4

```
(defun product-tail (w) (maplist #'(lambda (a) (reduce #'* a)) w))

> (product-tail '(1 2 3 4 5 6))
(720 720 360 120 30 6)
```

Решение 4.154.5

```
(defun product-tail (w) (loop for a on w collect (apply #'* a)))

> (product-tail '(1 2 3 4 5 6))
(720 720 360 120 30 6)
```

Задача 4.155 min-trio.lisp

Определить функцию, вычисляющую три минимальных элемента списка.

Решение 4.155.1

```
(defun min-trio (w) (subseq (sort w #'<) 0 3 ))

> (min-trio '(6 5 4 3 2 1 1 2 3))
(1 1 2)
```

Решение 4.155.2

```
(defun min-trio (w &optional (a (car w)) (b a) (c a))
  (cond ((null w) (list a b c))
        ((< (car w) a) (min-trio (cdr w) (car w) a b))
        ((< (car w) b) (min-trio (cdr w) a (car w) b))
        ((< (car w) c) (min-trio (cdr w) a b (car w)))
        ((min-trio (cdr w) a b c))))

> (min-trio '(6 5 4 3 2 1 1 2 3))
(1 1 2)
```

Задача 4.156 delete-on.lisp

In some sense, Lisp is the most fundamental kind of programming language. It is very close to Lambda calculus and has almost no syntax. For this reason, Lisp has a reverence that no other language will ever get. Of course you can do anything in any language, but if you do it in Lisp it is more noble. The oldest neckbeards used Lisp. They were more like wizards than neckbeards.

Vivek Nagarajan

Определить функцию, удаляющую из списка все элементы, являющиеся подписками и числами, лежащими в некотором интервале.

Решение 4.156.1

```
(defun delete-on (w n m &aux (a (car w)) (d (cdr w)))
  (cond ((null w) nil)
        ((or (listp a) (and (numberp a) (<= n a m)))
         (delete-on d n m))
        ((cons a (delete-on d n m)))))

> (delete-on '(1 2 3 4 11 22 33 (w e r) w e r) 2 4)
(1 11 22 33 W E R)
```

Задача 4.157 list>a.lisp

Определить функцию, заменяющую в списке все элементы, являющиеся вложенными списками на атом А.

Решение 4.157.1

```
(defun list>a (w)
  (cond ((null w) nil)
        ((listp (car w)) (cons 'a (list>a (cdr w))))
        ((cons (car w) (list>a (cdr w))))))
```

```
> (list>a '((3 4) 5 6 (5 (5 8))))
(A 5 6 A)
```

Задача 4.158 neighbors.lisp

Дан одноуровневый числовой список. Необходимо заменить каждое положительное число ближайшим отрицательным справа или слева. Если нет отрицательных - заменять на nil.

Решение 4.158.1

```
(defun mi (w) (find-if #'minusp w))

(defun neighbors (w &optional ac &aux (a (car w)) (d (cdr w)))
  (when w
    (if (plusp a)
        (cond ((and (some #'minusp ac) (some #'minusp d))
              (if (< (position-if #'minusp ac)
                    (position-if #'minusp d))
                  (cons (mi ac) (neighbors d (cons a ac)))
                  (cons (mi d) (neighbors d (cons a ac)))))
              ((some #'minusp d)
               (cons (mi d) (neighbors d (cons a ac))))
              ((some #'minusp ac)
               (cons (mi ac) (neighbors d (cons a ac))))
              ((cons nil (neighbors d (cons a ac)))))
        (cons a (neighbors d (cons a ac)))))
```

```
> (neighbors '(1 -2 3 -4 -5 6 7 7 7 -2 8))
(-2 -2 -4 -4 -5 -5 -5 -2 -2 -2 -2)
```

Решение 4.158.2

```
(defun mi (w) (find-if #'minusp w))

(defun sm (w) (some #'minusp w))

(defun ps (w) (position-if #'minusp w))

(defun ne (w a v) (neighbors w (cons a v)))
```

```
(defun neighbors (w &optional ac &aux (a (car w)) (d (cdr w)))
  (when w (if (plusp a)
    (cond ((and (sm ac) (sm d))
      (if (< (ps ac) (ps d))
        (cons (mi ac) (ne d a ac))
        (cons (mi d) (ne d a ac))))
      ((sm d) (cons (mi d) (ne d a ac)))
      ((sm ac) (cons (mi ac) (ne d a ac)))
      ((cons nil (ne d a ac))))
    (cons a (ne d a ac)))))

> (neighbors '(1 -2 3 -4 -5 6 7 7 7 -2 8))
(-2 -2 -4 -4 -5 -5 -5 -2 -2 -2 -2)
```

Задача 4.159 sbst.lisp

Определить функцию *f* (*a w*), где *a* – *s*-выражение, а *w* – список, которая заменяет на *a* все атомы списка *w*.

Решение 4.159.1

```
(defun sbst (a w)
  (substitute-if a #'atom w))

> (sbst '(+ 1 2) '(a b c))
((+ 1 2) (+ 1 2) (+ 1 2))
```

Задача 4.160 za-sentence.lisp

Дан текст. Переписать каждое предложение, расположив слова в обратном алфавитном порядке.

Решение 4.160.1

```
(defun za-words (s)
  (sort s #'char-greaterp :key #'(lambda (a) (char a 0))))

(defun za-sentence (w)
  (mapcar #'za-words w))

> (za-sentence '("aaa" "ccc" "bbb") ("ddd" "fff" "eee"))
(("ccc" "bbb" "aaa") ("fff" "eee" "ddd"))
```

Задача 4.161 pos/neg.lisp

Создать функцию, которая разделит исходный список из целых чисел на два списка: список положительных чисел и список отрицательных чисел.

Решение 4.161.1


```
(defun pos/neg (w)
  (loop for a in w
        if (plusp a) collect a into pos
        if (minusp a) collect a into neg
        finally (return (values pos neg))))

> (pos/neg '(-2 -1 0 1 2))
(1 2)
(-2 -1)
```

Задача 4.162 ini-lets.lisp

Создать функцию, которая возвращает список первых букв слов, представленных списками.

Решение 4.162.1

```
(defun ini-lets (w)
  (loop for a in w when a collect (car a)))

> (ini-lets'((W o r l d) (W a r) (I)))
(W W I)
```

Задача 4.163 check.lisp

Написать функцию, что для аргумента-списка формирует список-результат по правилу: если первый и последний элементы списка-аргумента - символы, то сформировать список из первого и последнего элементов, в противном случае вернуть исходный список, из которого изъят второй элемент.

Решение 4.163.1

```
(defun check (w &aux (a (car w)) (z (last w)))
  (cons a (if (every #'symbolp (cons a z)) z (cddr w))))

> (check '(a b c g e))
(A E)
> (check '(a b c g 10))
(A C G 10)
```

Задача 4.164 num-letters.lisp

Написать функцию, определяющую количество символов в каждом слове. Пример: исходная строка (abc defg hijk), результат (3 4 4)

Решение 4.164.1

```
(defun num-letters (w)
  (mapcar #'length (mapcar #'symbol-name w)))
```

```
> (num-letters '(abc defg hijk))
(3 4 4)
```

Решение 4.164.2

```
(defun num-letters (w)
  (mapcar #'(lambda (a) (length (symbol-name a))) w))

> (num-letters '(abc defg hijk))
(3 4 4)
```

Решение 4.164.3

```
(defun num-letters (w)
  (loop for a in w collect (length (symbol-name a))))

> (num-letters '(abc defg hijk))
(3 4 4)
```

Решение 4.164.4

```
(defun num-letters (w)
  (mapcar #'length (mapcar #'string w)))

> (num-letters '(a7bc d7efg h7ijk))
(4 5 5)
```

Решение 4.164.5

```
(defun num-letters (w)
  (mapcar #'(lambda (a) (length (string a))) w))

> (num-letters '(a7bc d7efg h7ijk))
(4 5 5)
```

Решение 4.164.6

```
(defun num-letters (w)
  (loop for a in w collect (length (string a))))

> (num-letters '(a7bc d7efg h7ijk))
(4 5 5)
```

Решение 4.164.7

```
(defun num-letters (w)
  (mapcar
    #'(lambda (a)
      (length (if (symbolp a) (string a) (write-to-string a)))) w))

> (num-letters '(abc d7efg 123))
(3 5 3)
```

Решение 4.164.8

```
(defun num-letters (w)
  (loop for a in w collect
    (length (if (symbolp a) (string a) (write-to-string a)))))

> (num-letters '(abc d7efg 123))
(3 5 3)
```

Задача 4.165 *polish-notation.lisp*

Разработать функцию, преобразующую арифметическое выражение, заданное в форме списка, в польскую обратную запись.

Решение 4.165.1

```
(defun polish-notation (w)
  (cond ((null w) nil)
        ((numberp (car w)) (cons (car w) (polish-notation (cdr w))))
        ((cons (cadr w) (cons (car w) (polish-notation (cddr w)))))))

> (polish-notation '(3 * 2 - 5))
(3 2 * 5 -)
```

Решение 4.165.2

```
(defun polish-notation (w &aux (a (car w)))
  (when w
    (if (numberp a)
        (cons a (polish-notation (cdr w)))
        (cons (cadr w)
              (cons a (polish-notation (cddr w)))))))

> (polish-notation '(3 * 2 - 5))
(3 2 * 5 -)
```

Задача 4.166 *odd-even.lisp*

Разработать функцию, увеличивающую в исходном списке нечетные элементы (по значению) на 1 и уменьшающую четные на 2.

Решение 4.166.1

```
(defun odd-even (w &aux (a (car w)))
  (when w (cons (if (oddp a) (1+ a) (- a 2))
                (odd-even (cdr w)))))

> (odd-even '(1 2 3 4 5 6 7 8 9))
(2 0 4 2 6 4 8 6 10)
```

Задача 4.167 reverse-annex.lisp

Разработать функцию, которая преобразует список вида (a b c d) в список вида (((d)c)b)a).

Решение 4.167.1

```
(defun reverse-annex (w &aux (v (reverse w)))
  (reduce #'list (cdr v) :initial-value (list (car v))))

> (reverse-annex '(a b c d e))
((((E) D) C) B) A)
```

Решение 4.167.2

```
(defun reverse-annex (w &aux (v (reverse w)))
  (reduce #'list (cdr v) :initial-value (list (car v))))

> (reverse-annex '(a b c d e))
((((E) D) C) B) A)
```

Задача 4.168 unique-symb.lisp

Из заданного предложения выбрать только те символы, которые встречаются в нем только один раз, в том порядке, как они расположены в тексте.

Решение 4.168.1

```
(defun unique-symb (w)
  (loop for a in w when (= (count a w) 1) collect a))

> (unique-symb '(aa f aa bb ff gg ff))
(F BB GG)
```

Задача 4.169 sum-shift.lisp

Разработать функцию, формирующую на основе исходного списка длины N список, содержащий суммы элементов с номерами 1 и $N/2+1$, 2 и $N/2+2$ и т. д. N — аргумент функции. Например: Вход: (1 2 3 4 5 6 7 8 9 10). Выход: (7 9 11 13 15).

Решение 4.169.1

```
(defun sum-shift (w &optional (v (nthcdr (/ (length w) 2) w)))
  (when v (cons (+ (car w) (car v)) (sum-shift (cdr w) (cdr v)))))

> (time (sum-shift (loop for i from 1 to 100 collect (random 100))))
Real time: 0.0 sec.
Run time: 0.0 sec.
Space: 5436 Bytes
```

```
(106 167 154 34 61 50 71 147 190 115 118 120 67 36 188 89 184 121 93
74 106 95 135 64 94 147 158 45 116 73 87 120 101 73 154 67 114 35 32
105 160 169 155 78 101 92 107 120 114 92)
```

Решение 4.169.2

```
(defun sum-shift (w)
  (loop for a in w
        for b in (subseq w (/ (length w) 2))
        collect (+ a b)))

> (time (sum-shift (loop for i from 1 to 100 collect (random 100))))
Real time: 0.0 sec.
Run time: 0.0 sec.
Space: 5836 Bytes
(115 93 24 87 54 83 10 49 76 79 41 30 75 172 102 148 68 85 80 166 66
84 115 180 128 69 28 87 164 107 151 56 114 124 150 146 130 153 60 87
177 111 92 101 93 80 96 11 78 72)
```

Решение 4.169.3

```
(defun sum-shift (w)
  (loop for a in w
        for b in (nthcdr (/ (length w) 2) w)
        collect (+ a b)))

> (time (sum-shift (loop for i from 1 to 100 collect (random 100))))
Real time: 0.0 sec.
Run time: 0.0 sec.
Space: 5436 Bytes
(68 88 116 93 173 153 17 28 29 114 39 141 55 135 41 124 147 134 57 106
134 106 99 25 112 109 112 111 87 69 52 165 151 160 74 124 85 151 64
115 0 71 85 89 117 49 99 109 108 151)
```

Задача 4.170 del-list.lisp

Написать функцию, которая в списке удаляет все не атомарные элементы.

Решение 4.170.1

```
(defun del-list (w) (delete-if #'listp w))

> (del-list '(1 (2) 3 (4) 5))
(1 3 5)
```

Задача 4.171 add.lisp

Дан список. Увеличить каждый элемент на 1.

Решение 4.171.1

```
(defun add (w)
  (when w
    (cons (if (atom (car w)) (1+ (car w)) (add (car w)))
          (add (cdr w))))))

> (add '((1 2) (3 4) 6))
((2 3) (4 5) 7)
```

Решение 4.171.2

```
(defun add (w)
  (mapcar #'(lambda (a) (if (atom a) (1+ a) (add a))) w))

> (add '((1 2) (3 4) 6))
((2 3) (4 5) 7)
```

Решение 4.171.3

```
(defun add (w)
  (loop for a in w
        collect (if (atom a) (1+ a) (add a))))

> (add '((1 2) (3 4) 6))
((2 3) (4 5) 7)
```

Задача 4.172 *increase.lisp*

Увеличить каждый элемент списка в произвольное число раз. Анонимная функция должна быть передана параметром в вызываемую функцию.

Решение 4.172.1

```
(defun increase (w f &aux (a (car w)))
  (cond ((null w) nil)
        ((atom a) (cons (funcall f a) (increase (cdr w) f)))
        ((cons (increase a f) (increase (cdr w) f)))))

> (increase '((1 2) (3 4)) (5)) (lambda (x) (* x 3))
(((3 6) (9 12)) (15))
```

Задача 4.173 *let-number.lisp*

Дан список. Заменить буквы английского алфавита на соответствующий им номер в алфавите.

Решение 4.173.1

```
(defun let-number
  (w &aux
    (p (position
        (car w)

```

```

      '(a b c d e f g h i j k l m n o p q r s t u v w x y z)))
    (when w (cons (if p (1+ p) (car w)) (let-number (cdr w)))))

> (let-number '(a s d ! r z))
(1 19 4 ! 18 26)

```

Задача 4.174 list-unlist.lisp

Написать программу - элементы списка должны стать подписками, а подписки - элементами главного списка: (a b c (d e) e f (g k)) -> ((a b c) d e (e f) g k).

Решение 4.174.1

```

(defun list-unlist (w &optional ac &aux (a (car w)) (d (cdr w)))
  (cond ((null w) (when ac (cons (reverse ac) nil)))
        ((atom a) (list-unlist d (cons a ac)))
        (ac (nconc (cons (reverse ac) a) (list-unlist d nil)))
        ((nconc a (list-unlist d nil)))))

> (list-unlist '((a) a b c (d e) f g (h i j) k l))
(A (A B C) D E (F G) H I J (K L))
> (list-unlist '(a b c (d e) f g (h i j) k l))
((A B C) D E (F G) H I J (K L))
> (list-unlist '(a b c (d e) f g (h i j)))
((A B C) D E (F G) H I J)

```

Решение 4.174.2 (автор - VH, www.cyberforum.ru)

```

(defun list-unlist (w)
  (when w ((lambda (e r) (if (atom e)
                              (if (atom (car r))
                                  (cons (list e) r)
                                  (cons (cons e (car r)) (cdr r)))
                              (nconc e r)))
          (car w) (list-unlist (cdr w)))))

> (list-unlist '((a) b c (d e) f g (h i)))
(A (B C) D E (F G) H I)

```

Задача 4.175 last-first.lisp

Определите функцию, которая меняет местами первый и последний элементы списка, оставляя остальные на своих местах.

Решение 4.175.1

```

(defun last-first (w)
  (nconc (last w) (cdr (butlast w)) `((, (car w)))))

```

```
> (last-first '(nil 1 2 3 4 5))
(5 1 2 3 4 NIL)
> (last-first '(1 2 3 4 5))
(5 2 3 4 1)
```

Задача 4.176 min-max.lisp

Emacs is a Emacs Lisp interpreter and it just happened to be designed as an editor. It can be anything you want it to be (file manager, music player etc). Other editors are purely editors which allow extension of certain features using certain supported programming languages.

Gopi Vangaveti

Найти элементы одноуровневого числового списка, имеющие минимальное значение и максимальное значение, вывести на экран их порядковые номера.

Решение 4.176.1

```
(defun min-max (w &aux
  (n (apply #'min w))
  (x (apply #'max w))
  (v (loop for a in w
    for b from 0 to (1- (length w))
    collect (list a b))))
  (loop for c in v
    if (eql (car c) n) collect (cadr c) into nn
    if (eql (car c) x) collect (cadr c) into xx
    finally (return (values nn xx))))

> (min-max '(1 2 3 4 1 2 3 4 1 2 3 4))
(0 4 8)
(3 7 11)
```

Решение 4.176.2* (для XLISP)

```
(defun min-max (w)
  (list-min-max (list-an w) (apply #'min w) (apply #'max w)))

(defun list-min-max (w n x &optional nn xx)
  (cond ((null w) (list (reverse nn) (reverse xx)))
        ((eql (caar w) n)
         (list-min-max (cdr w) n x (cons (cadr w) nn) xx))
        ((eql (caar w) x)
         (list-min-max (cdr w) n x nn (cons (cadr w) xx)))
        (t (list-min-max (cdr w) n x nn xx))))

(defun list-an (w &optional (n 0))
```



```
(when w (cons (list (car w) n) (list-an (cdr w) (1+ n)))))

> (min-max '(1 2 3 4 1 2 3 4 1 2 3 4))
((0 4 8) (3 7 11))
```

Задача 4.177 letter-likeness.lisp

Для заданного текста определите пару слов, буквенный состав которого наиболее схож.

Решение 4.177.1

```
(defun letter-likeness (w)
  (mapcar #'
    (lambda (a) (read-from-string (apply #'concatenate 'string a)))
    (cdar (sort (len-w1-w2 (cartesian-mill (letters w)))
      #'> :key #'car))))

(defun len-w1-w2 (w)
  (mapcar #'(lambda (a)
    (cons (length (intersection (str-sym (car a))
      (str-sym (cadr a))))
      a)) w))

(defun str-sym (w)
  (mapcar #'(lambda (a) (read-from-string a)) w))

(defun cartesian-mill (w)
  (when (cdr w) (nconc (cartesian (list (car w)) (cdr w))
    (cartesian-mill (cdr w)))))

(defun cartesian (w v)
  (mapcan (lambda (e) (mapcar (lambda (a) `(,a ,e)) w)) v))

(defun letters (w)
  (mapcar #'(lambda (a) (map 'list #'string (string a))) w))

> (letter-likeness '(abcde defgi bcdef))
(ABCDE BCDEF)
```

Задача 4.178 matrix-rotate-max-min.lisp

Найти в каждой строке наибольший и наименьший элементы матрицы и поменять их местами.

Решение 4.178.1

```
(defun rotate-max-min (w &aux (x (apply #'max w)) (n (apply #'min w)))
  (rotatef (nth (position x w) w) (nth (position n w) w)) w)

(defun matrix-rotate-max-min (w) (mapcar #'rotate-max-min w))
```

```
> (matrix-rotate-max-min '((1 2 3 4 5) (5 4 3 2 1) (1 2 3 4 5)))
((5 2 3 4 1) (1 4 3 2 5) (5 2 3 4 1))
```

Решение 4.178.2 (для XLISP)

```
(defun rotate-max-min
  (w &aux
    (x (apply #'max w))
    (n (apply #'min w))
    (px (pos x w))
    (pn (pos n w)))
  (let ((tmp (nth px w)))
    (setf (nth px w) (nth pn w) (nth pn w) tmp))
  w)

(defun pos (a w &optional (p 0))
  (cond ((null w) nil)
        ((eql (car w) a) p)
        (t (pos a (cdr w) (+ p 1)))))

(defun matrix-rotate-max-min (w) (mapcar #'rotate-max-min w))

> (matrix-rotate-max-min '((1 2 3 4 5) (5 4 3 2 1) (1 2 3 4 5)))
((5 2 3 4 1) (1 4 3 2 5) (5 2 3 4 1))
```

Задача 4.179 del-second.lisp

Определить рекурсивный и итерационный варианты функции, удаляющей каждый второй элемент списка.

Решение 4.179.1

```
(defun del-second (w)
  (when w (cons (car w) (del-second (cddr w)))))

> (del-second '(1 2 3 4 5 6 7))
(1 3 5 7)
```

Решение 4.179.2

```
(defun del-second (w)
  (loop for a in w by #'cddr collect a))

> (del-second '(1 2 3 4 5 6 7))
(1 3 5 7)
```

Задача 4.180 sublist-cdddr.lisp

Дан многоуровневый список. Вывести все списки второго уровня, где количество элементов больше трех.

Решение 4.180.1

```
(defun sublist-cdddr (w)
  (remove-if-not #'(lambda (a) (and (listp a) (cdddr a))) w))

> (sublist-cdddr '(1 (2 3) (4 5 6 7) 8 (9 0)))
((4 5 6 7))
```

Задача 4.181 pos-max-min.lisp

Найти элементы одномерного числового массива, имеющие максимальное значение и минимальное значение, вывести на экран их порядковые номера.

Решение 4.181.1

```
(defun pos-max-min (w)
  (max-min (cdr w) (list (car w) 0) (list (car w) 0)))

(defun max-min (w x n &optional (u 1))
  (cond ((null w) (list (cadr x) (cadr n)))
        ((> (car w) (car x))
         (max-min (cdr w) (list (car w) u) n (1+ u)))
        ((< (car w) (car n))
         (max-min (cdr w) x (list (car w) u) (1+ u)))
        (t (max-min (cdr w) x n (1+ u)))))

> (time (pos-max-min (loop for i from 1 to 4096 collect (random
1000))))
Real time: 0.0312001 sec.
Run time: 0.0312002 sec.
Space: 37260 Bytes
(1622 605)
```

Решение 4.181.2

```
(defun pos-max-min (v)
  (mapcar #'(lambda (a) (position a v))
    `((, (apply #'max v) , (apply #'min v)))))

> (time (pos-max-min (loop for i from 1 to 4096 collect (random
1000))))
Real time: 0.0300004 sec.
Run time: 0.0312002 sec.
Space: 37036 Bytes
(468 937)
```

Решение 4.181.3

```
(defun pos-max-min (v)
  (values (pos #'max v) (pos #'min v)))
```

```
(defun pos (p v) (position (reduce p v) v))

> (time (pos-max-min (loop for i from 1 to 4096 collect (random
1000))))
Real time: 0.0337001 sec.
Run time: 0.0312002 sec.
Space: 37004 Bytes
GC: 1, GC time: 0.0 sec.
624
879
```

Задача 4.182 name-years.lisp

Напишите функцию, которая спрашивает у пользователя фамилию студента из группы и выдает год его рождения.

Решение 4.182.1

```
(defstruct student name year)

(defun group (n)
  (when (> n 0) (cons (make-student :name (read) :year (read))
    (group (1- n)))))

(defun years (w input)
  (loop for a in w
    when (equalp (student-name a) input)
    collect (student-year a)))

(defun name-years (w)
  (years w (read)))

> (setf 3a (group 3))
ivanov
1994
petrov
1993
ivanov
1995
(#S(STUDENT :NAME IVANOV :YEAR 1994) #S(STUDENT :NAME PETROV :YEAR
1993) #S(STUDENT :NAME IVANOV :YEAR 1995))
> (name-years 3a)
petrov
(1993)
> (name-years 3a)
ivanov
(1994 1995)
```

Решение 4.182.2

```
(defparameter students '((Ivanov 1994) (Petrov 1993) (Ivanov 1995)))
```

```
(defun years (w input)
  (loop for a in w
        when (equalp (car a) input)
        collect (cadr a)))
```

```
(defun name-years (w)
  (years w (read)))
```

```
> (name-years students)
petrov
(1993)
> (name-years students)
ivanov
(1994 1995)
```

Решение 4.182.3

```
(defun years (w input)
  (loop for a in w
        when (equalp (car a) input)
        collect (cadr a)))
```

```
(defun name-years (w)
  (years w (read)))
```

```
> (name-years '((Ivanov 1994) (Petrov 1993) (Ivanov 1995)))
petrov
(1993)
> (name-years '((Ivanov 1994) (Petrov 1993) (Ivanov 1995)))
ivanov
(1994 1995)
```

Решение 4.182.4

```
(defun years (w input)
  (if (numberp input)
      (loop for a in w when (eq (cadr a) input)
            collect (car a))
      (loop for a in w when (eq (car a) input)
            collect (cadr a))))
```

```
(defun name-years (w)
  (years w (read)))
```

```
> (name-years '((Ivanov 1994) (Petrov 1993) (Ivanov 1995)))
ivanov
(1994 1995)
> (name-years '((Ivanov 1994) (Petrov 1993) (Ivanov 1995)))
1993
(PETROV)
```

Задача 4.183 flat-zero.lisp

Определить рекурсивную функцию, которая выравнивает список и применяет предикат `zerop` к каждому элементу списка.

Решение 4.183.1

```
(defun flat-zero (w &optional acc)
  (cond ((null w) acc)
        ((atom w) (cons (zerop w) acc))
        ((flat-zero (car w) (flat-zero (cdr w) acc)))))

> (flat-zero '(((1)) 0 (1)))
(NIL T NIL)
```

Решение 4.183.2

```
(defun flat-zero (w)
  (mapcan
   #'(lambda (a) (if (listp a)
                      (flat-zero a)
                      `(, (zerop a))))
   w))

> (flat-zero '(((1)) 0 (1)))
(NIL T NIL)
```

Решение 4.183.3

```
(defun flat-zero (w)
  (loop for a in w
        if (listp a) append (flat-zero a)
        else collect (zerop a)))

> (flat-zero '(((1)) 0 (1)))
(NIL T NIL)
```

Задача 4.184 n-atoms-n-lists.lisp

Создать рекурсивную функцию, которая считает сколько в списке атомов и сколько подписков.

Решение 4.184.1

```
(defun n-atoms-n-lists (w)
  (list (n-atoms w) (n-lists w)))

(defun n-atoms (w)
  (cond ((null w) 0)
        ((atom w) 1)
        ((+ (n-atoms (car w)) (n-atoms (cdr w))))))
```

```
(defun n-lists (w)
  (cond ((null w) 0)
        ((atom (car w)) (n-lists (cdr w)))
        ((+ 1 (n-lists (car w)) (n-lists (cdr w))))))
```

```
> (n-atoms-n-lists '((1 2) (3 (4)) 5 ((6))))
(6 5)
```

Решение 4.184.2

```
(defun n-atoms-n-lists (w)
  (values (n-atoms w) (n-lists w)))

(defun n-atoms (w)
  (cond ((null w) 0)
        ((atom w) 1)
        ((+ (n-atoms (car w)) (n-atoms (cdr w))))))

(defun n-lists (w)
  (cond ((null w) 0)
        ((atom (car w)) (n-lists (cdr w)))
        ((+ 1 (n-lists (car w)) (n-lists (cdr w))))))

> (n-atoms-n-lists '((1 2) (3 (4)) 5 ((6))))
6
5
```

Решение 4.184.3

```
(defun n-atoms-n-lists (w)
  (values (n-atoms w) (n-lists w)))

(defun n-atoms (w)
  (if w
      (if (atom w)
          1
          (+ (n-atoms (car w)) (n-atoms (cdr w))))
      0))

(defun n-lists (w &aux (a (car w)))
  (if w
      (+ (if (atom a)
              0
              (1+ (n-lists a)))
         (n-lists (cdr w)))
      0))

> (n-atoms-n-lists '((1 2) (3 (4)) 5 ((6))))
6
5
```

Задача 4.185 multiply.lisp

Определить функцию (multiply (list-1) (list-2) ... (list-n)), которая перемножит почленно списки и вернет новый список.

Решение 4.185.1

```
(defun multiply (&rest w) (apply #'mapcar #'* w))

> (multiply '(1 2) '(10 20) '(100 200))
(1000 8000)
```

Решение 4.185.2

```
(defmacro multiply (&rest w) `(mapcar #'* ,@w))

> (multiply '(1 2) '(10 20) '(100 200))
(1000 8000)
```

Задача 4.186 pair-change.lisp

Определить функцию, попарно меняющую элементы списка.

Решение 4.186.1

```
(defun pair-change (w)
  (if (cdr w)
      (cons (cadr w) (cons (car w) (pair-change (cddr w))))
      w))

> (pair-change '(1 2 3 4 5 6 7))
(2 1 4 3 6 5 7)
> (pair-change '(1 2 3 4 5 6))
(2 1 4 3 6 5)
> (pair-change '(1 2))
(2 1)
> (pair-change '(1))
(1)
> (pair-change '())
NIL
```

Решение 4.186.2

```
(defun pair-change (w &aux (v (loop for a in w by #'cddr
                                     for b in (cdr w) by #'cddr
                                     nconc ` (,b ,a))))
  (if (oddp (length w)) (nconc v (last w)) v))

> (pair-change '(1 2 3 4 5 6 7))
(2 1 4 3 6 5 7)
> (pair-change '(1 2 3 4 5 6))
(2 1 4 3 6 5)
```



```
> (pair-change '(1 2))
(2 1)
> (pair-change '(1))
(1)
> (pair-change '())
NIL
```

Решение 4.186.3 (автор - transformator.t, www.cyberforum.ru)

```
(defun pair-change (w)
  (loop for (a b) on w by #'cddr
        if b nconc (list b a)
        else nconc (list a)))

> (pair-change '(1 2 3 4 5 6 7))
(2 1 4 3 6 5 7)
> (pair-change '(1 2 3 4 5 6))
(2 1 4 3 6 5)
> (pair-change '(1 2))
(2 1)
> (pair-change '(1))
(1)
> (pair-change '())
NIL
```

Решение 4.186.4

```
(defun pair-change (w)
  (loop for (a b) on w by #'cddr
        if b nconc ` (,b ,a)
        else nconc ` (,a)))

> (pair-change '(1 2 3 4 5 6 7))
(2 1 4 3 6 5 7)
> (pair-change '(1 2 3 4 5 6))
(2 1 4 3 6 5)
> (pair-change '(1 2))
(2 1)
> (pair-change '(1))
(1)
> (pair-change '())
NIL
```

Задача 4.187 *del-over-average.lisp*

Определить функцию, удаляющую из списка элементы, которые больше среднего арифметического.

Решение 4.187.1

```
(defun del-over-average (w)
  (over-aver w (/ (sum w) (len w))))
```

```

(defun len (w)
  (if w (1+ (len (cdr w))) 0))

(defun sum (w)
  (if w (+ (car w) (sum (cdr w))) 0))

(defun over-aver (w n)
  (cond ((null w) nil)
        ((> (car w) n) (over-aver (cdr w) n))
        ((cons (car w) (over-aver (cdr w) n)))))

> (del-over-average '(1 2 3 4 5 6 7))
(1 2 3 4)

```

Задача 4.188 eventh-max-minusp.lisp

Определить функцию, заменяющую значения элементов с четными номерами на значение максимального отрицательного элемента.

Решение 4.188.1

```

(defun eventh-max-minusp (w)
  (eventh w (max-minusp w)))

(defun eventh (w n)
  (when (cdr w) (cons (car w) (cons n (eventh (cddr w) n)))))

(defun max-minusp (w)
  (reduce #'max (remove-if-not #'minusp w)))

> (eventh-max-minusp '(-2 -1 0 1 2 3))
(-2 -1 0 -1 2 -1)

```

Задача 4.189 _reverse.lisp

Определить функцию, обращающую список.

Решение 4.189.1

```

(defun _reverse (w &optional ac)
  (if w (_reverse (cdr w) (cons (car w) ac)) ac))

> (_reverse '(1 2 3 4 5))
(5 4 3 2 1)

```

Решение 4.189.2

```

(defun _reverse (w)
  (cond ((null w) nil)
        ((append (_reverse (cdr w)) (list (car w))))))

```

```
> (_reverse '(1 2 3 4 5))
(5 4 3 2 1)
```

Решение 4.189.3

```
(defun _reverse (w)
  (when w (append (_reverse (cdr w)) (list (car w)))))
```

```
> (_reverse '(1 2 3 4 5))
(5 4 3 2 1)
```

Решение 4.189.4

```
(defun _reverse (w)
  (cond ((null w) nil)
        ((_append (_reverse (cdr w)) (list (car w)))))
```

```
(defun _append (w v)
  (cond ((null w) v)
        ((cons (car w) (_append (cdr w) v)))))
```

```
> (_reverse '(1 2 3 4 5))
(5 4 3 2 1)
```

Решение 4.189.5

```
(defun _reverse (w)
  (when w (_append (_reverse (cdr w)) (list (car w)))))
```

```
(defun _append (w v)
  (if w (cons (car w) (_append (cdr w) v)) v))
```

```
> (_reverse '(1 2 3 4 5))
(5 4 3 2 1)
```

Решение 4.189.6

```
(defun _reverse (w)
  (when w (nconc (_reverse (cdr w)) (list (car w)))))
```

```
> (_reverse '(1 2 3 4 5))
(5 4 3 2 1)
```

Решение 4.189.7

```
(defun _reverse (list &aux acc) (dolist (e list) (push e acc)) acc)
```

```
> (_reverse '(1 2 3 4 5))
(5 4 3 2 1)
```

Задача 4.190 drop-eventh.lisp

Определить функцию, удаляющую из списка все элементы с четными номерами.

Решение 4.190.1

```
(defun drop-eventh (w)
  (cond ((null w) nil)
        (t (cons (car w) (drop-eventh (cddr w))))))
```

```
> (drop-eventh '(1 2 3 4 5 6))
(1 3 5)
```

Решение 4.190.2

```
(defun drop-eventh (w)
  (cond ((null w) nil)
        (t (cons (car w) (drop-eventh (cddr w))))))
```

```
> (drop-eventh '(1 2 3 4 5 6))
(1 3 5)
```

Решение 4.190.3

```
(defun drop-eventh (w &optional f)
  (cond ((null w) nil)
        (f (drop-eventh (cdr w) nil))
        ((cons (car w) (drop-eventh (cdr w) t)))))
```

```
> (drop-eventh '(1 2 3 4 5 6))
(1 3 5)
```

Решение 4.190.4

```
(defun drop-eventh (w &optional f)
  (when w (if f
              (drop-eventh (cdr w) nil)
              (cons (car w) (drop-eventh (cdr w) t)))))
```

```
> (drop-eventh '(1 2 3 4 5 6))
(1 3 5)
```

Решение 4.190.5

```
(defun drop-eventh (w)
  (loop for a in w by #'cddr collect a))
```

```
> (drop-eventh '(1 2 3 4 5 6))
(1 3 5)
```

Задача 4.191 delm-thirds.lisp

Определить функцию, удаляющую каждое третье вхождение каждого элемента в список.

Решение 4.191.1

```
(defun numbers (w &optional ac &aux (a (car w)))
  (when w (cons (1+ (count a ac)) (numbers (cdr w) (cons a ac)))))

(defun delm-third (w v)
  (cond ((null w) nil)
        ((zerop (rem (car v) 3)) (delm-third (cdr w) (cdr v)))
        ((cons (car w) (delm-third (cdr w) (cdr v))))))

(defun delm-thirds (w)
  (delm-third w (numbers w)))

> (delm-thirds '(a 1 s 2 d 3 a 1 s 2 2 d a 1 s d))
(A 1 S 2 D 3 A 1 S 2 D)
```

Решение 4.191.2

```
(defun numbers (w &optional ac &aux (a (car w)))
  (when w (cons (1+ (count a ac)) (numbers (cdr w) (cons a ac)))))

(defun delm-thirds (w)
  (loop for a in w
        for b in (numbers w)
        unless (zerop (rem b 3))
        collect a))

> (delm-thirds '(a 1 s 2 d 3 a 1 s 2 2 d a 1 s d))
(A 1 S 2 D 3 A 1 S 2 D)
```

Задача 4.192 drop-sym-with-digits.lisp

Определить функцию, удаляющую из списка атомы, содержащие цифры.

Решение 4.192.1

```
(defun drop-sym-with-digits (w)
  (labels
    ((drop (v)
      (cond ((null v) nil)
            ((some #'digit-char-p
                  (coerce (string (car v)) 'list))
              (drop (cdr v)))
            ((cons (car v) (drop (cdr v)))))))
    (drop (remove-if #'numberp w))))

> (drop-sym-with-digits '(asd gh fl2s 6 7 a s h7))
```

```
(ASD GH A S)
```

Решение 4.192.2

```
(defun drop (w)
  (cond ((null w) nil)
        ((some #'digit-char-p
                 (coerce (string (car w)) 'list))
         (drop (cdr w)))
        ((cons (car w) (drop (cdr w))))))

(defun drop-sym-with-digits (w)
  (drop (remove-if #'numberp w)))

> (drop-sym-with-digits '(asd gh f12s 6 7 a s h7))
(ASD GH A S)
```

Решение 4.192.3

```
(defun drop-sym-with-digits (w)
  (remove-if
   #'(lambda (a)
        (some #'digit-char-p
                (coerce (string a) 'list)))
   (remove-if #'numberp w)))

> (drop-sym-with-digits '(asd gh f12s 6 7 a s h7))
(ASD GH A S)
```

Решение 4.192.4

```
(defun drop-sym-with-digits (w)
  (remove-if
   #'(lambda (a)
        (or (numberp a)
            (some #'digit-char-p
                    (coerce (string a) 'list))))
   w))

> (drop-sym-with-digits '(asd gh f12s 6 7 a s h7))
(ASD GH A S)
```

Задача 4.193 *drop-core.lisp*

Определить функцию удаления элемента списка, стоящего в середине. Если количество элементов четное то удалить два элемента, что в середине.

Решение 4.193.1

```
(defun drop-core (w &aux
                  (n (length w))
                  (h (floor n 2)))
```

```

(m (zerop (rem n 2)))
(when w (if m
  (nconc (subseq w 0 (1- h)) (subseq w (1+ h)))
  (nconc (subseq w 0 h) (subseq w (1+ h))))))

> (drop-core '(1 2 3 4))
(1 4)
> (drop-core '(1 2 3))
(1 3)
> (drop-core '(1 2))
NIL

```

Решение 4.193.2

```

(defun drop-core (w &aux
  (n (length w))
  (h (floor n 2))
  (m (zerop (rem n 2))))
  (when w (nconc (if m (subseq w 0 (1- h)) (subseq w 0 h))
    (subseq w (1+ h)))))

> (drop-core '(1 2 3 4))
(1 4)
> (drop-core '(1 2 3))
(1 3)
> (drop-core '(1 2))
NIL

```

Решение 4.193.3

```

(defun drop-core (w &aux (n (length w)) (h (floor n 2)))
  (when w (nconc (if (zerop (rem n 2))
    (subseq w 0 (1- h))
    (subseq w 0 h))
    (subseq w (1+ h)))))

> (drop-core '(1 2 3 4))
(1 4)
> (drop-core '(1 2 3))
(1 3)
> (drop-core '(1 2))
NIL

```

Задача 4.194 *sum-underscore-code.lisp*

Составить рекурсивную функцию для обработки списка из любого количества атомов, которые составлены из любых знаков. Каждый атом составить из суммы цифр, знака подчеркивания и ASCII кодов букв. Пример: на вход подается список (2d 4r 1s). На выходе должно быть (7_100 7_114 7_115).

Решение 4.194.1

```
(defun sum-digits (w &aux (a (car w)) (d (cdr w)))
  (if w (+ (apply #' + (mapcar #'digit-char-p
                                (remove-if-not
                                 #'digit-char-p
                                 (coerce (string a) 'list)))))
    (sum-digits d))
  0))
```

```
(defun charr (a)
  (car (remove-if-not #'alpha-char-p
                     (coerce (string-downcase (string a))
                             'list))))
```

```
(defun sum-code (w s)
  (mapcar
   #'(lambda (a)
        (concatenate 'string
                     (write-to-string s)
                     " "
                     (write-to-string (char-code (charr a)))))
    w))
```

```
(defun sum-underscore-code (w)
  (mapcar #'intern (sum-code w (sum-digits w))))
```

```
> (sum-underscore-code '(2d 4r 1s))
(|7_100| |7_114| |7_115|)
```

Решение 4.194.2

```
(defun mapcr (p w)
  (when w (cons (funcall p (car w)) (mapcr p (cdr w)))))
```

```
(defun rem-if-not (p w)
  (cond ((null w) nil)
        ((funcall p (car w)) (cons (car w) (rem-if-not p (cdr w))))
        ((rem-if-not p (cdr w)))))
```

```
(defun app (w)
  (if w (+ (car w) (app (cdr w))) 0))
```

```
(defun sum-digits (w &aux (a (car w)) (d (cdr w)))
  (if w (+ (app (mapcr #'digit-char-p
                      (rem-if-not
                       #'digit-char-p
                       (coerce (string a)
                               'list))))
    (sum-digits d))
  0))
```

```
(defun charr (a)
  (car (rem-if-not #'alpha-char-p
```



```

(coerce (string-downcase (string a))
  'list))))

(defun sum-code (w s)
  (mapcar
    #'(lambda (a)
      (concatenate 'string
        (write-to-string s)
        "-"
        (write-to-string (char-code (charr a)))))
      w))

(defun sum-underscore-code (w)
  (mapcar #'intern
    (sum-code w (sum-digits w))))

> (sum-underscore-code '(2d 4r 1s))
(|7_100| |7_114| |7_115|)

```

Задача 4.195 _maplist.lisp

A very small percentage of Emacs is implemented in C to build an Emacs Lisp layer which translates Emacs Lisp commands. The majority of Emacs is then written using this Emacs Lisp layer. What this results in is that your editor can be updated dynamically. You do not need to exit the program and recompile for you to see the changes. You can change most of the editor behavior while using Emacs... Hence the reason lots of people say that Emacs is an OS. You never have to exit Emacs and do anything you need from within it.

Gopi Vangaveti

Определите функционал, аналогичный встроенному предикату `maplist` для одноуровневого списка. Используйте применяющие функционалы. Проверьте работу функционала для функции `reverse`.

Решение 4.195.1

```

(defun _maplist (p w)
  (when w (cons (funcall p w)
    (_maplist p (cdr w)))))

> (_maplist #'reverse '(1 2 3 4 5 6))
((6 5 4 3 2 1) (6 5 4 3 2) (6 5 4 3) (6 5 4) (6 5) (6))

```

Задача 4.196 *sum-list-atom.lisp*

Дан список, который состоит из подсписков и атомов. Необходимо построить список атомов и сумм каждого подсписка.

Решение 4.196.1

```
(defun sum-list-atom (w)
  (when w (cons (if (atom (car w)) (car w) (apply #'+ (car w)))
                (sum-list-atom (cdr w)))))
```

```
> (sum-list-atom '(1 (1 1)))
(1 2)
```

Решение 4.196.2

```
(defun sum-list-atom (w)
  (loop for a in w collect (if (atom a) a (apply #'+ a))))
```

```
> (sum-list-atom '(1 (1 1)))
(1 2)
```

Задача 4.197 *drop-elm.lisp*

Удалить заданный элемент из списка на всех уровнях вложенности.

Решение 4.197.1

```
(defun drop-elm (a w)
  (cond ((null w) nil)
        ((equal a (car w)) (drop-elm a (cdr w)))
        ((listp (car w))
         (cons (drop-elm a (car w)) (drop-elm a (cdr w))))
        ((cons (car w) (drop-elm a (cdr w))))))
```

```
> (drop-elm '(2) '(1 ((1 2 (2)) 1) (2) 1 2))
(1 ((1 2) 1) 1 2)
> (drop-elm 2 '(1 ((1 2 (2)) 1) (2) 1 2))
(1 ((1 NIL) 1) NIL 1)
```

Решение 4.197.2

```
(defun drop-elm (z w &aux (a (car w)) (d (cdr w)))
  (cond ((null w) nil)
        ((equal z a) (drop-elm z d))
        ((listp a)
         (cons (drop-elm z a) (drop-elm z d)))
        ((cons a (drop-elm z d)))))
```

```
> (drop-elm '(2) '(1 ((1 2 (2)) 1) (2) 1 2))
(1 ((1 2) 1) 1 2)
> (drop-elm 2 '(1 ((1 2 (2)) 1) (2) 1 2))
```

```
(1 ((1 NIL) 1) NIL 1)
```

Решение 4.197.3

```
(defun drop-elm (z w)
  (loop for a in w unless (equal a z)
        collect (if (listp a) (drop-elm z a) a)))

> (drop-elm '(2) '(1 ((1 2 (2)) 1) (2) 1 2))
(1 ((1 2) 1) 1 2)
> (drop-elm 2 '(1 ((1 2 (2)) 1) (2) 1 2))
(1 ((1 NIL) 1) NIL 1)
```

Решение 4.197.4

```
(defun drop-elm (z w)
  (loop for a in (remove z w :test #'equal)
        collect (if (atom a) a (drop-elm z a))))

> (drop-elm '(2) '(1 ((1 2 (2)) 1) (2) 1 2))
(1 ((1 2) 1) 1 2)
> (drop-elm 2 '(1 ((1 2 (2)) 1) (2) 1 2))
(1 ((1 NIL) 1) NIL 1)
```

Задача 4.198 flat-unique.lisp

Определить функцию (множество w), аргументом которой является список с подписками, а результатом – список атомов без повторения.

Решение 4.198.1

```
(defun flat-unique (w &optional ac)
  (cond ((null w) ac)
        ((atom w) (if (member w ac) ac (cons w ac)))
        ((flat-unique (car w) (flat-unique (cdr w) ac)))))

> (flat-unique '((a b) ((a)) a c))
(B A C)
```

Решение 4.198.2

```
(defun flat-unique (w)
  (remove-duplicates
   (labels ((flat (w)
              (mapcan
               #'(lambda (a) (if (listp a)
                                (flat a)
                                `((a))))
              w)))
    (flat w))))

> (flat-unique '((a b) ((a)) a c))
```

```
(B A C)
```

Решение 4.198.3

```
(defun flat-unique (w)
  (remove-duplicates
   (labels ((flat (w)
              (loop for a in w
                    if (listp a) append (flat a)
                    else collect a)))
     (flat w))))
```

```
> (flat-unique '((a b) ((a)) a c))
(B A C)
```

Решение 4.198.4

```
(defun flat-unique (w &optional ac)
  (cond ((null w) ac)
        ((atom w) (adjoin w ac))
        ((flat-unique (car w) (flat-unique (cdr w) ac)))))
```

```
> (flat-unique '((a b) ((a)) a c))
(B A C)
```

Задача 4.199 compress.lisp

Определить функцию, которая оставляет в списке w из каждой группы подряд идущих одинаковых элементов только один.

Решение 4.199.1

```
(defun compress (w &optional b)
  (cond ((null w) nil)
        ((equal (car w) b) (compress (cdr w) b))
        ((cons (car w) (compress (cdr w) (car w)))))
```

```
> (compress '(a a b c c c d e e))
(A B C D E)
> (compress '(a b))
(A B)
> (compress '(a a))
(A)
> (compress '(a))
(A)
> (compress '())
NIL
```

Со знака & начинаются спецификаторы объявляемых переменных, например &optional (необязательные) или &aux - (вспомогательные). Применение &optional можно заменить созданием функции-обертки:

Решение 4.199.2

```

(defun over-compress (w) (compress w nil))

(defun compress (w b)
  (cond ((null w) nil)
        ((equal (car w) b) (compress (cdr w) b))
        ((cons (car w) (compress (cdr w) (car w))))))

> (compress '(a a b c c c d e e))
(A B C D E)
> (compress '(a b))
(A B)
> (compress '(a a))
(A)
> (compress '(a))
(A)
> (compress '())
NIL

```

Решение 4.199.3

```

(defun compress (w)
  (cond ((null (cdr w)) w)
        ((equal (car w) (cadr w)) (compress (cdr w)))
        ((cons (car w) (compress (cdr w))))))

> (compress '(a a b c c c d e e))
(A B C D E)
> (compress '(a b))
(A B)
> (compress '(a a))
(A)
> (compress '(a))
(A)
> (compress '())
NIL

```

Решение 4.199.4

```

(defun compress (w)
  (cond ((null w) nil)
        ((equal (car w) (cadr w)) (compress (cdr w)))
        ((cons (car w)
                  (labels ((press (w b)
                           (cond ((null w) nil)
                                 ((equal (car w) b) (press (cdr w) b))
                                 ((cons (car w)
                                         (press (cdr w) (car w))))))
                    (press (cdr w) nil))))))

> (compress '(nil nil nil a a b c nil nil nil c c d e e nil nil nil))
(NIL A B C NIL C D E NIL)

```

Решение 4.199.5

```
(defun compress (w &optional b f)
  (cond ((null w) nil)
        (f (if (equal (car w) b)
                (compress (cdr w) b t)
                (cons (car w) (compress (cdr w) (car w) t))))
        ((cons (car w) (compress (cdr w) (car w) t)))))

> (compress '(nil nil nil a a b c nil nil nil c c d e e nil nil nil))
(NIL A B C NIL C D E NIL)
```

Решение 4.199.6

```
(defun compress (w &optional b f)
  (cond ((null w) nil)
        ((and f (equal (car w) b)) (compress (cdr w) b t))
        ((cons (car w) (compress (cdr w) (car w) t)))))

> (compress '(nil nil nil a a b c nil nil nil c c d e e nil nil nil))
(NIL A B C NIL C D E NIL)
```

Задача 4.200 del-deviant.lisp

Определить функцию, которая удаляет из числового списка числа, нарушающие возрастающий порядок.

Решение 4.200.1

```
(defun del-deviant (w &optional (b (car w)))
  (cond ((null w) nil)
        ((> b (car w)) (del-deviant (cdr w) b))
        ((cons (car w) (del-deviant (cdr w) (car w)))))

> (del-deviant '(1 2 3 4 1))
(1 2 3 4)
> (del-deviant '(1 2 3 1 4))
(1 2 3 4)
```

Решение 4.200.2

```
(defun del-deviant (w)
  (if (cdr w)
      (if (<= (car w) (cadr w))
          (cons (car w) (del-deviant (cdr w)))
          (del-deviant (cons (car w) (cddr w))))
      w))

> (del-deviant '(1 2 3 4 1))
(1 2 3 4)
> (del-deviant '(1 2 3 1 4))
```

```
(1 2 3 4)
```

Решение 4.200.3

```
(defun del (w a) (if (< a (car w)) w (cons a w)))

(defun del-deviant (w)
  (when w
    (nreverse
     (reduce #'del (cdr w) :initial-value `((, (car w)))))))

> (del-deviant '(1 2 3 1 4))
(1 2 3 4)
> (del-deviant '(1 2 3 4 1))
(1 2 3 4)
```

Решение 4.200.4

```
(defun del-deviant (w)
  (cond ((null (cdr w)) w)
        ((< (cadr w) (car w)) (del-deviant (cons (car w) (cddr w))))
        ((cons (car w) (del-deviant (cdr w))))))

> (del-deviant '(1 2 3 1 4))
(1 2 3 4)
> (del-deviant '(1 2 3 4 1))
(1 2 3 4)
```

Задача 4.201 maxeq.lisp

Определить функцию, которая находит в одноуровневом числовом списке возрастающий подсписок максимальной длины.

Решение 4.201.1

```
(defun seq (w &optional (b (car w)))
  (cond ((null w) nil)
        ((> b (car w)) nil)
        ((cons (car w) (seq (cdr w) (car w))))))

(defun maxeq (w &optional ac &aux (v (seq w)))
  (cond ((null w) ac)
        ((> (length v) (length ac)) (maxeq (cdr w) v))
        ((maxeq (cdr w) ac))))

> (maxeq '(1 2 3 1 2 3 4 1 2))
(1 2 3 4)
```

Решение 4.201.2

```
(defun seq (w &optional (b (car w)))
  (when w (when (<= b (car w)) (cons (car w) (seq (cdr w) (car w))))))
```

```
(defun maxeq (w &optional ac &aux (v (seq w)))
  (cond ((null w) ac)
        ((> (length v) (length ac)) (maxeq (cdr w) v))
        ((maxeq (cdr w) ac))))
```

```
> (maxeq '(1 2 3 1 2 3 4 1 2))
(1 2 3 4)
```

Решение 4.201.3

```
(defun seq (w &optional (b (car w)))
  (when w (when (<= b (car w)) (cons (car w)
                                       (seq (cdr w) (car w))))))
```

```
(defun maxeq (w &optional ac &aux (v (seq w)))
  (if w
      (if (> (length v) (length ac))
          (maxeq (cdr w) v)
          (maxeq (cdr w) ac))
      ac))
```

```
> (maxeq '(1 2 3 1 2 3 4 1 2))
(1 2 3 4)
```

Решение 4.201.4

```
(defun seq (w &optional (b (car w)))
  (when w (when (<= b (car w)) (cons (car w)
                                       (seq (cdr w) (car w))))))
```

```
(defun maxeq (w &optional ac &aux (v (seq w)))
  (if w (maxeq (cdr w) (if (> (length v) (length ac)) v ac)) ac))
```

```
> (maxeq '(1 2 3 1 2 3 4 1 2))
(1 2 3 4)
```

Решение 4.201.5

```
(defun seq (w &optional (b (car w)))
  (when (and w (<= b (car w))) (cons (car w)
                                       (seq (cdr w) (car w))))))
```

```
(defun maxeq (w &optional ac &aux (v (seq w)))
  (if w (maxeq (cdr w) (if (> (length v) (length ac)) v ac)) ac))
```

```
> (maxeq '(1 2 3 1 2 3 4 1 2))
(1 2 3 4)
```

Решение 4.201.6

```
(defun check (w v)
  (cond ((null w) nil)
```



```

((null v) t)
((check (cdr w) (cdr v))))))

(defun seq (w &optional (b (car w)))
  (when (and w (<= b (car w))) (cons (car w) (seq (cdr w) (car w)))))

(defun maxeq (w &optional ac &aux (v (seq w)))
  (if w (maxeq (cdr w) (if (check v ac) v ac)) ac))

> (maxeq '(1 2 3 1 2 3 4 1 2))
(1 2 3 4)

```

Решение 4.201.7

```

(defun check (w v)
  (when w (if v (check (cdr w) (cdr v)) t)))

(defun seq (w &optional (b (car w)))
  (when (and w (<= b (car w))) (cons (car w) (seq (cdr w) (car w)))))

(defun maxeq (w &optional ac &aux (v (seq w)))
  (if w (maxeq (cdr w) (if (check v ac) v ac)) ac))

> (maxeq '(1 2 3 1 2 3 4 1 2))
(1 2 3 4)

```

Решение 4.201.8

```

(defun check (w v)
  (when w (if v (check (cdr w) (cdr v)) t)))

(defun seq (w &optional (b (car w)) &aux (a (car w)))
  (when (and w (<= b a)) (cons a (seq (cdr w) a))))

(defun maxeq (w &optional ac &aux (v (seq w)))
  (if w (maxeq (cdr w)
    (if (check v ac) v ac)) ac))

> (maxeq '(1 2 3 1 2 3 4 1 2))
(1 2 3 4)

```

Задача 4.202 max-chain.lisp

Дан список, элементами которого являются списки, состоящие из трех элементов: две буквы и одна цифра. Необходимо составить цепочки из списков (по буквам без учета цифр) и вывести ту цепочку, сумма третьих элементов которых максимальная.

Решение 4.202.1

```

(defun del (v w)
  (cond ((null w) nil)

```

```

      ((equal (car w) v) (del v (cdr w)))
      ((cons (car w) (del v (cdr w))))))

(defun chains (w z)
  (when w (cons (chain w z) (chains (cdr w) z))))

(defun chain (w z
              &optional (b (car w))
              &aux (v (same b z)))
  (if v (cons b (chain (cdr w) (del b (del v z)) v))
      `(.b)))

(defun same (v w)
  (cond ((null w) nil)
        ((eq (cadr v) (caar w)) (car w))
        ((same v (cdr w)))))

(defun lens (w)
  (mapcar #'(lambda (a) (apply #'(mapcar #'caddr a))) w))

(defun max-chain (w &aux (v (chains w w)) (z (lens v)))
  (values v (nth (position (apply #'max z) z) v)))

> (max-chain '((c e 5) (a b 1) (d k 2) (b c 3) (f p 4) (p f 6) (k b
7)))
(((C E 5)) ((A B 1) (B C 3) (C E 5)) ((D K 2) (K B 7) (B C 3) (C E 5))
((B C 3) (C E 5)) ((F P 4) (P F 6)) ((P F 6) (F P 4)) ((K B 7) (B C 3)
(C E 5)))
((D K 2) (K B 7) (B C 3) (C E 5))

```

Задача 4.203 *our-mapcon.lisp*

Построить функцию `mapcon` - аналог `maplist`, но объединяет результаты (подписки) в один список.

Решение 4.203.1

```

(defun our-mapcon (f w)
  (when w (cons (apply f w)
                 (our-mapcon f (cdr w)))))

> (our-mapcon #'list '(1 2 3 4 5))
((1 2 3 4 5) (2 3 4 5) (3 4 5) (4 5) (5))

```

Задача 4.204 *drop-last.lisp*

Определить рекурсивную функцию удаления последнего элемента списка.

Решение 4.204.1

```
(defun drop-last (w)
  (when (cdr w) (cons (car w)
                      (drop-last (cdr w)))))

> (drop-last '(1 2 3 4))
(1 2 3)
```

Задача 4.205 deep-reverse.lisp

Дан многоуровневый список. Построить новый список, изменив порядок элементов в исходном списке на обратный и изменить порядок на обратный в каждом подсписке.

Решение 4.205.1

```
(defun deep-reverse (w &optional acc &aux (a (car w)))
  (if w
      (deep-reverse (cdr w)
                    (push (if (atom a) a (deep-reverse a)) acc))
      acc))

> (deep-reverse '(0 1 (((2 3) 4) 5) 7 (8 (9 10) 11)))
((11 (10 9) 8) 7 (5 (4 (3 2))) 1 0)
```

Решение 4.205.2

```
(defun deep-reverse (w &optional acc)
  (if w
      (deep-reverse (cdr w)
                    (cons (if (atom (car w))
                              (car w)
                              (deep-reverse (car w)))
                          acc))
      acc))

> (deep-reverse '(0 1 (((2 3) 4) 5) 7 (8 (9 10) 11)))
((11 (10 9) 8) 7 (5 (4 (3 2))) 1 0)
```

Решение 4.205.3

```
(defun deep-reverse (w &aux ac)
  (loop for a in w do
    (if (atom a)
        (setf ac (cons a ac))
        (setf ac (cons (deep-reverse a) ac)))
    finally (return ac)))

> (deep-reverse '(0 1 (((2 3) 4) 5) 7 (8 (9 10) 11)))
((11 (10 9) 8) 7 (5 (4 (3 2))) 1 0)
```

Решение 4.205.4 (автор - Catstail, www.cyberforum.ru)

```
(defun deep-reverse (w &aux ac)
  (dolist (a w ac)
    (push (if (atom a) a
              (deep-reverse a)) ac)))

> (deep-reverse '(0 1 ((2 3) 4) 5) 7 (8 (9 10) 11)))
((11 (10 9) 8) 7 (5 (4 (3 2))) 1 0)
```

Решение 4.205.5

```
(defun deep-reverse (w &aux ac)
  (loop for a in w do
    (setf ac (cons (if (atom a) a (deep-reverse a)) ac))
    finally (return ac)))

> (deep-reverse '(0 1 ((2 3) 4) 5) 7 (8 (9 10) 11)))
((11 (10 9) 8) 7 (5 (4 (3 2))) 1 0)
```

Задача 4.206 not-numbers-and-sum.lisp

Определить функцию, возвращающую элементы исходного списка, не являющиеся числами, и сумму всех элементов - чисел.

Решение 4.206.1

```
(defun not-numbers-and-sum (w)
  (values (remove-if #'numberp w)
          (apply #' + (remove-if-not #'numberp w))))

> (not-numbers-and-sum '(1 a (2 3) (a b) 2 (4)))
(A (2 3) (A B) (4))
3
```

Решение 4.206.2

```
(defun not-numbers-and-sum (w)
  (loop for a in w
    if (numberp a) collect a into numbers
    else collect a into notnumbers
    finally return (values notnumbers (apply #' + numbers))))

> (not-numbers-and-sum '(1 a (2 3) (a b) 2 (4)))
(A (2 3) (A B) (4))
3
```

Задача 4.207 elm-share.lisp

Определить функцию, вычисляющую отношения количества атомов - не чисел, чисел, подсписков к общему числу элементов исходного списка у, и записывающую эти отношения в список.

Решение 4.207.1

```
(defun elm-share (w &aux (z (length w)))
  (when w (list (/ (count-if #'symbolp w) z)
                (/ (count-if #'numberp w) z)
                (/ (count-if #'listp w) z))))

> (elm-share '(1 a (2 3) (a b) 2 (4)))
(1/6 1/3 1/2)
```

Задача 4.208 *smash-mx.lisp*

Пусть дана прямоугольная матрица a ($m * n$), элементами которой являются целые числа. Замените все положительные четные числа на числа, являющиеся их "перевертышами". Составьте функцию, получающую для заданного числа его "перевертыш" (число a будем считать "перевертышем" числа b , если, читая a справа на лево, получим число b).

Решение 4.208.1

```
(defun reverse-digits (n &optional (m 0))
  (cond ((zerop n) m)
        (t (multiple-value-bind (q r) (truncate n 10)
              (reverse-digits q (+ (* m 10) r))))))

(defun smash (w &aux (a (car w)))
  (cond ((null w) nil)
        ((and (numberp a) (plusp a) (evenp a))
         (cons (reverse-digits a) (smash (cdr w))))
        ((cons a (smash (cdr w)))))

(defun smash-mx (w) (mapcar #'smash w))

> (smash-mx '((12 45 -72) (a b 15) (49 c 18)))
((21 45 -72) (A B 15) (49 C 81))
```

Решение 4.208.2

```
(defun reverse-digits (n)
  (labels ((next (n v)
             (if (zerop n) v
                 (multiple-value-bind (q r)
                     (truncate n 10)
                     (next q (+ (* v 10) r))))))
    (next n 0)))

(defun smash (w)
  (loop for a in w
        if (and (numberp a) (plusp a) (evenp a))
        collect (reverse-digits a)
        else collect a))
```

```
(defun smash-mx (w)
  (mapcar #'smash w))

> (smash-mx '((12 45 -72) (a b 15) (49 c 18)))
((21 45 -72) (A B 15) (49 C 81))
```

Решение 4.208.3

```
(defun reverse-digits (n)
  (parse-integer (reverse (write-to-string n))))

(defun smash (w)
  (mapcar #'(lambda (a)
    (if (and (numberp a) (plusp a) (evenp a))
        (reverse-digits a)
        a))
    w))

(defun smash-mx (w)
  (mapcar #'smash w))

> (smash-mx '((12 45 -72) (a b 15) (49 c 18)))
((21 45 -72) (A B 15) (49 C 81))
```

Далее, в одну функцию (плохой стиль):

Решение 4.208.4

```
(defun smash-mx (w)
  (loop for v in w
    collect
      (loop for a in v
        if (and (numberp a) (plusp a) (evenp a))
        collect (parse-integer (reverse (write-to-string a)))
        else collect a)))

> (smash-mx '((12 45 -72) (a b 15) (49 c 18)))
((21 45 -72) (A B 15) (49 C 81))
```

Решение 4.208.5

```
(defun smash-mx (w)
  (mapcar #'
    (lambda (v)
      (mapcar #'
        (lambda (a) (if (and (numberp a) (plusp a) (evenp a))
          (parse-integer (reverse (write-to-string a)))
          a))
        v))
    w))

> (smash-mx '((12 45 -72) (a b 15) (49 c 18)))
((21 45 -72) (A B 15) (49 C 81))
```

Задача 4.209 primes.lisp

Определить функцию, которая проверяет, являются ли элементы списка простыми числами (то есть, делящимися нацело только на себя и на единицу). Если элемент является простым числом, функция возвращает в результирующем списке значение Т. Если элемент не является простым числом, функция возвращает в результирующем списке NIL.

Решение 4.209.1

```
(defun primep (n)
  (loop for i from 2 to (isqrt n)
        never (zerop (rem n i))))

(defun primes (w) (mapcar #'primep w))

> (primes '(2 3 4 5 6 7 8 9 10 11))
(T T NIL T NIL T NIL NIL NIL T)
```

Решение 4.209.2

```
(defun primes (w)
  (mapcar #'(lambda (n) (loop for i from 2 to (isqrt n)
                              never (zerop (rem n i)))) w))

> (primes '(2 3 4 5 6 7 8 9 10 11))
(T T NIL T NIL T NIL NIL NIL T)
```

Задача 4.210 all-chains.lisp

Сформировать такие цепочки из списка списков, в которых последний элемент предыдущего списка совпадает с первым следующего. Пример: ((a 2) (n 6) (2 f) (a v) (6 b) (f p) (d a)) => ((a 2) (2 f) (f p)) и ((n 6) (6 b)) и т. п. И чтобы подписки, которые уже включены в какую-нибудь из цепочек, сразу после этого удалялись из основного списка.

Решение 4.210.1

```
(defun del (v w)
  (cond ((null w) nil)
        ((equal (car w) v) (del v (cdr w)))
        ((cons (car w) (del v (cdr w))))))

(defun chains (w z)
  (when w (cons (chain w z) (chains (cdr w) z))))

(defun clear (v w)
  (cond ((null v) nil)
        ((some #'(lambda (e) (subsetp (car v) e :test #'equalp))
                (remove (car v) w))
         (clear (cdr v) w))
        ((cons (car v) (clear (cdr v) w)))))
```

```
(defun chain (w z &optional (b (car w)) &aux (v (same b z)))
  (if v (cons b (chain (cdr w) (del b (del v z)) v)) ` (,b)))
```

```
(defun same (v w)
  (cond ((null w) nil)
        ((eq (cadr v) (caar w)) (car w))
        ((same v (cdr w)))))
```

```
(defun all-chains (w &aux (z (chains w w)))
  (delete-if-not #'cdr (clear z z)))
```

```
> (all-chains '((a 2) (n 6) (2 f) (a v) (6 b) (f p) (d a)))
((N 6) (6 B)) ((D A) (A 2) (2 F) (F P))
```

Решение 4.210.2

```
(defun chains (w z) (when w (cons (chain w z) (chains (cdr w) z))))
```

```
(defun clear (v w)
  (cond ((null v) nil)
        ((some #'(lambda (e) (subsetp (car v) e :test #'equal))
                 (remove (car v) w))
         (clear (cdr v) w))
        ((cons (car v) (clear (cdr v) w)))))
```

```
(defun chain (w z &optional (b (car w)) &aux (v (same b z)))
  (if v (cons b (chain (cdr w) (remove b (remove v z)) v)) ` (,b)))
```

```
(defun same (v w)
  (cond ((null w) nil)
        ((eq (cadr v) (caar w)) (car w))
        ((same v (cdr w)))))
```

```
(defun all-chains (w
                   &aux (z (chains w w)))
  (delete-if-not #'cdr (clear z z)))
```

```
> (all-chains '((a 2) (n 6) (2 f) (a v) (6 b) (f p) (d a)))
((N 6) (6 B)) ((D A) (A 2) (2 F) (F P))
```

Задача 4.211 reverse-letters.lisp

В списке слова зашифрованы - каждое из них написано наоборот. Написать программу расшифровки.

Решение 4.211.1

```
(defun reverse-letters (w) (mapcar #'reverse w))
```

```
> (reverse-letters '("latrommi" "sdeed"))
("immortal" "deeds")
```


Задача 4.212 reverse-words.lisp

В строке слова зашифрованы - каждое из них написано наоборот. Написать программу расшифровки.

Решение 4.212.1

```
(defun reverse-words (s)
  (string-right-trim
   " "
   (format
    nil "~{~s ~}"
    (mapcar
     #'intern
     (mapcar
      #'reverse
      (mapcar
       #'string
       (read-from-string
        (concatenate
         'string
         "(" s ")")))))))))

> (time (reverse-words "latrommi sdeed"))
Space: 1340 Bytes
"IMMORTAL DEEDS"
```

Решение 4.212.2

```
(defun reverse-words (s)
  (string-left-trim
   " "
   (apply
    #'concatenate
    'string
    (mapcar
     #'reverse
     (mapcar
      #'(lambda (a) (concatenate 'string a " "))
      (mapcar
       #'string
       (read-from-string
        (concatenate 'string "(" s ")")))))))))

> (time (reverse-words "latrommi sdeed"))
Space: 492 Bytes
"IMMORTAL DEEDS"
```

Решение 4.212.3

```
(defun rv (w &optional ac &aux (a (car w)))
  (cond ((null w) ac)
```

```

      ((alpha-char-p a) (rv (cdr w) (cons a ac)))
      ((nconc ac (cons a (rv (cdr w) nil))))))

(defun reverse-words (s)
  (coerce (rv (loop for c across s collect c))
    'string))

> (time (reverse-words "latrommi sdeed"))
Space: 288 Bytes
"immortal deeds"

```

Решение 4.212.4

```

(defun rv (w &optional ac &aux (a (car w)))
  (cond ((null w) ac)
        ((alpha-char-p a) (rv (cdr w) (cons a ac)))
        ((nconc ac (cons a (rv (cdr w) nil))))))

(defun reverse-words (s)
  (coerce (rv (map 'list #'identity s)) 'string))

> (time (reverse-words "latrommi sdeed"))
Space: 288 Bytes
"immortal deeds"

```

Решение 4.212.5

```

(defun rv (w &optional ac &aux (a (car w)))
  (cond ((null w) ac)
        ((alpha-char-p a) (rv (cdr w) (cons a ac)))
        ((nconc ac (cons a (rv (cdr w) nil))))))

(defun reverse-words (s) (coerce (rv (coerce s 'list)) 'string))

> (time (reverse-words "latrommi sdeed"))
Space: 288 Bytes
"immortal deeds"

```

Задача 4.213 reverse-input.lisp

Вводимые слова зашифрованы – каждое из них написано наоборот. Написать программу расшифровки.

Решение 4.213.1

```

(defun reverse-input ()
  (string-right-trim
    " "
    (format
      nil "~{~s ~}"
      (mapcar
        #'intern

```

```

    (mapcar
      #'reverse
      (mapcar
        #'string
        (read-from-string
          (concatenate
            'string
            "(" (read-line) ")"))))))))
> (reverse-input)
latrommi sdeed
"IMMORTAL DEEDS"

```

Решение 4.213.1

```

(defun rv (w &optional ac)
  (cond ((null w) ac)
        ((alpha-char-p (car w)) (rv (cdr w) (cons (car w) ac)))
        ((nconc ac (cons (car w) (rv (cdr w) nil))))))

(defun reverse-input ()
  (coerce (rv (coerce (read-line) 'list)) 'string))

> (reverse-input)
latrommi sdeed
"immortal deeds"

```

Задача 4.214 eventh-min.lisp

Заменить значения элементов с нечетными номерами на значение минимального положительного элемента.

Решение 4.214.1

```

(defun eventh-min (w) (ev w (apply #'min (remove-if-not #'plusp w))))

(defun ev (w m)
  (if (cdr w) (cons (car w) (cons m (ev (cddr w) m))) w))

> (eventh-min '(0 1 2 3 4 5 6 7 8))
(0 1 2 1 4 1 6 1 8)
> (eventh-min '(0 1 2 3 4 5 6 7))
(0 1 2 1 4 1 6 1)

```

Задача 4.215 second-half.lisp

Определить функцию, возвращающую вторую половину списка.

Решение 4.215.1

```

(defun second-half (w) (nthcdr (truncate (/ (length w) 2)) w))

```

```
> (second-half '(1 2 3 4 5 6))
(4 5 6)
> (second-half '(1 2 3 4 5 6 7))
(4 5 6 7)
```

Задача 4.216 compress&ratio.lisp

Составьте функцию "сжатия" исходной последовательности символов, которая заменяет последовательность, состоящую из одинаковых символов, текстом вида $x(k)$, где x -символ последовательности, k -число вхождений. Определите для указанной последовательности коэффициент сжатия (отношение исходной длины последовательности к полученной).

Решение 4.216.1

```
(defun compress (w &optional (n 1) ac (m (count-nil (reverse w))))
  (cond ((null w) (if (zerop m)
                      (reverse ac)
                      (if (= m 1)
                          (reverse (cons nil ac))
                          (reverse (cons (list nil m) ac)))))
        ((equal (car w) (cadr w)) (compress (cdr w) (1+ n) ac m))
        ((> n 1) (compress (cdr w) 1 (cons (list (car w) n) ac) m))
        ((compress (cdr w) 1 (cons (car w) ac) m))))

(defun count-nil (w)
  (if (or (null w) (car w)) 0 (1+ (count-nil (cdr w)))))

(defun compress&ratio (w &aux (v (compress w)))
  (values v (float (/ (length w) (length v)))))

> (compress&ratio '(nil nil a b b b c c c d nil nil nil))
(NIL 2) A (B 3) (C 3) D (NIL 3)
2.1666667
```

Решение 4.216.2

```
(defun compress (w &optional (n 1) ac (m (count-nil (reverse w))))
  (cond ((null w)
        (if (zerop m)
            (reverse ac)
            (reverse (cons (if (= m 1) nil (list nil m)) ac))))
        ((equal (car w) (cadr w)) (compress (cdr w) (1+ n) ac m))
        ((> n 1) (compress (cdr w) 1 (cons (list (car w) n) ac) m))
        ((compress (cdr w) 1 (cons (car w) ac) m))))

(defun count-nil (w)
  (if (or (null w) (car w)) 0 (1+ (count-nil (cdr w)))))

(defun compress&ratio (w &aux (v (compress w)))
  (values v (float (/ (length w) (length v)))))
```

```
> (compress&ratio '(nil nil a b b b c c c d nil nil nil))
((NIL 2) A (B 3) (C 3) D (NIL 3))
2.1666667
Решение 4.216.3
```

```
(defun compress (w &optional (n 1) ac)
  (cond ((null w) (reverse ac))
        ((equal w '(nil))
         (compress (cdr w) 1 (cons (if (= n 1)
                                         (car w)
                                         (list (car w) n))
                                     ac)))
        ((equal (car w) (cadr w)) (compress (cdr w) (1+ n) ac))
        ((> n 1) (compress (cdr w) 1 (cons (list (car w) n) ac)))
        ((compress (cdr w) 1 (cons (car w) ac)))))
```

```
(defun compress&ratio (w &aux (v (compress w)))
  (values v (float (/ (length w) (length v)))))
```

```
> (compress&ratio '(nil nil a b b b c c c d nil nil nil))
((NIL 2) A (B 3) (C 3) D (NIL 3))
2.1666667
```

Решение 4.216.4

```
(defun mp (w &optional (n 1) ac)
  (cond ((null w) (reverse ac))
        ((or (not (eq (car w) (cadr w))) (equal w '(nil)))
         (mp (cdr w) 1 (cons (if (= n 1) (car w) (list (car w) n))
                               ac)))
        ((mp (cdr w) (1+ n) ac))))
```

```
(defun compress&ratio (w &aux (v (mp w)))
  (values v (float (/ (length w) (length v)))))
```

```
> (compress&ratio '(nil nil a b b b c c c d nil nil nil))
((NIL 2) A (B 3) (C 3) D (NIL 3))
2.1666667
```

Задача 4.217 *plus-num-sum.lisp*

Определить количество положительных элементов одномерного массива. Вывести на экран их количество и значение их суммы.

Решение 4.217.1

```
(defun cnt (w)
  (cond ((null w) 0)
        ((plusp (car w)) (1+ (cnt (cdr w))))
        ((cnt (cdr w)))))
```

```
(defun sum (w)
  (cond ((null w) 0)
        ((plusp (car w)) (+ (car w) (sum (cdr w))))
        ((sum (cdr w)))))
```

```
(defun plus-num-sum (w) (values (cnt w) (sum w)))
```

```
> (plus-num-sum '(-1 0 1 2 3))
3
6
```

Решение 4.217.2

```
(defun plus-num-sum (w &aux (v (delete-if-not #'plusp w)))
  (values (length v) (apply #' + v)))
```

```
> (plus-num-sum '(-1 0 1 2 3))
3
6
```

Решение 4.217.3 * для списков с числом элементов больше 4096

```
(defun plus-num-sum (w &aux (v (delete-if-not #'plusp w)))
  (values (length v) (reduce #' + v)))
```

```
> (plus-num-sum '(-1 0 1 2 3))
3
6
```

Задача 4.218 div3.lisp

Выбрать из числового списка все элементы, делящиеся на 3, и сформировать из них список.

Решение 4.218.1

```
(defun div3 (w)
  (cond ((null w) nil)
        ((zerop (rem (car w) 3)) (cons (car w) (div3 (cdr w))))
        ((div3 (cdr w)))))
```

```
> (div3 '(1 2 3 4 5 6 7 8 9 10))
(3 6 9)
```

Решение 4.218.2

```
(defun div3 (w)
  (remove-if-not #'(lambda (a) (zerop (rem a 3))) w))
```

```
> (div3 '(1 2 3 4 5 6 7 8 9 10))
(3 6 9)
```

Задача 4.219 div.lisp

Выбрать из числового списка все элементы, делящиеся на n , и сформировать из них список.

Решение 4.219.1

```
(defun div (w n)
  (cond ((null w) nil)
        ((zerop (rem (car w) n)) (cons (car w) (div (cdr w) n)))
        ((div (cdr w) n))))

> (div '(1 2 3 4 5 6 7 8 9 10) 3)
(3 6 9)
> (div '(1 2 3 4 5 6 7 8 9 10) 4)
(4 8)
```

Решение 4.219.2

```
(defun div (w n)
  (remove-if-not #'(lambda (a) (zerop (rem a n))) w))

> (div '(1 2 3 4 5 6 7 8 9 10) 3)
(3 6 9)
> (div '(1 2 3 4 5 6 7 8 9 10) 4)
(4 8)
```

Решение 4.219.3

```
(defun div (w n)
  (loop for a in w when (zerop (rem a n)) collect a))

> (div '(1 2 3 4 5 6 7 8 9 10) 3)
(3 6 9)
> (div '(1 2 3 4 5 6 7 8 9 10) 4)
(4 8)
```

Задача 4.220 column-plus-sum-num.lisp

Вычислить сумму и число положительных элементов каждого столбца матрицы.

Решение 4.220.1

```
(defun transpose (m) (apply #'mapcar #'list m))

(defun plus-num (w)
  (mapcar #'(lambda (a) (count-if #'plusp a)) w))

(defun plus-sum (w)
  (mapcar #'(lambda (e) (apply #'+ e))
    (mapcar #'(lambda (a) (delete-if-not #'plusp a)) w)))
```

```
(defun column-plus-sum-num (w &aux (v (transpose w)))
  (values (plus-sum v) (plus-num v)))

> (column-plus-sum-num '((-1 2 3) (4 -5 6) (7 8 -9)))
(11 10 9)
(2 2 2)
```

Решение 4.220.2

```
(defun transpose (m) (apply #'mapcar #'list m))

(defun plus-num (w)
  (mapcar #'(lambda (a) (length a)) w))

(defun plus-sum (w)
  (mapcar #'(lambda (e) (apply #'+ e)) w))

(defun column-plus-sum-num
  (w &aux (v (mapcar
    #'(lambda (u) (remove-if-not #'plusp u))
    (transpose w))))
  (values (plus-sum v) (plus-num v)))

> (column-plus-sum-num '((-1 2 3) (4 -5 6) (7 8 -9)))
(11 10 9)
(2 2 2)
```

Задача 4.221 *nx-insert.lisp*

Дан список чисел. Добавить элемент с максимальным значением после минимального элемента.

Решение 4.221.1

```
(defun nx-insert (w)
  (insert w
    (reduce #'min w)
    (reduce #'max w)))

(defun insert (w n x)
  (cond ((null w) nil)
        ((= (car w) n) (cons n (cons x (insert (cdr w) n x))))
        ((cons (car w) (insert (cdr w) n x)))))

> (nx-insert '(0 1 2 5 1 2 3 0 1 2 3 0 4))
(0 5 1 2 5 1 2 3 0 5 1 2 3 0 5 4)
```

Решение 4.221.2

```
(defun nx-insert (w
  &optional
```



```

      (n (reduce #'min w))
      (x (reduce #'max w)))
(cond ((null w) nil)
      ((= (car w) n) (cons n (cons x (nx-insert (cdr w) n x))))
      ((cons (car w) (nx-insert (cdr w) n x)))))

> (nx-insert '(0 1 2 5 1 2 3 0 1 2 3 0 4))
(0 5 1 2 5 1 2 3 0 5 1 2 3 0 5 4)

```

Задача 4.222 max-min-mx.lisp

It just so happens that Emacs was a product of the MIT AI Lab, so when Richard Stallman created the first modern version of Emacs (ie GNU Emacs v1), Lisp was the logical choice based on the environment he was used to. I imagine if he'd worked at PARC instead of MIT it would have been a Smalltalk dialect. (Or maybe REXX at IBM, or (shudder) Fortran at Honeywell.) Stallman's promotion of Lisp as the One True Way over the years aside, Lisp really was one perfectly good choice among many, and a whole hell of a lot better than TECO, in which Emacs was originally implemented.

Brian Connors

Найти в каждой строке матрицы максимальный и минимальный элементы и поместить их на место первого и последнего элемента строки соответственно.

Решение 4.222.1

```

(defun max-min-mx (w) (mapcar #'max-min w))

(defun max-min (w &aux (x (apply #'max w)) (n (apply #'min w)))
  (cons x (append (butlast (cdr w)) `,(n))))

> (max-min-mx '((1 2 3) (4 5 6) (7 8 9)))
((3 2 1) (6 5 4) (9 8 7))

```

Решение 4.222.2

```

(defun max-min (w)
  (cons (apply #'max w)
        (nconc (butlast (cdr w)) `,(apply #'min w))))

(defun max-min-mx (w)
  (mapcar #'max-min w))

> (max-min-mx '((1 2 3) (4 5 6) (7 8 9)))

```

```
((3 2 1) (6 5 4) (9 8 7))
```

Решение 4.222.3

```
(defun max-min-mx (w)
  (mapcar #'(lambda (v) (cons (apply #'max v)
                              (nconc (butlast (cdr v))
                                     `((, (apply #'min v))))))
    w))

> (max-min-mx '((1 2 3) (4 5 6) (7 8 9)))
((3 2 1) (6 5 4) (9 8 7))
```

Задача 4.223 sets.lisp

Для заданного атомного списка построить список всех подмножеств этого списка.

Решение 4.223.1

```
(defun sets (w)
  (if w ((lambda (r)
            (nconc r (mapcar #'(lambda (a) (cons (car w) a)) r)))
        (sets (cdr w)))
    '(nil)))

> (sets '(a b c))
(NIL (C) (B) (B C) (A) (A C) (A B) (A B C))
```

Задача 4.224 integers.lisp

Определить функцию, проверяющую являются ли элементы списка натуральными числами. Если элемент является натуральным числом, функция возвращает в результирующем списке значение Т. Если элемент не является натуральным числом, функция возвращает в результирующем списке NIL.

Решение 4.224.1

```
(defun integers (w) (mapcar #'integerp w))

> (integers '(1 2 3 a a 2.3 4 5.6))
(T T T NIL NIL NIL T NIL)
```

Решение 4.224.2

```
(defun integers (w)
  (loop for a in w collect (integerp a)))

> (integers '(1 2 3 a a 2.3 4 5.6))
(T T T NIL NIL NIL T NIL)
```

Решение 4.224.3

```
(defun integers (w)
  (when w (cons (integerp (car w)) (integers (cdr w)))))

> (integers '(1 2 3 a a 2.3 4 5.6))
(T T T NIL NIL NIL T NIL)
```

Задача 4.225 rotate.lisp

Написать функцию, меняющую местами элементы списка w с номерами n и m.

Решение 4.225.1

```
(defun rotate (w a b &optional ac n acc f)
  (cond ((zerop b)
        (nreconc ac (cons (car w) (nreconc acc (cons n (cdr w))))))
        (f (rotate (cdr w) 0 (1- b) ac n (cons (car w) acc) t))
        ((zerop a) (rotate (cdr w) 0 (1- b) ac (car w) acc t))
        ((rotate (cdr w) (1- a) (1- b) (cons (car w) ac) n acc nil))))

> (setf w '(1 2 3 4 5 6 7 8 9))
(1 2 3 4 5 6 7 8 9)
> (rotate w 3 8)
(1 2 3 9 5 6 7 8 4)
> w
(1 2 3 4 5 6 7 8 9)
```

Решение 4.225.2 (автор - VH, www.cyberforum.ru)

```
(defun rotate (w a b)
  (when w
    (if (zerop a)
        (if (zerop b)
            w
            ((lambda (result)
              (if (= b 1)
                  (cons (cadr w) result)
                  (cons (car result) (cons (cadr w) (cdr result))))))
        (rotate (cons (car w) (cddr w)) 0 (1- b))))
    (cons (car w) (rotate (cdr w) (1- a) (1- b)))))

> (setf w '(1 2 3 4 5 6 7 8 9))
(1 2 3 4 5 6 7 8 9)
> (rotate w 3 8)
(1 2 3 9 5 6 7 8 4)
> w
(1 2 3 4 5 6 7 8 9)
```

Решение 4.225.3

```
(defun rotate (w a b)
```

```

(when w
  (if (zerop a)
      (if (zerop b)
          w
          (labels ((next (r)
                     (nconc (if (= b 1)
                                (list (cadr w) (car r))
                                (list (car r) (cadr w)))
                             (cdr r))))
      (next (rotate (remove (cadr w) w :count 1) 0 (1- b)))))
  (cons (car w) (rotate (cdr w) (1- a) (1- b)))))

> (setf w '(1 2 3 4 5 6 7 8 9 10))
(1 2 3 4 5 6 7 8 9 10)
> (rotate w 3 8)
(1 2 3 9 5 6 7 8 4 10)
> w
(1 2 3 4 5 6 7 8 9)

```

Решение 4.225.4

```

(defun rotate (w a b)
  (when w
    (if (zerop a)
        (if (zerop b)
            w
            (labels ((next (r)
                       (nconc (if (= b 1)
                                  `((, (cadr w) , (car r))
                                  `((, (car r) , (cadr w)))
                                  (cdr r))))
                          (next (rotate (remove (cadr w) w :count 1) 0 (1- b)))))
        (cons (car w) (rotate (cdr w) (1- a) (1- b)))))

> (setf w '(1 2 3 4 5 6 7 8 9 10))
(1 2 3 4 5 6 7 8 9 10)
> (rotate w 3 8)
(1 2 3 9 5 6 7 8 4 10)
> w
(1 2 3 4 5 6 7 8 9 10)

```

Решение 4.225.5

```

(defun rotate (w a b)
  (when w
    (if (zerop a)
        (if (zerop b)
            w
            (labels ((next (r &aux (v (list (car r) (cadr w))))
                       (nconc (if (= b 1)
                                  (reverse v)
                                  v)
                               (cdr r))))
        (cons (car w) (rotate (cdr w) (1- a) (1- b)))))

```

```

      (next (rotate (remove (cadr w) w :count 1) 0 (1- b))))
    (cons (car w) (rotate (cdr w) (1- a) (1- b)))))

```

```

> (setf w '(1 2 3 4 5 6 7 8 9 10))
(1 2 3 4 5 6 7 8 9 10)
> (rotate w 3 8)
(1 2 3 9 5 6 7 8 4 10)
> w
(1 2 3 4 5 6 7 8 9 10)

```

Решение 4.225.6

```

(defun rotate (w a b)
  (when w
    (if (zerop a)
        (if (zerop b)
            w
            (labels ((next (r &aux (v `(, (car r) , (cadr w))))
                      (nconc (if (= b 1) (reverse v) v) (cdr r))))
          (next (rotate (remove (cadr w) w :count 1) 0 (1- b)))))
        (cons (car w) (rotate (cdr w) (1- a) (1- b))))))

> (setf w '(1 2 3 4 5 6 7 8 9 10))
(1 2 3 4 5 6 7 8 9 10)
> (rotate w 3 8)
(1 2 3 9 5 6 7 8 4 10)
> (1 2 3 4 5 6 7 8 9 10)

```

Решение 4.225.7

```

(defun rotate (w a b)
  (when w
    (if (zerop a)
        (if (zerop b)
            w
            (labels ((next (r &aux (v `(, (car r) , (cadr w))))
                      (nconc (if (= b 1) (reverse v) v) (cdr r))))
          (next (rotate (cons (car w) (cddr w)) 0 (1- b)))))
        (cons (car w) (rotate (cdr w) (1- a) (1- b))))))

> (setf w '(1 2 3 4 5 6 7 8 9 10))
(1 2 3 4 5 6 7 8 9 10)
> (rotate w 3 8)
(1 2 3 9 5 6 7 8 4 10)
> (1 2 3 4 5 6 7 8 9 10)

```

Задача 4.226 nrotate.lisp

Написать функцию, меняющую местами элементы с номерами n и m , разрушающую исходный список w .

Решение 4.226.1

```
(defun nrotate (w n m)
  (progn (rotatef (nth n w) (nth m w)) w))

> (setf w '(1 2 3 4 5 6 7 8 9))
(1 2 3 4 5 6 7 8 9)
```

Задача 4.227 drop-center.lisp

Дан список, элементами которого являются списки. Удалить из каждого подсписка элементы, расположенные посередине, но только если длина соответствующего подсписка нечетная.

Решение 4.227.1

```
(defun drop-center (w &aux (a (car w)) (n (length a)))
  (when w (cons (if (zerop (rem n 2))
                    a
                    (let ((m (truncate (/ n 2))))
                      (append (subseq a 0 m) (subseq a (1+ m))))))
                (drop-center (cdr w)))))

> (drop-center '((3 7) (2 7 4) (2 9 5) (7 1) (8 9 2 4 5)))
((3 7) (2 4) (2 5) (7 1) (8 9 4 5))
```

Решение 4.227.2

```
(defun drop-center (w)
  (mapcar
   #'(lambda (a)
       (if (zerop (rem (length a) 2))
           a
           (append (subseq a 0 (truncate (/ (length a) 2)))
                   (subseq a (1+ (truncate (/ (length a) 2)))))))
   w))

> (drop-center '((3 7) (2 7 4) (2 9 5) (7 1) (8 9 2 4 5)))
((3 7) (2 4) (2 5) (7 1) (8 9 4 5))
```

Решение 4.227.3

```
(defun drop-center (w)
  (mapcar
   #'(lambda (a)
       (let* ((n (length a)) (m (truncate (/ n 2))))
         (if (zerop (rem n 2))
             a
             (append (subseq a 0 m)
                     (subseq a (1+ m))))))
   w))

> (drop-center '((3 7) (2 7 4) (2 9 5) (7 1) (8 9 2 4 5)))
```

```
((3 7) (2 4) (2 5) (7 1) (8 9 4 5))
```

Задача 4.228 *main-dmax.lisp*

What makes Lisp interesting is its ability to handle data structures in a freeform manner. There's really only two data structures in Lisp, the atom and the cons cell, and a list is just a string of cons cells. When dealing with data with no specific form, where type checking could become a liability or at least an unnecessary annoyance, this can be useful... and as it happens, artificial intelligence is pretty much exactly that kind of field, which is why Lisp became its lingua franca.

Brian Connors

Дана матрица. Найти в каждой строке наибольший элемент и элемент главной диагонали и поменять их местами.

Решение 4.228.1

```
(defun main-dmax (w
                  &optional (n 0)
                  &aux (a (car w)))
  (when w (cons (progn
                  (rotatef (nth (position (reduce #'max a) a) a)
                            (nth n a))
                  a)
                (main-dmax (cdr w) (1+ n)))))

> (main-dmax '((1 2 3) (4 5 6) (7 8 9)))
((3 2 1) (4 6 5) (7 8 9))
```

Задача 4.229 *simplify-n.lisp*

Арифметическое выражение имеет форму $a + - + + - a - - - + a + + - a$, где a - это произвольное число. Упростите выражение, применяя правила замещения:

```
+ + → +
+ - → -
- + → -
- - → +
```

Решение 4.229.1

```
(defun simplify-n (w &optional b ac &aux (a (car w)))
  (if w
      (simplify-n (cdr w)
```

```

(cond ((or (and (eq b '+) (eq a '+))
            (and (eq b '-') (eq a '-))) '+)
      ((or (and (eq b '+) (eq a '-))
            (and (eq b '-') (eq a '+))) '-')
      ((numberp a) nil)
      (t a))
      (if (numberp a) (cons a (when b (cons b ac))) ac)
      (reverse ac)))

> (simplify-n '(1 + - + + - 1 - - - + 1 + + - 1))
(1 + 1 - 1 - 1)

```

Задача 4.230 *simplify-a.lisp*

Арифметическое выражение имеет форму $a + - + + - a - - - + a + + - a$ где a это произвольный атом. Упростите выражение, применяя правила замещения:

```

+ + → +
+ - → -
- + → -
- - → +

```

Решение 4.230.1

```

(defun s (w &optional ac &aux (a (car w)) (d (cdr w)))
  (cond ((null w) nil)
        ((or (eq a '+) (eq a '-)) (s d (cons a ac)))
        ((cons (if (evenp (count '- ac)) '+ '-') (cons a (s d nil))))))

(defun simplify-a (w) (cdr (s w)))

> (simplify-a '(a + - + + - a - - - + a + + - a))
(A + A - A - A)
> (simplify-a '(1 + - + + - 1 - - - + 1 + + - 1))
(1 + 1 - 1 - 1)

```

Задача 4.231 *cohere.lisp*

Задан список произвольного уровня вложенности. Превратить его в список атомов, который будет состоять только из букв, если сумма их кодов превышает сумму цифр исходного. В противном случае, список будет состоять только из цифр. В результирующем списке не должно быть повторов.

Решение 4.231.1

```

(defun cohere (w)
  (show (split (flat w))))

(defun flat (w &optional acc)
  (cond ((null w) acc)
        ((atom w) (cons w acc))

```



```

      ((flat (car w) (flat (cdr w) acc))))))

(defun split (w &aux n m)
  (dolist (a w) (if (numberp a) (push a n) (push a m)))
  (list n m))

(defun show (w)
  ((lambda (n m)
    (if (< (reduce #'+ n)
          (reduce #'+
                (mapcar #'char-code
                      (mapcar #'(lambda (a) (char a 0)) m))))
      (remove-duplicates m :test #'string-equal)
      (remove-duplicates n)))
   (car w) (cadr w)))

> (cohere '(1 ("a" ("b") 300) (1 "A")))
(300 1)

```

Решение 4.231.2

```

(defun cohere (w)
  (show (split (flat w))))

(defun flat (w)
  (mapcan #'(lambda (a) (if (listp a) (flat a) `(,a))) w))

(defun split (w &aux n m)
  (dolist (a w) (if (numberp a) (push a n) (push a m)))
  (list n m))

(defun show (w)
  ((lambda (n m)
    (if (< (reduce #'+ n)
          (reduce #'+
                (mapcar #'char-code
                      (mapcar #'(lambda (a) (char a 0)) m))))
      (remove-duplicates m :test #'string-equal)
      (remove-duplicates n)))
   (car w) (cadr w)))

> (cohere '(1 ("a" ("b") 300) (1 "A")))
(300 1)

```

Решение 4.231.3

```

(defun cohere (w)
  (show (split (flat w))))

(defun flat (w)
  (loop for a in w
        if (listp a) append (flat a)
        else collect a))

```

```

(defun split (w &aux n m)
  (dolist (a w) (if (numberp a) (push a n) (push a m)))
  (list n m))

(defun show (w)
  ((lambda (n m)
    (if (< (reduce #'+ n)
          (reduce #'+
                 (mapcar #'char-code
                        (mapcar #'(lambda (a) (char a 0)) m))))
      (remove-duplicates m :test #'string-equal)
      (remove-duplicates n)))
   (car w) (cadr w)))

> (cohere '(1 ("a" ("b") 300) (1 "A")))
(300 1)

```

Задача 4.232 *opposite-word.lisp*

Написать функцию, которая на вход, к примеру, получает список ("up" "right" "small"), а на выходе будет ("down" "left" "big").

Решение 4.232.1

```

(defun opposite-word (w)
  (mapcar
   #'(lambda (a)
      (case a
        (down 'up)
        (up 'down)
        (left 'right)
        (right 'left)
        (big 'small)
        (small 'big)
        (otherwise a)))
    w))

> (opposite-word '(up right small))
(DOWN LEFT BIG)

```

Задача 4.233 *prefix.lisp*

Арифметическое выражение с инфиксной записью и знаками + и * преобразовать в выражение с префиксной записью.

Решение 4.233.1

```

(defun pre (w &optional ac acc)
  (cond ((null w) (if (cdr ac)
                      (reverse (cons (cons '* (reverse ac)) acc))

```

```

                (reverse (nconc ac acc))))
((eq (cadr w) '+) (pre (cddr w)
                        nil
                        (cons (if ac
                                (cons '* (reverse
                                         (cons (car w) ac)))
                                (car w))
                                acc)))
  ((pre (cddr w) (cons (car w) ac) acc))))

(defun prefix (w)
  (if (member '+ w) (cons '+ (pre w)) (car (pre w))))

> (prefix '(a + b + c * d * e + f * g))
(+ A B (* C D E) (* F G))

```

Решение 4.233.2

```

(defun pre (w &optional ac acc)
  (cond ((null w) (if (cdr ac)
                      (reverse (cons (cons '* (reverse ac)) acc))
                      (reverse (nconc ac acc))))
        ((eq (cadr w) '+) (pre (cddr w)
                                nil
                                (cons (if ac
                                        (cons '* (reverse
                                                (cons (car w) ac)))
                                        (car w))
                                        acc)))
        ((pre (cddr w) (cons (car w) ac) acc))))

(defun prefix (w)
  (if (member '+ w) (cons '+ (pre w)) (cons '* (remove '* w))))

> (prefix '(a + b + c * d * e + f * g))
(+ A B (* C D E) (* F G))

```

Решение 4.233.3

```

(defun pre (w &optional ac acc &aux (a (car w)) (d (cddr w)))
  (if w
      (if (eq (cadr w) '+)
          (pre d
              nil
              (cons (if ac
                      (cons '* (reverse (cons a ac)))
                      a)
                      acc))
          (pre d (cons a ac) acc))
      (reverse (if (cdr ac)
                  (cons (cons '* (reverse ac)) acc)
                  (nconc ac acc)))))

```

```
(defun prefix (w)
  (if (member '+ w) (cons '+ (pre w)) (car (pre w))))

> (prefix '(a + b + c * d * e + f * g))
(+ A B (* C D E) (* F G))
```

Задача 4.234 rotate-core.lisp

Personally, it's made it much easier for me to customize Emacs to make it work exactly the way I want it to. It also allows me to quickly create simple functions to toggle between modes of operation. For example, different projects may require different coding styles, often variations on the default built-in styles such as `gnu` and `elmentel`; (`M-x c-set-style <enter> gnu <enter>`). With a bit of `elisp` I can bind the transition to a hot key. With a bit more effort I could even automate the switch, but generally that level of automation is too constrictive and not worth the effort. As it is out-of-the-box the default configuration of `Emscs` is OK; (try `M-x load-theme <enter> light-blue <enter>` or maybe `tango-dark` is more to your liking); but there are a lot of little things in the default setup which annoy me. With a bit of customization, all the annoyances vanish, and it's just me and the job at hand.

H. M. Wamboldt

Дан список. Переставить в обратном порядке элементы списка, расположенные между его минимальным и максимальным элементами, включая минимальный и максимальный элементы.

Решение 4.234.1

```
(defun rotate-core (w &aux (n (apply #'min w)) (x (apply #'max w))
  (a (position n w)) (b (position x w)))
  (if (< a b)
      (nconc (subseq w 0 a)
              (reverse (subseq w a (1+ b)))
              (subseq w (1+ b)))
      (nconc (subseq w 0 b)
              (reverse (subseq w b (1+ a)))
              (subseq w (1+ a)))))

> (rotate-core '(5 6 1 2 3 4 8 2 3))
(5 6 8 4 3 2 1 2 3)
> (rotate-core '(5 6 8 2 3 4 1 2 3))
```

```
(5 6 1 4 3 2 8 2 3)
> (rotate-core '(8 8 8 8 8 8 8 8))
(8 8 8 8 8 8 8 8)
```

Задача 4.235 filter.lisp

Дан список *w*. Напишите функцию, которая возвращает элементы списка *w*, удовлетворяющие предикату *p*.

Решение 4.235.1

```
(defun filter (p w)
  (cond ((null w) nil)
        ((funcall p (car w)) (cons (car w) (filter p (cdr w))))
        (t (filter p (cdr w)))))

> (filter #'oddp '(1 2 3))
(1 3)
```

Решение 4.235.2

```
(defun filter (p w)
  (when w (if (funcall p (car w))
              (cons (car w) (filter p (cdr w)))
              (filter p (cdr w)))))

> (filter #'oddp '(1 2 3))
(1 3)
```

Решение 4.235.3

```
(defun filter (p w) (loop for a in w when (funcall p a) collect a))

> (filter #'oddp '(1 2 3))
(1 3)
```

Решение 4.235.4

```
(defun filter (p w) (remove-if-not #'(lambda (a) (funcall p a)) w))

> (filter #'oddp '(1 2 3))
(1 3)
```

Задача 4.236 inner-reverse.lisp

Сделать так, чтобы в каждом атоме списка цифры шли в обратном порядке, но буквы при этом оставались не тронуты.

Решение 4.236.1

```
(defun symbols (a)
```

```

(mapcar #'read-from-string (map 'list #'string (string a))))

(defun rev-ns (w v)
  (cond ((null w) nil)
        ((numberp (car w))
         (cons (car v) (rev-ns (cdr w) (cdr v))))
        ((cons (car w) (rev-ns (cdr w) v)))))

(defun reverse-numbers (a &aux (v (symbols a)))
  (read-from-string
   (apply #'concatenate
           'string (mapcar #'write-to-string
                           (rev-ns v (reverse (remove-if-not
                                                #'numberp v)))))))

(defun inner-reverse (w) (mapcar #'reverse-numbers w))

> (inner-reverse '(a1b2c a10b23))
(A2B1C A32B01)

```

Задача 4.237 *grades.lisp*

Определить функцию, которая переводит оценки в баллы и обратно.

Решение 4.237.1

```

(defun grades (w)
  (mapcar
   #'(lambda (a)
       (case a
         (4 'A)
         (3 'B)
         (2 'C)
         (1 'D)
         (0 'F)
         (A 4)
         (B 3)
         (C 2)
         (D 1)
         (F 0)
         (otherwise 'unknown)))
   w))

> (grades '(A B 4 3))
(4 3 A B)

```

Задача 4.238 *minusp-to-zero.lisp*

Определить функцию, которая заменяет все отрицательные элементы списка значением 0.

Решение 4.238.1

```
(defun minusp-to-zero (w) (substitute-if 0 #'minusp w))

> (minusp-to-zero '(-1 0 1))
(0 0 1)
```

Задача 4.239 *len&sum.lisp*

Lisp is the red pill.

John Fraser

Файл содержит список целых чисел, разделителем которых служат пробелы. Требуется создать модуль с функцией, которая получает имя файла и возвращает список из 2-х элементов: количество чисел в файле и сумма этих чисел.

Решение 4.239.1

```
numbers.in:
11 11 11 11
12 12 12 12

(defun len&sum (&optional (path-in "d:/numbers.in"))
  (let ((u)) (with-open-file (s path-in :direction :input)
    (do ((c (read-line s nil :eof) (read-line s nil :eof)))
      ((eql c :eof))
      (setq u (nconc (list-nums c) u))))
    (values (length u) (reduce #' + u))))

(defun list-nums (s)
  (read-from-string (concatenate 'string "(" s ")))

> (len&sum "d:/numbers.in")
8
92
> (len&sum)
8
92
```

Решение 4.239.2

```
numbers.in:
11 11 11 11
12 12 12 12

(defun len&sum (path-in)
  (let ((u)) (with-open-file (s path-in :direction :input)
    (do ((c (read-line s nil :eof) (read-line s nil :eof)))
      ((eql c :eof))
      (setq u (nconc (list-nums c) u))))
```

```

(list (length u) (reduce #' + u)))

(defun list-nums (s)
  (read-from-string (concatenate 'string "(" s ")")))

> (len&sum "d:/numbers.in")
(8 92)

```

Решение 4.239.3

```

numbers.in:
11 11 11 11
12 12 12 12

(defun len&sum (path-in &aux u)
  (with-open-file (s path-in :direction :input)
    (do ((c (read-line s nil :eof) (read-line s nil :eof)))
        ((eql c :eof))
        (setq u (nconc (list-nums c) u))))
  (list (length u) (reduce #' + u)))

(defun list-nums (s)
  (read-from-string (concatenate 'string "(" s ")")))

> (len&sum "d:/numbers.in")
(8 92)

```

Решение 4.239.4

```

numbers.in:
11 11 11 11
12 12 12 12

(defun len&sum (path-in &aux u)
  (with-open-file (s path-in :direction :input)
    (do ((c (read-line s nil :eof) (read-line s nil :eof)))
        ((eql c :eof))
        (setq u (nconc (read-from-string
                        (concatenate 'string "(" c ")")) u))))
  (list (length u) (reduce #' + u)))

> (len&sum "d:/numbers.in")
(8 92)

```

Задача 4.240 co-insiders.lisp

Из числового списка построить список чисел, совпадающих со своими индексами.

Решение 4.240.1

```

(defun co-inciders (w)

```



```

(loop for a in w
      for b upfrom 0
      when (= a b) collect a))

> (co-inciders '(0 1 1 3))
(0 1 3)

```

Задача 4.241 neg.lisp

Определите функцию, которая из одноуровневого числового списка создает новый список, меняя знак у каждого атома.

Решение 4.241.1

```

(defun neg (w)
  (cond ((null w) nil)
        (t (cons (- (car w)) (neg (cdr w))))))

> (neg '(0 -1 1))
(0 1 -1)

```

Решение 4.241.2

```

(defun neg (w) (when w (cons (- (car w)) (neg (cdr w)))))

> (neg '(0 -1 1))
(0 1 -1)

```

Решение 4.241.2

```

(defun neg (w) (mapcar #'- w))

> (neg '(0 -1 1))
(0 1 -1)

```

Задача 4.242 overs-n.lisp

Определить функцию, которая возвращает числа из одноуровневого числового списка *w* больше, чем *n*.

Решение 4.242.1

```

(defun overs-n (n w)
  (cond ((null w) nil)
        ((> (car w) n) (cons (car w) (overs-n n (cdr w))))
        (t (overs-n n (cdr w)))))

> (overs-n 3 '(1 2 3 4 5 6 7))
(4 5 6 7)

```

Решение 4.242.2

```
(defun overs-n (n w)
  (when w
    (if (> (car w) n)
        (cons (car w) (overs-n n (cdr w)))
        (overs-n n (cdr w)))))
```

```
> (overs-n 3 '(1 2 3 4 5 6 7))
(4 5 6 7)
```

Решение 4.242.3

```
(defun overs-n (n w)
  (loop for a in w when (> a n) collect a))
```

```
> (overs-n 3 '(1 2 3 4 5 6 7))
(4 5 6 7)
```

Решение 4.242.4

```
(defun overs-n (n w) (remove-if-not #'(lambda (a) (> a n)) w))
```

```
> (overs-n 3 '(1 2 3 4 5 6 7))
(4 5 6 7)
```

Задача 4.243 backward.lisp

Определить функцию, которая по линейному списку вида (a b c d e ...) строит сложный список вида (a (b (c (d (e (...)))))).

Решение 4.243.1

```
(defun backward (w)
  (cond ((null (cdr w)) w)
        ((cons (car w) (list (backward (cdr w)))))))
```

```
> (backward '(a b c d e))
(A (B (C (D (E)))))
```

Решение 4.243.2

```
(defun backward (w)
  (if (cdr w) (cons (car w) (list (backward (cdr w)))) w))
```

```
> (backward '(a b c d e))
(A (B (C (D (E)))))
```

Задача 4.244 evens.lisp

Определить функцию, которая из линейного числового списка выбирает все четные числа.

Решение 4.244.1

```
(defun evens (w)
  (cond ((null w) nil)
        ((evenp (car w)) (cons (car w) (evens (cdr w))))
        (t (evens (cdr w)))))

> (evens '(0 1 2 3 4 5))
(0 2 4)
```

Решение 4.244.2

```
(defun evens (w)
  (when w (if (evenp (car w))
              (cons (car w) (evens (cdr w)))
              (evens (cdr w)))))

> (evens '(0 1 2 3 4 5))
(0 2 4)
```

Решение 4.244.3

```
(defun evens (w)
  (loop for a in w when (evenp a) collect a))

> (evens '(0 1 2 3 4 5))
(0 2 4)
```

Решение 4.244.4

```
(defun evens (w) (remove-if-not #'evenp w))

> (evens '(0 1 2 3 4 5))
(0 2 4)
```

Задача 4.245 del-thirds.lisp

Составить функцию с одним аргументом – сложным многоуровневым списком, которая превращала бы этот список таким образом, чтобы был удален каждый третий элемент во всех подсписках любого уровня вложенности.

Решение 4.245.1

```
(defun del-thirds (w)
  (cond ((atom w) w)
        (t (cons (del-thirds(car w))
                  (if (cadr w)
                      (cons (del-thirds (cadr w))
                            (del-thirds (caddr w)))
                      nil))))))
```

```
> (del-thirds '(1 2 3 (4 5 6)))
(1 2 (4 5))
```

Решение 4.245.2

```
(defun del-thirds (w)
  (cond ((atom w) w)
        (t (cons (del-thirds(car w))
                  (when (cadr w)
                    (cons (del-thirds (cadr w))
                        (del-thirds (cddddr w))))))))
```

```
> (del-thirds '(1 2 3 (4 5 6)))
(1 2 (4 5))
```

Решение 4.245.3

```
(defun del-thirds (w)
  (if (atom w)
      w
      (cons (del-thirds(car w))
            (when (cadr w)
              (cons (del-thirds (cadr w))
                  (del-thirds (cddddr w)))))))
```

```
> (del-thirds '(1 2 3 (4 5 6)))
(1 2 (4 5))
```

Задача 4.246 *make-assoc.lisp*

Дан список ((a b) (b c) (c d) ...). Превратить его в ассоциативный список. В качестве ключей взять первые элементы вложенных списков. В качестве значений - вторые.

Решение 4.246.1

```
(defun make-assoc (w)
  (loop for (a b) in w collect (cons a b)))

> (make-assoc '((a 1) (b 2) (c 3)))
((A . 1) (B . 2) (C . 3))
```

Решение 4.246.2

```
(defun make-assoc (w)
  (when w (cons
    (apply #'cons (car w))
    (make-assoc (cdr w)))))

> (make-assoc '((a 1) (b 2) (c 3)))
((A . 1) (B . 2) (C . 3))
```

Задача 4.247 num-eventh.lisp

Определить функцию, которая перед каждым четным элементом списка вставляет его порядковый номер. Пример: (a b c d ...) ==> (a 2 b c 4 d ...).

Решение 4.247.1

```
(defun num-eventh (w &optional (n 1))
  (cond ((null w) nil)
        ((evenp n) (cons n (cons (car w)
                                   (num-eventh (cdr w) (1+ n)))))
        ((cons (car w) (num-eventh (cdr w) (1+ n)))))

> (num-eventh '(a b c d e))
(A 2 B C 4 D E)
```

Решение 4.247.2

```
(defun num-eventh (w &optional (n 1))
  (when w (if (evenp n)
              (cons n (cons (car w) (num-eventh (cdr w) (1+ n))))
              (cons (car w) (num-eventh (cdr w) (1+ n)))))

> (num-eventh '(a b c d e))
(A 2 B C 4 D E)
```

Решение 4.247.3

```
(defun num-eventh (w &optional (n 1))
  (when w (if (evenp n)
              (nconc `,(n , (car w)) (num-eventh (cdr w) (1+ n)))
              (cons (car w) (num-eventh (cdr w) (1+ n)))))

> (num-eventh '(a b c d e))
(A 2 B C 4 D E)
```

Решение 4.247.4

```
(defun num-eventh (w &optional (n 1))
  (when w (nconc (if (evenp n) `,(n , (car w)) `,(, (car w)))
                 (num-eventh (cdr w) (1+ n)))))

> (num-eventh '(a b c d e))
(A 2 B C 4 D E)
```

Задача 4.248 sum-pairs.lisp

Определить функцию, которая возвращает список сумм двух соседних элементов линейного числового списка четной длины.

Решение 4.248.1

```
(defun sum-pairs (w)
  (cond ((null w) nil)
        (t (cons (+ (car w) (cadr w)) (sum-pairs (cddr w))))))
```

```
> (sum-pairs '(1 2 3 4 5 6 7 8))
(3 7 11 15)
```

Решение 4.248.2

```
(defun sum-pairs (w) (when w (cons (+ (car w) (cadr w))
                                     (sum-pairs (cddr w)))))
```

```
> (sum-pairs '(1 2 3 4 5 6 7 8))
(3 7 11 15)
```

Решение 4.248.3

```
(defun sum-pairs (w)
  (loop for (a b) on w by #'cddr collect (+ a b)))
```

```
> (sum-pairs '(1 2 3 4 5 6 7 8))
(3 7 11 15)
```

Задача 4.249 sub-lengths.lisp

Дан линейный список списков, которые имеют произвольную длину. определить функцию, который считает длину каждого из вложенных списков. Результат вернуть в виде линейного списка.

Решение 4.249.1

```
(defun sub-lengths (w)
  (cond ((null w) nil)
        (t (cons (length (car w)) (sub-lengths (cdr w))))))
```

```
> (sub-lengths '((a b) (c d e f) (g h i)))
(2 4 3)
```

Решение 4.249.2

```
(defun sub-lengths (w)
  (when w (cons (length (car w)) (sub-lengths (cdr w)))))
```

```
> (sub-lengths '((a b) (c d e f) (g h i)))
(2 4 3)
```

Решение 4.249.3

```
(defun sub-lengths (w) (mapcar #'length w))
```

```
> (sub-lengths '((a b) (c d e f) (g h i)))
```

```
(2 4 3)
```

Задача 4.250 *make-pairs.lisp*

Определить функцию, которая каждые два соседних элемента списка превращает в двухэлементный подсписок.

Решение 4.250.1

```
(defun make-pairs(w)
  (if (cdr w)
      (cons (list (car w) (cadr w))
            (make-pairs (cddr w)))
      (when w (list w))))

> (make-pairs '(a b c d e f))
((A B) (C D) (E F))
> (make-pairs '(a b c d e))
((A B) (C D) (E))
```

Решение 4.250.2

```
(defun make-pairs (w)
  (loop for (a b) on w by #'cddr
        if b collect (list a b)
        else collect (list a)))

> (make-pairs '(a b c d e f))
((A B) (C D) (E F))
> (make-pairs '(a b c d e))
((A B) (C D) (E))
```

Задача 4.251 *abstract-sum.lisp*

Дан список двухэлементных числовых списков произвольной длины. Определить функцию, которая считает разность двухэлементных списков с нечетными номерами и сумму - с нечетными.

Решение 4.251.1

```
(defun abstract-sum (w &optional (p #'-))
  (when w
    (cons (apply p (car w))
          (abstract-sum (cdr w)
                        (if (equalp p #'-)
                            #'+
                            #'-)))))

> (abstract-sum '((5 3) (8 6) (2 9)))
(2 14 -7)
```

Решение 4.251.2

```
(defun abstract-sum (w &optional f)
  (when w
    (cons (apply (if f #' + #' -) (car w))
          (abstract-sum (cdr w) (not f))))))

> (abstract-sum '((5 3) (8 6) (2 9)))
(2 14 -7)
```

Задача 4.252 *insert-nums.lisp*

C allows you to do whatever the hell you wish. Just that you'd need to implement 90% of it just so you can step through a list. In Lisp you get the same freedoms, but are given lots of powerful helpers which turn pages of C into one-liners of Lisp.

Irné Barnard

Определить функцию, которая после каждого элемента списка вставляет его порядковый номер.

Решение 4.252.1

```
(defun insert-nums (w &optional (n 1))
  (cond ((null w) nil)
        ((cons (car w) (cons n (insert-nums (cdr w) (1+ n)))))))

> (insert-nums '(a b c d))
(A 1 B 2 C 3 D 4)
```

Решение 4.252.2

```
(defun insert-nums (w &optional (n 1))
  (when w (cons (car w) (cons n (insert-nums (cdr w) (1+ n))))))

> (insert-nums '(a b c d))
(A 1 B 2 C 3 D 4)
```

Решение 4.252.3

```
(defun insert-nums (w &aux (n 0))
  (mapcan #'(lambda (a) (list a (incf n)))
          w))

> (insert-nums '(a b c d))
(A 1 B 2 C 3 D 4)
```

Решение 4.252.4


```
(defun insert-nums (w)
  (loop for a in w
        for n upfrom 1
        nconc (list a n)))
```

```
> (insert-nums '(a b c d))
(A 1 B 2 C 3 D 4)
```

Решение 4.252.5

```
(defun insert-nums (w)
  (mapcan #'list w (loop for a from 1 to (length w)
                        collect a)))
```

```
> (insert-nums '(a b c d))
(A 1 B 2 C 3 D 4)
```

Задача 4.253 *lets-marks.lisp*

Определить функцию, которая по заданному тексту, представленному списком символов, строит список пар: (<знак> <частота встречаемости в тексте>), а также выводит число точек, восклицательных знаков, вопросительных знаков и пробелов.

Решение 4.253.1

```
(defun lets-marks (w &aux (v (mapcan
                              #'(lambda (a)
                                   (concatenate 'list
                                                (string a)))
                              w)))
  (values (mapcar #'(lambda (a) (list a (count a v)))
                (remove-duplicates v))
    (+ (reduce #'+ (mapcar #'(lambda (a) (list a (count a v)))
                          '(&#\. &#\! &#\?)) :key #'cadr)
      (1- (length w)))))
```

```
> (lets-marks '(aaa. bbb. ccc!))
((#\A 3) (#\B 3) (#\ 2) (#\C 3) (#\! 1))
5
```

Решение 4.253.2

```
(defun lets-marks (w &aux (v (mapcan
                              #'(lambda (a)
                                   (concatenate 'list
                                                (string a)))
                              w)))
  (values (mapcar #'(lambda (a) (list a (count a v)))
                (remove-duplicates v))
    (+ (count-if #'(lambda (a) (or (eq a &#\.)
                                   (eq a &#\!))
                  w)
      (count-if #'(lambda (a) (or (eq a &#\.)
                                   (eq a &#\!))
                  w)))))
```

```

                                (eq a #\?)))
                                v)
      (1- (length w))))))

> (lets-marks '(aaa. bbb. ccc!))
((#\A 3) (#\B 3) (#\. 2) (#\C 3) (#\! 1))
5

```

Решение 4.253.3

```

(defun lets-marks (w &aux (v (mapcan
                              #'(lambda (a)
                                  (concatenate 'list
                                                (string a)))
                              w))))
  (values
   (loop for a in (remove-duplicates v)
         collect (list a (count a v)))
   (+ (count-if #'(lambda (a) (or (eq a #\.)
                                   (eq a #\!)
                                   (eq a #\?)))
       v)
      (1- (length w))))))

> (lets-marks '(aaa. bbb. ccc!))
((#\A 3) (#\B 3) (#\. 2) (#\C 3) (#\! 1))
5

```

Решение 4.253.3

```

(defun lets-marks (w &aux (v (mapcan
                              #'(lambda (a)
                                  (concatenate 'list
                                                (string a)))
                              w))))
  (values
   (loop for a in (remove-duplicates v)
         collect (list a (count a v)))
   (+ (count #\. v) (count #\! v) (count #\? v)
      (1- (length w))))))

> (lets-marks '(aaa. bbb. ccc!))
((#\A 3) (#\B 3) (#\. 2) (#\C 3) (#\! 1))
5

```

Задача 4.254 1-2-2-1.lisp

Дан список произвольной длины. Определить функцию, которая возвращает первый, второй, предпоследний и последний элементы этого списка.

Решение 4.254.1

```
(defun 1-2-2-1 (w)
  (cons (car w) (cons (cadr w) (last w 2)))))

> (1-2-2-1 '(1 2 3 4 5 6 7))
(1 2 6 7)
> (1-2-2-1 '(1 2))
(1 2 1 2)
> (1-2-2-1 '(1))
(1 NIL 1)
> (1-2-2-1 '())
(NIL NIL)
```

Решение 4.254.2

```
(defun 1-2-2-1 (w)
  (when w (cons (car w) (if (cadr w)
                           (cons (cadr w)
                                   (last w 2))
                           (last w 2))))))

> (1-2-2-1 '(1 2 3))
(1 2 2 3)
> (1-2-2-1 '(1 2))
(1 2 1 2)
> (1-2-2-1 '(1))
(1 1)
> (1-2-2-1 '())
NIL
```

Задача 4.255 *if-intp-abs-min-max-list-average.lisp*

Определить функцию, которая принимает пять чисел. Найти минимальное и максимальное по модулю и если они целые то сформировать из этих двух чисел список, иначе подсчитать среднее арифметическое пяти чисел.

Решение 4.255.1

```
(defun if-intp-abs-min-max-list-average
  (&rest w &aux (v (mapcar #'abs w))
                 (n (apply #'min v)) (x (apply #'max v)))
  (if (and (integerp n) (integerp x))
      (list n x)
      (/ (apply #'+ w) (length w))))

> (if-intp-abs-min-max-list-average 1 2 3 4 -5)
(1 5)
> (if-intp-abs-min-max-list-average 1 2 3 4 -5.5)
0.9
> (if-intp-abs-min-max-list-average 1 2 3 4 5.5)
3.1
```

Задача 4.256 class.lisp

Определить структуру списка студентов - имя, адрес, средний балл.

Решение 4.256.1

```
(defstruct sum name address average)

(defun res (n)
  (when (> n 0)
    (cons (make-sum :name (read) :address (read) :average (read))
          (res (1- n)))))

> (setf class (res 2))
ivanov
pushkina-10
4
petrov
gogolya-20
5
(#S(SUM :NAME IVANOV :ADDRESS PUSHKINA-10 :AVERAGE 4)
 #S(SUM :NAME PETROV :ADDRESS GOGOLYA-20 :AVERAGE 5))
```

Задача 4.257 rev-ev-lev.lisp

Создать функцию, которая делает реверс элементов во всех подсписках четных уровней, не затрагивая нечетные уровни. Принимает в качестве своего аргумента многоуровневый список, например (a b (c d (e f (g h)) i) (j (k (l m n)))) и возвращающую список (a b (i (e f (h g)) d c) ((k (n m l)) j)).

Решение 4.257.1

```
(defun rev-ev-lev (w &optional (v 1))
  (when w (mapcar
    #'(lambda (a)
      (if (listp a)
          (rev-ev-lev a (1+ v))
          a))
    (if (evenp v) (reverse w) w))))

> (rev-ev-lev '(a b (c d (e f (g h)) i) (j (k (l m n))))
(A B (I (E F (H G)) D C) ((K (N M L)) J))
```

Решение 4.257.2

```
(defun rev-ev-lev (w &optional f)
  (when w (mapcar
    #'(lambda (a)
      (if (listp a)
          (rev-ev-lev a (not f))
          a))
    w))
```

```
(if f (reverse w) w)))
```

```
> (rev-ev-lev '(a b (c d (e f (g h)) i) (j (k (l m n)))))
(A B (I (E F (H G)) D C) ((K (N M L)) J))
```

Решение 4.257.3

```
(defun rev-ev-lev (w &optional f)
  (when w (mapcar
    #'(lambda (a)
      (if (atom a) a (rev-ev-lev a (not f))))
    (if f (reverse w) w))))

> (rev-ev-lev '(a b (c d (e f (g h)) i) (j (k (l m n)))))
(A B (I (E F (H G)) D C) ((K (N M L)) J))
```

Задача 4.258 list-elm-num.lisp

Дан список. Определить функцию, возвращающую список списков, состоящих из элемента исходного списка, знака минус и порядкового номера элемента.

Решение 4.258.1

```
(defun list-elm-num (w &optional (n 1))
  (cond ((null w) nil)
        (t (cons (list (car w) '- n)
                  (list-elm-num (cdr w) (1+ n))))))

> (list-elm-num '(a b c d))
((A - 1) (B - 2) (C - 3) (D - 4))
```

Решение 4.258.2

```
(defun list-elm-num (w &optional (n 1))
  (when w (cons (list (car w) '- n)
                (list-elm-num (cdr w) (1+ n)))))

> (list-elm-num '(a b c d))
((A - 1) (B - 2) (C - 3) (D - 4))
```

Решение 4.258.3

```
(defun list-elm-num (w)
  (loop for a in w
        for b from 1
        collect (list a '- b)))

> (list-elm-num '(a b c d))
((A - 1) (B - 2) (C - 3) (D - 4))
```

Задача 4.259 *lessen.lisp*

Определить функцию, которая преобразует список целых чисел в список, каждый из элементов которого на единицу меньше соответствующего элемента.

Решение 4.259.1

```
(defun lessen (w)
  (cond ((null w) nil)
        ((cons (1- (car w)) (lessen (cdr w))))))

> (lessen '(1 2 3 4 5))
(0 1 2 3 4)
```

Решение 4.259.2

```
(defun lessen (w)
  (if w (cons (1- (car w)) (lessen (cdr w))) nil))

> (lessen '(1 2 3 4 5))
(0 1 2 3 4)
```

Решение 4.259.3

```
(defun lessen (w) (when w (cons (1- (car w)) (lessen (cdr w)))))

> (lessen '(1 2 3 4 5))
(0 1 2 3 4)
```

Решение 4.259.4

```
(defun lessen (w) (mapcar #'1- w))

> (lessen '(1 2 3 4 5))
(0 1 2 3 4)
```

Задача 4.260 *drop-second.lisp*

Из заданного списка удалить каждый второй элемент.

Решение 4.260.1

```
(defun drop-second (w)
  (cond ((null w) nil)
        (t (cons (car w) (drop-second (cddr w))))))

> (drop-second '(1 2 3 4 5))
(1 3 5)
```

Решение 4.260.2

```
(defun drop-second (w)
  (if w (cons (car w) (drop-second (cddr w))) nil))

> (drop-second '(1 2 3 4 5))
(1 3 5)
```

Решение 4.260.3

```
(defun drop-second (w)
  (when w (cons (car w) (drop-second (cddr w)))))

> (drop-second '(1 2 3 4 5))
(1 3 5)
```

Решение 4.260.4

```
(defun drop-second (w) (loop for a in w by #'cddr collect a))

> (drop-second '(1 2 3 4 5))
(1 3 5)
```

Задача 4.261 flat-pair.lisp

Дан многоуровневый список. Определить функцию, группирующую элементы попарно

Решение 4.261.1

```
(defun flat (w &optional ac)
  (cond ((null w) ac)
        ((atom w) (cons w ac))
        ((flat (car w) (flat (cdr w) ac)))))

(defun pair (w)
  (if (cdr w)
      (cons (list (car w) (cadr w))
            (pair (cddr w)))
      (when w (list w))))

(defun flat-pair (w)
  (pair (flat w)))

> (flat-pair '(a b ((c d e) f g (h i)) j k))
((A B) (C D) (E F) (G H) (I J) (K))
> (flat-pair '(a b ((c d e) f g (h i)) j k l))
((A B) (C D) (E F) (G H) (I J) (K L))
```

Решение 4.261.2

```
(defun flat (w)
  (mapcan #'(lambda (a) (if (listp a) (flat a) `(,a))) w))
```

```
(defun pair (w)
  (loop for (a b) on w by #'cddr
        if b collect (list a b)
        else collect (list a)))

(defun flat-pair (w)
  (pair (flat w)))

> (flat-pair '(a b ((c d e) f g (h i)) j k))
((A B) (C D) (E F) (G H) (I J) (K))
> (flat-pair '(a b ((c d e) f g (h i)) j k l))
((A B) (C D) (E F) (G H) (I J) (K L))
```

Решение 4.261.3

```
(defun flat (w)
  (loop for a in w
        if (listp a) append (flat a)
        else collect a))

(defun pair (w)
  (loop for (a b) on w by #'cddr
        if b collect `(,a ,b)
        else collect `( ,a)))

(defun flat-pair (w)
  (pair (flat w)))

> (flat-pair '(a b ((c d e) f g (h i)) j k))
((A B) (C D) (E F) (G H) (I J) (K))
> (flat-pair '(a b ((c d e) f g (h i)) j k l))
((A B) (C D) (E F) (G H) (I J) (K L))
```

Задача 4.262 minus-plus.lisp

Для заданного числового списка построить новый список, в котором сначала идут все отрицательные элементы, а затем все неотрицательные.

Решение 4.262.1

```
(defun minus-plus (w &optional minus plus &aux (a (car w)))
  (cond ((null w) (nconc minus plus))
        ((minusp a) (minus-plus (cdr w) (cons a minus) plus))
        ((minus-plus (cdr w) minus (cons a plus)))))

> (minus-plus '(3 -7 0 5 -4))
(-4 -7 5 0 3)
```

Решение 4.262.2

```
(defun minus-plus (w)
  (loop for i in w
```



```

    if (minusp i)
      collect i into minus
    else collect i into plus
    finally (return (nconc minus plus))))

```

```

> (minus-plus '(3 -7 0 5 -4))
(-7 -4 3 0 5)

```

Решение 4.262.3

```

(defun minus-plus (w) (sort w #'<))

```

```

> (minus-plus '(3 -7 0 5 -4))
(-7 -4 0 3 5)

```

Задача 4.263 odd*2-even*3.lisp

Для заданного числового списка построить новый список, в котором сначала идут все отрицательные элементы, а затем все неотрицательные.

Решение 4.263.1

```

(defun odd*2-even*3 (w)
  (when w (cons (* (car w) (if (oddp (car w)) 2 3))
                (odd*2-even*3 (cdr w)))))

```

```

> (odd*2-even*3 '(3 -7 0 5 -4))
(6 -14 0 10 -12)

```

Решение 4.263.2

```

(defun odd*2-even*3 (w &aux (a (car w)))
  (when w (cons (* a (if (oddp a) 2 3))
                (odd*2-even*3 (cdr w)))))

```

```

> (odd*2-even*3 '(3 -7 0 5 -4))
(6 -14 0 10 -12)

```

Решение 4.263.3

```

(defun odd*2-even*3 (w)
  (mapcar #'(lambda (a) (* a (if (oddp a) 2 3)))
          w))

```

```

> (odd*2-even*3 '(3 -7 0 5 -4))
(6 -14 0 10 -12)

```

Решение 4.263.4

```

(defun odd*2-even*3 (w)
  (loop for a in w
        collect (* a (if (oddp a) 2 3))))

```

```
> (odd*2-even*3 '(3 -7 0 5 -4))
(6 -14 0 10 -12)
```

Задача 4.264 oddth/3.lisp

Дан числовой список. Определить функцию, возвращающую список из уменьшенных в три раза элементов исходного списка, которые имеют нечетные порядковые номера.

Решение 4.264.1

```
(defun oddth/3 (w)
  (cond ((null w) nil)
        (t (cons (/ (car w) 3) (oddth/3 (cddr w))))))
```

```
> (oddth/3 '(7 1 2 13 24))
(7/3 2/3 8)
```

Решение 4.264.2

```
(defun oddth/3 (w)
  (if w (cons (/ (car w) 3) (oddth/3 (cddr w))) nil))
```

```
> (oddth/3 '(7 1 2 13 24))
(7/3 2/3 8)
```

Решение 4.264.3

```
(defun oddth/3 (w)
  (when w (cons (/ (car w) 3) (oddth/3 (cddr w)))))
```

```
> (oddth/3 '(7 1 2 13 24))
(7/3 2/3 8)
```

Решение 4.264.4

```
(defun oddth/3 (w)
  (loop for a in w by #'cddr collect (/ a 3)))
```

```
> (oddth/3 '(7 1 2 13 24))
(7/3 2/3 8)
```

Задача 4.265 twin-elm.lisp

Дан одноуровневый список. Продублировать каждый элемент списка с помощью спискообразующих функций.

Решение 4.265.1

```
(defun twin-elm (w)
```

```
(cond ((null w) nil)
      (t (cons (car w) (cons (car w) (twin-elm (cdr w)))))))

> (twin-elm '(a b c))
(A A B B C C)
```

Решение 4.265.2

```
(defun twin-elm (w)
  (when w (cons (car w) (cons (car w) (twin-elm (cdr w))))))

> (twin-elm '(a b c))
(A A B B C C)
```

Задача 4.266 *plus-exp.lisp*

Дан одноуровневый числовой список. Все положительные элементы списка возвести в квадрат.

Решение 4.266.1

```
(defun plus-exp (w)
  (cond ((null w) nil)
        ((plusp (car w)) (cons (expt (car w) 2)
                                (plus-exp (cdr w))))
        (t (plus-exp (cdr w)))))

> (plus-exp '(-2 2))
(-2 4)
```

Решение 4.266.2

```
(defun plus-exp (w)
  (when w (cons (if (plusp (car w)) (* (car w) (car w)) (car w))
                (plus-exp (cdr w)))))

> (plus-exp '(-2 2))
(-2 4)
```

Решение 4.266.3

```
(defun plus-exp (w)
  (when w (cons (* (car w) (if (plusp (car w)) (car w) 1))
                (plus-exp (cdr w)))))

> (plus-exp '(-2 2))
(-2 4)
```

Решение 4.266.4

```
(defun plus-exp (w)
  (mapcar #'(lambda (a) (if (plusp a) (* a a) a)) w))
```

```
> (plus-exp '(-2 2))
(-2 4)
```

Решение 4.266.5

```
(defun plus-exp (w)
  (loop for a in w collect (if (plusp a) (* a a) a)))
```

```
> (plus-exp '(-2 2))
(-2 4)
```

Решение 4.266.6

```
(defun plus-exp (w)
  (cond ((null w) nil)
        ((plusp (car w)) (cons (expt (car w) 2) (plus-exp (cdr w))))
        (t (cons (car w) (plus-exp (cdr w))))))
```

```
> (plus-exp '(-2 2))
(-2 4)
```

В общем виде:

Решение 4.266.7

```
(defun plus-exp (w p n)
  (cond ((null w) nil)
        ((funcall p (car w)) (cons (expt (car w) n)
                                     (plus-exp (cdr w) p n)))
        (t (cons (car w) (plus-exp (cdr w) p n)))))
```

```
> (plus-exp '(-2 2) #'plusp 2)
(-2 4)
```

Задача 4.267 snap.lisp

В каждой строке заданной матрицы вычислить сумму, количество и среднее арифметическое положительных элементов.

Решение 4.267.1

```
(defun snap (w
  &aux
    (a (remove-if-not #'plusp (car w)))
    (m (length a))
    (s (reduce #' + a)))
  (when w (cons (list s m (when (plusp m) (/ s m)))
                 (snap (cdr w)))))
```

```
> (snap '((1 2 3 -1 -2 -3) (0 0 5 6 -7 -8) (0 0 0 -7 -9)))
((6 3 2) (11 2 11/2) (0 0 NIL))
```

Решение 4.267.2

```
(defun snap (w
  &aux
    (a (remove-if-not #'plusp (car w)))
    (m (length a))
    (s (reduce #'+ a)))
  (when w (cons ` (,s ,m , (when (plusp m) (/ s m)))
    (snap (cdr w)))))

> (snap '((1 2 3 -1 -2 -3) (0 0 5 6 -7 -8) (0 0 0 -7 -9)))
((6 3 2) (11 2 11/2) (0 0 NIL))
```

Решение 4.267.3

```
(defun snap (w
  &aux
    (a (remove-if-not #'plusp (car w)))
    (m (length a))
    (s (reduce #'+ a)))
  (when w (cons ` (,s ,m , (unless (zerop m) (/ s m)))
    (snap (cdr w)))))

> (snap '((1 2 3 -1 -2 -3) (0 0 5 6 -7 -8) (0 0 0 -7 -9)))
((6 3 2) (11 2 11/2) (0 0 NIL))
```

Задача 4.268 plus-minus.lisp

Определить функцию, переставляющую все отрицательные элементы числового списка в конец списка.

Решение 4.268.1

```
(defun plus-minus (w &optional acc)
  (cond ((null w) (reverse acc))
        ((minusp (car w)) (plus-minus (cdr w) (cons (car w) acc)))
        (t (cons (car w) (plus-minus (cdr w) acc)))))

> (plus-minus '(1 -2 3 -4 -7 4 5))
(1 3 4 5 -2 -4 -7)
```

Решение 4.268.2

```
(defun plus-minus (w)
  (loop for a in w
    if (minusp a)
    collect a into minus
    else collect a into plus
    finally (return (nconc plus minus))))

> (plus-minus '(1 -2 3 -4 -7 4 5))
```

```
(1 3 4 5 -2 -4 -7)
```

Задача 4.269 all-zero.lisp

Определить функцию, переставляющую все нулевые элементы числового списка в конец списка.

Решение 4.269.1

```
(defun all-zero (w &optional acc)
  (cond ((null w) (reverse acc))
        ((zerop (car w)) (all-zero (cdr w) (cons (car w) acc)))
        (t (cons (car w) (all-zero (cdr w) acc)))))
```

```
> (all-zero '(1 0 3 0 0 4 5))
(1 3 4 5 0 0 0)
```

Решение 4.269.2

```
(defun all-zero (w)
  (loop for a in w
        if (zerop a)
        collect a into zero
        else collect a into all
        finally (return (nconc all zero))))
```

```
> (all-zero '(1 0 3 0 0 4 5))
(1 3 4 5 0 0 0)
```

Задача 4.270 let>1.lisp

Определить функцию, которая каждую букву в списке '(1 (2 a) b 14 1 b) заменит цифрой 1.

Решение 4.270.1

```
(defun let>1 (w)
  (cond ((null w) nil)
        ((listp (car w)) (cons (let>1 (car w)) (let>1 (cdr w))))
        ((symbolp (car w)) (cons 1 (let>1 (cdr w))))
        (t (cons (car w) (let>1 (cdr w)))))
```

```
> (let>1 '(1 (2 a) b 14 1 b))
(1 (2 1) 1 14 1 1)
```

Решение 4.270.2

```
(defun let>1 (w)
  (when w (cons (cond ((listp (car w)) (let>1 (car w)))
                  ((symbolp (car w)) 1)
                  (t (car w)))
```

```
(let>1 (cdr w))))
```

```
> (let>1 '(1 (2 a) b 14 1 b))
(1 (2 1) 1 14 1 1)
```

Решение 4.270.3

```
(defun let>1 (w)
  (loop for a in w
        if (listp a) collect (let>1 a)
        else if (symbolp a) collect 1
        else collect a))
```

```
> (let>1 '(1 (2 a) b 14 1 b))
(1 (2 1) 1 14 1 1)
```

Задача 4.271 dig-over.lisp

Определить функцию, которая все цифры каждого элемента списка увеличивает на 1, а 9 заменяет на 0.

Решение 4.271.1

```
(defun dig-over (w)
  (mapcar
   #'(lambda (a)
       (parse-integer
        (apply #'concatenate
               'string
               (loop for e across (write-to-string a)
                     collect (write-to-string
                               (over
                                (parse-integer
                                 (string e)))))))
       w))
```

```
(defun over (n)
  (if (= n 9) 0 (1+ n)))
```

```
> (dig-over '(123 567 896 878 102))
(234 678 907 989 213)
```

Решение 4.271.2

```
(defun dig-over (w)
  (mapcar
   #'(lambda (a)
       (parse-integer
        (apply #'concatenate
               'string
               (mapcar
                #'(lambda (e)
```

```

        (write-to-string
         (over
          (parse-integer e))))
      (map 'list
        #'string
        (write-to-string a))))))
  w))

(defun over (n)
  (case n
    (0 1)
    (1 2)
    (2 3)
    (3 4)
    (4 5)
    (5 6)
    (6 7)
    (7 8)
    (8 9)
    (9 0)))

> (dig-over '(123 567 896 878 102))
(234 678 907 989 213)

```

Решение 4.271.3

```

(defun dig-over (w)
  (mapcar
   #'(lambda (a)
        (parse-integer
         (apply #'concatenate
                  'string
                  (mapcar
                   #'over
                   (map 'list
                      #'string
                      (write-to-string a)))))))
   w))

(defun over (n)
  (cond ((equal n "0") "1")
        ((equal n "1") "2")
        ((equal n "2") "3")
        ((equal n "3") "4")
        ((equal n "4") "5")
        ((equal n "5") "6")
        ((equal n "6") "7")
        ((equal n "7") "8")
        ((equal n "8") "9")
        (t "0")))

> (dig-over '(123 567 896 878 102))
(234 678 907 989 213)

```


Решение 4.271.4

```
(defun dig-over (w)
  (mapcar
    #'(lambda (a)
      (parse-integer
        (apply #'concatenate
          'string
          (loop for e across (write-to-string a)
            collect (over (string e))))))
    w))

(defun over (n)
  (cond ((equal n "0") "1")
        ((equal n "1") "2")
        ((equal n "2") "3")
        ((equal n "3") "4")
        ((equal n "4") "5")
        ((equal n "5") "6")
        ((equal n "6") "7")
        ((equal n "7") "8")
        ((equal n "8") "9")
        (t "0")))
```

```
> (dig-over '(123 567 896 878 102))
(234 678 907 989 213)
```

Задача 4.272 sieve-of-eratosthenes.lisp

Определить функцию нахождения всех простых чисел из заданного набора методом «решето Эратосфена»: задан список n целых чисел начиная с 2. Процедура нахождения простых чисел заключается в последовательной проверке на делимость на каждое последовательно взятое оставшееся первое число всех оставшихся чисел списка и "вычёркивания" всех делящихся.

Решение 4.272.1

```
(defun sieve-of-eratosthenes (w &optional acc)
  (cond ((null w) (reverse acc))
        ((sieve-of-eratosthenes
          (remove-if #'(lambda (a) (zerop (rem a (car w)))) w)
          (cons (car w) acc)))))

> (sieve-of-eratosthenes (loop for a from 2 to 100 collect a))
(2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97)
```

Решение 4.272.2

```
(defun sieve-of-eratosthenes (w &optional acc &aux (a (car w)))
  (if w
```

```

(sieve-of-eratosthenes
  (remove-if #'(lambda (e) (zerop (rem e a))) w)
  (cons a acc))
(reverse acc)))

> (sieve-of-eratosthenes (loop for a from 2 to 100 collect a))
(2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89
97)

```

Решение 4.272.3

```

(defun sieve-of-eratosthenes (n m)
  (reverse (sieve (loop for a from n to m collect a))))

(defun sieve (w &optional acc &aux (a (car w)))
  (if w
    (sieve
      (delete-if #'(lambda (e) (zerop (rem e a))) w)
      (cons a acc))
    acc))

> (sieve-of-eratosthenes 2 100)
(2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89
97)

```

Задача 4.273 *elm-occur.lisp*

Определить функцию, преобразующую список *w* в новый список, элементы которого имеют вид: (<элемент списка *w*> <кол-во вхождений этого элемента в список *w*>).

Решение 4.273.1

```

(defun elm-occur (w)
  (loop for a in (remove-duplicates w)
        collect (list a (count a w))))

> (elm-occur '(a b a a c b))
((A 3) (C 1) (B 2))

```

Решение 4.273.2

```

(defun elm-occur (w)
  (mapcar #'(lambda (a) (list a (count a w)))
    (remove-duplicates w)))

> (elm-occur '(a b a a c b))
((A 3) (C 1) (B 2))

```

Решение 4.273.3

```

(defun elm-occur (w)

```

```
(loop for a in (remove-duplicates w)
      collect `(,a ,(count a w)))

> (elm-occur '(a b a a c b))
((A 3) (C 1) (B 2))
```

Решение 4.273.4

```
(defun elm-occur (w)
  (mapcar #'(lambda (a) `(,a ,(count a w)))
    (remove-duplicates w)))

> (elm-occur '(a b a a c b))
((A 3) (C 1) (B 2))
```

Задача 4.274 prefix.lisp

Определить функцию, вычисляющую формулу с полной скобочной структурой.

Решение 4.274.1

```
(defun prefix (w)
  (if (atom w) w `(,(cadr w) ,(prefix (car w)) ,(prefix (caddr w))))))

> (prefix '((6 + (7 - 8)) * 6))
(* (+ 6 (- 7 8)) 6)
> (eval (prefix '((6 + (7 - 8)) * 6)))
30
> (prefix '((-8 + 10) * (1 + 2)))
(* (+ -8 10) (+ 1 2))
> (eval (prefix '((-8 + 10) * (1 + 2))))
6
```

Задача 4.275 atom-level.lisp

Определить функцию, вычисляющую формулу с полной скобочной структурой.

Решение 4.275.1

```
(defun atom-level (w &optional acc (n -1))
  (cond ((null w) acc)
        ((atom w) (cons (list w n) acc))
        ((atom-level (car w)
                      (atom-level (cdr w) acc n) (1+ n)))))

> (atom-level '(a s (d f (g h (j) q) w) r))
((A 0) (S 0) (D 1) (F 1) (G 2) (H 2) (J 3) (Q 2) (W 1) (R 0))
```

Решение 4.275.2

```
(defun atom-level (w &optional acc (n -1))
```

```

(cond ((null w) acc)
      ((atom w) (cons `(.w ,n) acc))
      ((atom-level (car w)
                    (atom-level (cdr w) acc n)
                    (1+ n)))))

> (atom-level '(a s (d f (g h (j) q) w) r))
((A 0) (S 0) (D 1) (F 1) (G 2) (H 2) (J 3) (Q 2) (W 1) (R 0))

```

Задача 4.276 matrix-max-min.lisp

Числовая матрица задана списком, в котором каждый элемент представляет собой подсписок, содержащий строку матрицы. Найти максимальный и минимальный элементы матрицы.

Решение 4.276.1

```

(defun matrix-max-min (w &aux (v (flat w)))
  (values (reduce #'max v) (reduce #'min v)))

(defun flat (w &optional acc)
  (cond ((null w) acc)
        ((atom w) (cons w acc))
        ((flat (car w) (flat (cdr w) acc)))))

> (matrix-max-min '((1 2) (3 4) (5 6)))
6
1

```

Решение 4.276.2

```

(defun matrix-max-min (w &aux (v (flat w)))
  (values (reduce #'max v) (reduce #'min v)))

(defun flat (w)
  (mapcan #'(lambda (a) (if (listp a) (flat a) `(.a)))
          w))

> (matrix-max-min '((1 2) (3 4) (5 6)))
6
1

```

Решение 4.276.3

```

(defun matrix-max-min (w &aux (v (flat w)))
  (values (reduce #'max v) (reduce #'min v)))

(defun flat (w)
  (loop for a in w
        if (listp a) append (flat a)
        else collect a))

```

```
> (matrix-max-min '((1 2) (3 4) (5 6)))
6
1
```

Решение 4.276.4

```
(defun matrix-max-min (w)
  (values
    (loop for a in w maximizing (maxi a))
    (loop for a in w minimizing (mini a))))

(defun maxi (w)
  (loop for a in w maximizing a))

(defun mini (w)
  (loop for a in w minimizing a))

> (matrix-max-min '((1 2) (3 4) (5 6)))
6
1
```

Решение 4.276.5

```
(defun matrix-max-min (w)
  (values (edge w #'max) (edge w #'min)))

(defun edge (w p)
  (reduce p (mapcar #'(lambda (a) (reduce p a)) w)))

> (matrix-max-min '((1 2) (3 4) (5 6)))
6
1
```

Решение 4.276.6

```
(defun matrix-max-min (w)
  (values (edge w #'max) (edge w #'min)))

(defun edge (w p)
  (reduce p (apply #'mapcar (cons p w))))

> (matrix-max-min '((1 2) (3 4) (5 6)))
6
1
```

Решение 4.276.7

```
(defun matrix-max-min (w &aux (v (reduce #'nconc w)))
  (values (reduce #'max v) (reduce #'min v)))

> (matrix-max-min '((1 2) (3 4) (5 6)))
6
1
```

Задача 4.277 max-ab.lisp

В квадратной матрице, заданной списком строк, найти максимальный элемент и его координаты.

Решение 4.277.1

```
(defun max-ab (w
              &aux
                (a (length (car w)))
                (v (reduce #'nconc w))
                (m (reduce #'max v))
                (p (position m v)))
  `(,m , (floor p a) , (mod p a)))

> (max-ab '((1 2 3) (15 2 6) (7 8 9)))
(15 1 0)
```

Решение 4.277.2

```
(defun max-ab (w
              &aux
                (a (length (car w)))
                (v (reduce #'nconc w))
                (m (reduce #'max v))
                (p (position m v)))
  (values m (floor p a) (mod p a)))

> (max-ab '((1 2 3) (15 2 6) (7 8 9)))
15
1
0
```

Задача 4.278 max-min-elms.lisp

Дан список, состоящий из подсписков. Построить список, элементами которого являются наибольшие и наименьшие элементы подсписков данного списка, собранные в подсписки.

Решение 4.278.1

```
(defun edge (w p)
  (mapcar #'(lambda (a) (reduce p a)) w))

(defun max-min-elms (w)
  (values (edge w #'max) (edge w #'min)))

> (max-min-elms '((1 2 3) (4 5 6) (7 8 9)))
(3 6 9)
(1 4 7)
```

Задача 4.279 atom-sublist.lisp

Создать рекурсивную функцию, которая считает сколько в списке атомов и сколько подсписков.

Решение 4.279.1

```
(defun atom-sublist (w &optional (a 0) (s 0))
  (cond ((null w) (list a s))
        ((atom (car w)) (atom-sublist (cdr w) (1+ a) s))
        (t (atom-sublist (cdr w) a (1+ s)))))

> (atom-sublist '(a '(b) c '(d) e))
(3 2)
```

Задача 4.280 missed-nums.lisp

Определить функцию, возвращающую числа, пропущенные в списке идущих подряд целых чисел.

Решение 4.280.1

```
(defun missed-nums (w &optional (n (car w)))
  (cond ((null w) nil)
        ((= (car w) n) (missed-nums (cdr w) (1+ n)))
        (t (cons n (missed-nums w (1+ n)))))

> (missed-nums '(2 4 6 8 10))
(3 5 7 9)
> (missed-nums '(2 10))
(3 4 5 6 7 8 9)
```

Решение 4.280.2

```
(defun missed-nums (w &optional (n (car w)))
  (when w (if (= (car w) n)
              (missed-nums (cdr w) (1+ n))
              (cons n (missed-nums w (1+ n))))))

> (missed-nums '(2 4 6 8 10))
(3 5 7 9)
> (missed-nums '(2 10))
(3 4 5 6 7 8 9)
```

Решение 4.280.3

```
(defun missed-nums (w &optional acc (n (car w)))
  (cond ((null w) (reverse acc))
        ((= (car w) n) (missed-nums (cdr w) acc (1+ n)))
        (t (missed-nums w (cons n acc) (1+ n)))))

> (missed-nums '(2 4 6 8 10))
```

```
(3 5 7 9)
> (missed-nums '(2 10))
(3 4 5 6 7 8 9)
```

Решение 4.280.4

```
(defun missed-nums (w &aux (a (car w)) (b (car (last w))))
  (nset-difference (loop for n from a to b collect n) w))

> (missed-nums '(2 4 6 8 10))
(3 5 7 9)
> (missed-nums '(2 10))
(3 4 5 6 7 8 9)
```

Задача 4.281 -list.lisp

Определить функцию конструирования новых списков, которая возвращает список из аргументов, идущих после нее.

Решение 4.281.1

```
(defun -list (&rest w) (listt w))

(defun listt (w)
  (cond ((null w) nil)
        (t (cons (car w) (listt (cdr w))))))

> (-list 1 3/7 "foo")
(1 3/7 "foo")
```

Решение 4.281.2

```
(defun -list (&rest w) (listt w))

(defun listt (w)
  (when w (cons (car w) (listt (cdr w)))))

> (-list 1 3/7 "foo")
(1 3/7 "foo")
```

Решение 4.281.3

```
(defun -list (&rest w)
  (labels ((listt (v)
    (when v (cons (car v) (listt (cdr v))))))
    (listt w)))

> (-list 1 3/7 "foo")
(1 3/7 "foo")
```

Решение 4.281.5


```
(defun -list (&rest w)
  (when w (cons (car w) (reduce #'list (cdr w)))))

> (-list 1 3/7 "foo")
(1 3/7 "foo")
```

Решение 4.281.6

```
(defun -list (&rest w) (copy-list w))

> (-list 1 3/7 "foo")
(1 3/7 "foo")
```

Решение 4.281.7

```
(defun -list (&rest w) w)

> (-list 1 3/7 "foo")
(1 3/7 "foo")
```

Задача 4.282 *matrix-sorted.lisp*

Определить, является ли матрица отсортированной.

Решение 4.282.1

```
(defun matrix-sorted (m p q)
  (and (sorted-rows m p) (sorted-rows (transpose m) q)))

(defun sorted-rows (m p)
  (every #'identity (mapcar #'(lambda (a) (apply p a)) m)))

(defun transpose (m) (apply #'mapcar #'list m))

> (matrix-sorted '((1 2 3 3) (1 2 3 4) (1 2 3 4)) #'<= #'<=)
T
> (matrix-sorted '((1 2 3 3) (1 2 3 4) (1 2 3 3)) #'<= #'<=)
NIL
```

Решение 4.282.2

```
(defun matrix-sorted (m r c)
  (and (sorted-rows m r) (sorted-rows (transpose m) c)))

(defun sorted-rows (m r)
  (every #'identity (mapcar #'(lambda (a) (eval (cons r a))) m)))

(defun transpose (m) (apply #'mapcar #'list m))

> (matrix-sorted '((1 2 3 3) (1 2 3 4) (1 2 3 4)) '<= '<=)
T
> (matrix-sorted '((1 2 3 3) (1 2 3 4) (1 2 3 3)) '<= '<=)
```

NIL

Решение 4.282.3

```
(defun matrix-sorted (m r c)
  (and (sorted-rows m r)
       (sorted-rows (apply #'mapcar #'list m) c)))

(defun sorted-rows (m r)
  (every #'identity
         (mapcar #'(lambda (a) (eval (cons r a)))
                 m)))

> (matrix-sorted '((1 2 3 3) (1 2 3 4) (1 2 3 4)) '<= '<=)
T
> (matrix-sorted '((1 2 3 3) (1 2 3 4) (1 2 3 3)) '<= '<=)
NIL
```

Решение 4.282.4

```
(defun matrix-sorted (m r c)
  (and (sorted-rows m r)
       (sorted-rows (apply #'mapcar #'list m) c)))

(defun sorted-rows (m r)
  (every #'identity
         (mapcar #'(lambda (a) (apply r a))
                 m)))

> (matrix-sorted '((1 2 3 3) (1 2 3 4) (1 2 3 4)) #'<= #'<=)
T
> (matrix-sorted '((1 2 3 3) (1 2 3 4) (1 2 3 3)) #'<= #'<=)
NIL
```

Задача 4.283 str-frequent.lisp

Написать функцию, которая находит чаще всего встречающийся в списке атом типа "строка".

Решение 4.283.1

```
(defun str-frequent (w
  &aux
  (v (remove-if-not #'stringp w))
  (z (mapcar
      #'(lambda (a)
          `((count a w :test #'equal) ,a))
        (remove-duplicates v :test #'equal)))
  (m (reduce #'max z :key #'car)))
  (mapcar #'cadr
          (remove-if-not #'(lambda (a) (eq (car a) m)) z)))
```

```
> (str-frequent '(1 "a" "a" "a" "b"))
("a")
> (str-frequent '(1 "a" "a" "b" "b"))
("a" "b")
```

Решение 4.283.1

```
(defun str-frequent (w
                    &aux
                      (v (remove-if-not #'stringp w))
                      (z (remove-duplicates v :test #'equal))
                      (m (loop for a in z
                              maximize
                              (count a v :test #'equal))))
  (remove-if
   #'(lambda (a) (< (count a v :test #'equal) m)) z))

> (str-frequent '(1 "a" "a" "a" "b"))
("a")
> (str-frequent '(1 "a" "a" "b" "b"))
("a" "b")
```

Задача 4.284 sublists.lisp

Определить функцию, которая возвращает список подсписков данного списка.

Решение 4.284.1

```
(defun sublists (w &optional acc &aux (a (car w)) (d (cdr w)))
  (cond ((null w) acc)
        ((atom a) (sublists d acc))
        (t (sublists a (cons a (sublists d acc))))))

> (sublists '((a 3 4) (b 2 (c 3 4) 5) (d 7) ((e 7))))
((A 3 4) (C 3 4) (B 2 (C 3 4) 5) (D 7) (E 7) ((E 7)))
> (sublists '((a 3 4)))
((A 3 4))
> (sublists '(a 3 4))
NIL
```

Задача 4.285 max-sum-sublists.lisp

Определить функцию, которая возвращает подписки верхнего уровня, сумма числовых элементов которых максимальна.

Решение 4.285.1

```
(defun sum-nums-sublists (w)
  (mapcar #'(lambda (a)
              (list (reduce #'+ (remove-if-not #'numberp a)) a)) w))
```

```
(defun max-sum-sublists (w
                        &aux
                        (v (sum-nums-sublists w))
                        (m (reduce #'max v :key #'car)))
  (mapcar #'cadr (remove-if-not #'(lambda (a) (= (car a) m)) v)))

> (max-sum-sublists '((a 3 4) (b 2 (c 3 4) 5) (d 7) ((e 7))))
((A 3 4) (B 2 (C 3 4) 5) (D 7))
> (max-sum-sublists '((a 3 4) (b 2 (c 3 4) 5) (d 8) ((e 8))))
((D 8))
```

Задача 4.286 max-sum-all-sublists.lisp

Определить функцию, которая возвращает со всех уровней вложенности подписки, сумма числовых элементов которых максимальна.

Решение 4.286.1

```
(defun all-sublists (w &optional acc &aux (a (car w)) (d (cdr w)))
  (cond ((null w) acc)
        ((atom a) (all-sublists d acc))
        (t (all-sublists a (cons a (all-sublists d acc))))))

(defun sum-nums-sublists (w)
  (mapcar #'(lambda (a)
              (list (reduce #'+ (remove-if-not #'numberp a)) a)) w))

(defun max-sum-all-sublists (w
                              &aux
                              (v (sum-nums-sublists (all-sublists w)))
                              (m (reduce #'max v :key #'car)))
  (mapcar #'cadr (remove-if-not #'(lambda (a) (= (car a) m)) v)))

> (max-sum-all-sublists '((a 3 4) (b 2 (c 3 4) 5) (d 7) ((e 7))))
((A 3 4) (C 3 4) (B 2 (C 3 4) 5) (D 7) (E 7))
> (max-sum-all-sublists '((a 3 4) (b 2 (c 3 4) 5) (d 7) ((e 8))))
((E 8))
```

Задача 4.287 max-t.lisp

Определить номера заявок, на обслуживание которых потребуется самое большое время.

Решение 4.287.1

```
(defun max-t (w &aux (m (reduce #'max w :key #'cadr)))
  (mapcar #'car (remove-if-not #'(lambda (a) (= (cadr a) m)) w)))

> (max-t '((a 1) (b 2) (c 2)))
```

(B C)

Решение 4.287.2

```
(defun max-t (w &aux (m (loop for a in w maximize (cadr a))))
  (loop for a in w when (= (cadr a) m) collect (car a)))

> (max-t '((a 1) (b 2) (c 2)))
(B C)
```

Задача 4.288 min-t.lisp

Определить номера заявок, на обслуживание которых потребуется самое малое время.

Решение 4.288.1

```
(defun min-t (w &aux (m (reduce #'min w :key #'cadr)))
  (mapcar #'car (remove-if-not #'(lambda (a) (= (cadr a) m)) w)))

> (min-t '((a 1) (b 2) (c 2)))
(A)
```

Решение 4.288.2

```
(defun min-t (w &aux (m (loop for a in w minimize (cadr a))))
  (loop for a in w when (= (cadr a) m) collect (car a)))

> (min-t '((a 1) (b 2) (c 2)))
(A)
```

Задача 4.289 listt.lisp

Определить функцию, которая по линейному списку вида (1 2 3 4 5) строит сложный список вида ((1) (2) (3) (4) (5)).

Решение 4.289.1

```
(defun listt (w)
  (cond ((null w) nil)
        (t (cons (list (car w)) (listt (cdr w))))))

> (listt '(1 2 3 4 5))
((1) (2) (3) (4) (5))
```

Решение 4.289.2

```
(defun listt (w) (when w (cons (list (car w)) (listt (cdr w)))))

> (listt '(1 2 3 4 5))
((1) (2) (3) (4) (5))
```

Решение 4.289.3

```
(defun listt (w)
  (when w (cons `,(car w) (listt (cdr w)))))
```

```
> (listt '(1 2 3 4 5))
((1) (2) (3) (4) (5))
```

Решение 4.289.4

```
(defun listt (w &optional acc)
  (cond ((null w) (reverse acc))
        (t (listt (cdr w) (cons (list (car w)) acc))))))
```

```
> (listt '(1 2 3 4 5))
((1) (2) (3) (4) (5))
```

Решение 4.289.5

```
(defun listt (w &optional acc)
  (cond ((null w) (reverse acc))
        (t (listt (cdr w) (push `,(car w) acc))))))
```

```
> (listt '(1 2 3 4 5))
((1) (2) (3) (4) (5))
```

Решение 4.289.6

```
(defun listt (w &optional acc)
  (if w (listt (cdr w) (push `,(car w) acc)) (reverse acc)))
```

```
> (listt '(1 2 3 4 5))
((1) (2) (3) (4) (5))
```

Решение 4.289.7

```
(defun listt (w &optional acc)
  (if w (listt (cdr w) (cons `,(car w) acc)) (reverse acc)))
```

```
> (listt '(1 2 3 4 5))
((1) (2) (3) (4) (5))
```

Решение 4.289.8

```
(defun listt (w)
  (loop for a in w collect (list a)))
```

```
> (listt '(1 2 3 4 5))
((1) (2) (3) (4) (5))
```

Решение 4.289.10

```
(defun listt (w)
  (mapcar #'list w))

> (listt '(1 2 3 4 5))
((1) (2) (3) (4) (5))
```

Задача 4.290 atom-counter.lisp

Определить функцию, возвращающую для заданного списка *w* список вида ((atom1 <число вхождений в список *w*>) (atom2 <число вхождений в список *w*>)...).

Решение 4.290.1

```
(defun atom-counter (w &aux (v (remove-duplicates w)))
  (mapcar #'(lambda (a) (list a (count a w))) v))

> (atom-counter '(a b b c c c))
((A 1) (B 2) (C 3))
```

Решение 4.290.2

```
(defun atom-counter (w)
  (mapcar #'(lambda (a) (list a (count a w)))
    (remove-duplicates w)))

> (atom-counter '(a b b c c c))
((A 1) (B 2) (C 3))
```

Задача 4.291 flatten-and-count.lisp

Определить функцию, возвращающую для заданного многоуровневого списка *w* список вида ((atom1 <число вхождений в список *w*>) (atom2 <число вхождений в список *w*>)...).

Решение 4.291.1

```
(defun flatten-and-count (w
  &aux
    (z (flat w))
    (v (remove-duplicates z)))
  (mapcar #'(lambda (a) (list a (count a z))) v))

(defun flat (w &optional acc)
  (cond ((null w) acc)
        ((atom w) (cons w acc))
        ((flat (car w) (flat (cdr w) acc)))))

> (flatten-and-count '(a (b (b) c) (c) c))
((A 1) (B 2) (C 3))
```

Решение 4.291.2

```
(defun flatten-and-count (w
                          &aux
                          (z (flat w))
                          (v (remove-duplicates z)))
  (mapcar #'(lambda (a) (list a (count a z))) v))

(defun flat (w)
  (mapcan #'(lambda (a) (if (listp a) (flat a) `(",a))) w))

> (flatten-and-count '(a (b (b) c) (c) c))
((A 1) (B 2) (C 3))
```

Решение 4.291.3

```
(defun flatten-and-count (w
                          &aux
                          (z (flat w))
                          (v (remove-duplicates z)))
  (mapcar #'(lambda (a) (list a (count a z))) v))

(defun flat (w)
  (loop for a in w
        if (listp a) append (flat a)
        else collect a))

> (flatten-and-count '(a (b (b) c) (c) c))
((A 1) (B 2) (C 3))
```

Задача 4.292 *get-plusp.lisp*

Определить функцию, возвращающую список положительных элементов исходного списка.

Решение 4.292.1

```
(defun get-plusp (w &optional acc &aux (a (car w)) (d (cdr w)))
  (cond ((null w) (nreverse acc))
        ((listp a) (get-plusp d (cons (get-plusp a) acc)))
        ((plusp a) (get-plusp d (cons a acc)))
        ((get-plusp d acc))))

> (get-plusp '((1 6 (-5 2) -3 7)))
((1 6 (2) 7))
```

Решение 4.292.2

```
(defun get-plusp (w &optional acc &aux (a (car w)) (d (cdr w)))
  (if w
      (get-plusp d (cond ((listp a) (cons (get-plusp a) acc))
                        ((plusp a) (cons a acc))
                        (t acc)))
      acc))
```



```

                (t acc)))
      (nreverse acc)))

> (get-plusp '((1 6 (-5 2) -3 7)))
((1 6 (2) 7))

```

Решение 4.292.3

```

(defun get-plusp (w)
  (loop for a in w
        if (listp a)
        collect (get-plusp a)
        else when (plusp a) collect a))

> (get-plusp '((1 6 (-5 2) -3 7)))
((1 6 (2) 7))

```

Задача 4.293 *sort-atoms-by-freq.lisp*

Дан список атомов. Написать программу, возвращающую список вида: первый элемент — атом исходного списка, появляющийся в списке один раз, второй элемент — атом, появляющийся два раза и т. д. Если есть несколько атомов, появляющихся одинаковое количество раз, то их объединить в список.

Решение 4.293.1

```

(defun sort-atoms-by-freq (w)
  (sort-atoms (sort (atom-freq w) #'< :key #'cadr)))

(defun atom-freq (w &aux (v (remove-duplicates w)))
  (mapcar #'(lambda (a) (list a (count a w))) v))

(defun sort-atoms (w &optional ac acc &aux (a (car w)) (d (cdr w)))
  (cond ((null w) (nreverse (cons (caar ac) acc)))
        ((or (null ac) (eq (cadar ac) (cadr a)))
         (sort-atoms d (cons a ac) acc))
        ((sort-atoms d (list a) (cons (if (cdr ac)
                                             (nreverse (mapcar #'car ac))
                                             (caar ac))
                                         acc)))))

> (sort-atoms-by-freq '(a a s s s d d f f f f))
((A D) S F)

```

Задача 4.294 *symbols-numbers.lisp*

Определить функцию, формирующую два списка. в первом расположены символы, а во втором численные элементы входного списка.

Решение 4.294.1

```
(defun symbols-numbers (w &optional s n &aux (a (car w)) (d (cdr w)))
  (cond ((null w) (list (reverse s) (reverse n)))
        ((symbolp a) (symbols-numbers d (cons a s) n))
        ((numberp a) (symbols-numbers d s (cons a n)))
        (t (symbols-numbers d s n))))
```

```
> (symbols-numbers '(a 1 b 2 c 3))
((A B C) (1 2 3))
```

Решение 4.294.2

```
(defun symbols-numbers (w &optional s n &aux (a (car w)) (d (cdr w)))
  (cond ((null w) (values (reverse s) (reverse n)))
        ((symbolp a) (symbols-numbers d (cons a s) n))
        ((numberp a) (symbols-numbers d s (cons a n)))
        (t (symbols-numbers d s n))))
```

```
> (symbols-numbers '(a 1 b 2 c 3))
(A B C)
(1 2 3)
```

Решение 4.294.3

```
(defun symbols-numbers (w)
  (loop for a in w
        if (symbolp a) collect a into s
        else when (numberp a) collect a into n
        finally (return (list s n))))
```

```
> (symbols-numbers '(a 1 b 2 c 3))
((A B C) (1 2 3))
```

Решение 4.294.4

```
(defun symbols-numbers (w)
  (loop for a in w
        if (symbolp a) collect a into s
        else when (numberp a) collect a into n
        finally (return (values s n))))
```

```
> (symbols-numbers '(a 1 b 2 c 3))
(A B C)
(1 2 3)
```

Решение 4.294.5

```
(defun symbols-numbers (w)
  (list (remove-if-not #'symbolp w) (remove-if-not #'numberp w)))
```

```
> (symbols-numbers '(a 1 b 2 c 3))
((A B C) (1 2 3))
```

Решение 4.294.6

```
(defun symbols-numbers (w)
  (values (remove-if-not #'symbolp w) (remove-if-not #'numberp w)))

> (symbols-numbers '(a 1 b 2 c 3))
(A B C)
(1 2 3)
```

Задача 4.295 *level-number-of-atoms.lisp*

Дан многоуровневый список. Определить функцию, формирующую список следующего вида: ((lev0 cnt0) (lev1 cnt1 ... (levn cntn))), где *levi* и *cnti* - уровень вложенности и количество атомов на нем.

Решение 4.295.1

```
(defun dive-count (n w &aux (a (car w)) (d (cdr w)))
  (cond ((null w) 0)
        ((= n 0) (count-if #'atom w))
        ((atom a) (dive-count n d))
        ((+ (dive-count (1- n) a) (dive-count n d)))))

(defun max-depth (w)
  (if (atom w)
      0
      (max (1+ (max-depth (car w)))
            (max-depth (cdr w)))))

(defun level-number-of-atoms (w &aux (n (1- (max-depth w))))
  (loop for a from 0 to n
        collect (list a (dive-count a w))))

> (level-number-of-atoms '(1 (a b) 1 (c) 1 ((3)) 1 ((3 3))))
((0 4) (1 3) (2 3))
```

Задача 4.296 *deploy-atoms.lisp*

Дан список, содержащий атомы и списки. Каждый элемент-список содержит символичный атом и номера позиций на которых должен размещаться этот атом в списке-результате. Элементы-атомы размещаются в результате на позициях, не занятых при размещении атомов из элементов-списков. Если после размещения всех данных элементов в списке - результате остаются незанятые промежуточные позиции, то на этих позициях должен стоять nil.

Решение 4.296.1

```
(defun deploy-atoms (w &aux (v (remove-if #'atom w))
  (z (remove-if-not #'atom w)))
  (deploy (sort (mapcan #'atom-num v) #'< :key #'cadr) z))
```

```

(defun deploy (w v &optional (n 1)
  &aux (a (car w)) (d (cdr w)))
  (cond ((null w) nil)
        ((= (cadr a) n) (cons (car a) (deploy d v (1+ n))))
        (t (cons (car v) (deploy w (cdr v) (1+ n))))))

(defun atom-num (w &aux (a (car w)) (d (cdr w)))
  (mapcar #'(lambda (e) `(,a ,e)) d))

> (deploy-atoms '((a 1 3 6) c (d 2 5)))
(A D A C D A)
> (deploy-atoms '((a 1 3 6) (d 2 5)))
(A D A NIL D A)

```

Решение 4.296.2

```

(defun deploy-atoms (w)
  (deploy (sort (mapcan #'atom-num
    (remove-if #'atom w))
    #'< :key #'cadr)
    (remove-if-not #'atom w)))

(defun deploy (w v &optional (n 1) &aux (a (car w)))
  (when w (if (= (cadr a) n)
    (cons (car a) (deploy (cdr w) v (1+ n)))
    (cons (car v) (deploy w (cdr v) (1+ n))))))

(defun atom-num (w &aux (a (car w)))
  (mapcar #'(lambda (e) `(,a ,e)) (cdr w)))

> (deploy-atoms '((a 1 3 6) c (d 2 5)))
(A D A C D A)
> (deploy-atoms '((a 1 3 6) (d 2 5)))
(A D A NIL D A)

```

Задача 4.297 *sum-len.lisp*

Дан список произвольной структуры, в который входят как числа, так и символьные атомы. Написать функцию, которая возвращает список из двух чисел, первое из них равно сумме чисел исходного списка, а второе их количеству.

Решение 4.297.1

```

(defun sum-len (w &aux (v (remove-if-not #'numberp (flat w))))
  (list (reduce #'+ v) (length v)))

(defun flat (w &optional acc)
  (cond ((null w) acc)
        ((atom w) (cons w acc))
        ((flat (car w) (flat (cdr w) acc)))))

```

```
> (sum-len '((1 a) (2 (3) b (4))))
(10 4)
```

Решение 4.297.2

```
(defun sum-len (w &aux (v (remove-if-not #'numberp (flat w))))
  (list (reduce #'+ v) (length v)))

(defun flat (w)
  (mapcan #'(lambda (a) (if (listp a) (flat a) `(,a))) w))

> (sum-len '((1 a) (2 (3) b (4))))
(10 4)
```

Решение 4.297.3

```
(defun sum-len (w &aux (v (remove-if-not #'numberp (flat w))))
  (values (reduce #'+ v) (length v)))

(defun flat (w)
  (loop for a in w
        if (listp a) append (flat a)
        else collect a))

> (sum-len '((1 a) (2 (3) b (4))))
10
4
```

Задача 4.298 drop-num.lisp

Определить функцию, удаляющую все числовые атомы из списка и всех его подсписков.

Решение 4.298.1

```
(defun drop-num (w &aux (a (car w)) (d (cdr w)))
  (cond ((null w) nil)
        ((listp a) (cons (drop-num (car w)) (drop-num d)))
        ((numberp a) (drop-num d))
        ((cons a (drop-num d)))))

> (drop-num '(a 1 (b 2) c))
(A (B) C)
> (drop-num '(a 1 (b 2) (c (3))))
(A (B) (C NIL))
```

Решение 4.298.2

```
(defun drop-num (w &aux (a (car w)) (d (cdr w)))
  (when w (if (atom a)
              (if (numberp a)
```

```

      (drop-num d)
      (cons a (drop-num d)))
    (cons (drop-num (car w)) (drop-num d))))

```

```

> (drop-num '(a 1 (b 2) c))
(A (B) C)
> (drop-num '(a 1 (b 2) (c (3))))
(A (B) (C NIL))

```

Решение 4.298.3

```

(defun drop-num (w)
  (loop for a in w
        if (listp a) collect (drop-num a)
        else unless (numberp a) collect a))

```

```

> (drop-num '(a 1 (b 2) c))
(A (B) C)
> (drop-num '(a 1 (b 2) (c (3))))
(A (B) (C NIL))

```

Задача 4.299 *reconquista.lisp*

Определить функцию, которая по списку `'(a b c d e f)` строит сложный список `(a (b (c (d (e (f)))))`).

Решение 4.299.1

```

(defun reconquista (w)
  (if (null (cdr w))
      (list (car w))
      (list (car w) (reconquista (cdr w)))))

```

```

> (reconquista '(a b c d e f))
(A (B (C (D (E (F)))))

```

Решение 4.299.2

```

(defun reconquista (w)
  (if (cdr w) `((, (car w) ,(reconquista (cdr w))) `((, (car w))))
      (list (car w)))
> (reconquista '(a b c d e f))
(A (B (C (D (E (F)))))

```

Задача 4.300 *separate.lisp*

Дан одноуровневый список, состоящий из чисел. Определить функцию, которая возвращает список отрицательных, нулей и положительных чисел.

Решение 4.300.1

```
(defun separate (w)
  (loop for a in w
        if (plusp a) collect a into p
        else if (minusp a) collect a into m
        else collect a into z
        finally (return (values m z p))))

> (separate '(-2 -1 0 1 2))
(-2 -1)
(0)
(1 2)
```

Задача 4.301 skim.lisp

Определите функцию, которая в одноуровневом списке чисел *s* переставляет элементы, удовлетворяющие некоторому предикату, в начало списка.

Решение 4.301.1

```
(defun skim (p w &optional good bad &aux (a (car w)) (d (cdr w)))
  (cond ((null w) (append (reverse good) (reverse bad)))
        ((funcall p a) (skim p d (cons a good) bad))
        ((skim p d good (cons a bad)))))
```

```
> (skim #'oddp '(1 2 3 4 5 6 7))
(1 3 5 7 2 4 6)
```

Решение 4.301.2

```
(defun skim (p w &optional good bad &aux (a (car w)) (d (cdr w)))
  (if w
      (if (funcall p a)
          (skim p d (cons a good) bad)
          (skim p d good (cons a bad)))
      (nconc (nreverse good) (nreverse bad))))
```

```
> (skim #'oddp '(1 2 3 4 5 6 7))
(1 3 5 7 2 4 6)
```

Решение 4.301.3

```
(defun skim (p w)
  (loop for a in w
        if (funcall p a) collect a into good
        else collect a into bad
        finally (return (nconc good bad))))
```

```
> (skim #'oddp '(1 2 3 4 5 6 7))
(1 3 5 7 2 4 6)
```

Решение 4.301.4

```
(defun skim (p w)
  (nconc (remove-if-not p w) (remove-if p w)))

> (skim #'oddp '(1 2 3 4 5 6 7))
(1 3 5 7 2 4 6)
```

Задача 4.302 deep-sort.lisp

Определите функцию, которая сортирует список чисел по возрастанию. Список может быть многоуровневым. Тогда вес (значение) любого подсписка для сортировки определяется суммой его элементов. Все подписки также должны быть отсортированы.

Решение 4.302.1

```
(defun deep-sort (w &aux (v (deep w)))
  (mapcar #'(lambda (a) (if (atom a) a (deep-sort a))) v))

(defun deep (w)
  (sort w #'<:key
        #'(lambda (a) (if (listp a) (reduce #'+ (flat a)) a))))

(defun flat (w &optional acc)
  (cond ((null w) acc)
        ((atom w) (cons w acc))
        ((flat (car w) (flat (cdr w) acc)))))

> (deep-sort '(9 2 (2 1) 5 (3 6 2) 12 (6 (3 1))))
(2 (1 2) 5 9 ((1 3) 6) (2 3 6) 12)
```

Решение 4.302.2

```
(defun deep-sort (w &aux (v (deep w)))
  (loop for a in v collect (if (atom a) a (deep-sort a))))

(defun deep (w)
  (sort w #'<:key #'
        (lambda (a) (if (atom a) a (reduce #'+ (flat a))))))

(defun flat (w)
  (mapcan #'(lambda (a) (if (listp a) (flat a) `(,a))) w))

> (deep-sort '(9 2 (2 1) 5 (3 6 2) 12 (6 (3 1))))
(2 (1 2) 5 9 ((1 3) 6) (2 3 6) 12)
```

Решение 4.302.3

```
(defun deep-sort (w &aux (v (deep w)))
  (loop for a in v collect (if (atom a) a (deep-sort a))))

(defun key-lambda (a)
  (if (atom a) a (reduce #'+ (flat a))))
```



```
(defun deep (w)
  (sort w #'< :key #'key-lambda))

(defun flat (w)
  (loop for a in w
        if (listp a) append (flat a)
        else collect a))

> (deep-sort '(9 2 (2 1) 5 (3 6 2) 12 (6 (3 1))))
(2 (1 2) 5 9 ((1 3) 6) (2 3 6) 12)
```

Решение 4.302.4

```
(defun deep-sort (w &aux (v (deep w)))
  (loop for a in v collect (if (atom a) a (deep-sort a))))

(defun key-lambda (a) (if (atom a) a (sum a)))

(defun sum (w)
  (cond ((null w) 0)
        ((atom w) w)
        ((+ (sum (car w)) (sum (cdr w))))))

(defun deep (w)
  (sort w #'< :key #'key-lambda))

> (deep-sort '(9 2 (2 1) 5 (3 6 2) 12 (6 (3 1))))
(2 (1 2) 5 9 ((1 3) 6) (2 3 6) 12)
```

Решение 4.302.5

```
(defun deep-sort (w &aux (v (deep w)))
  (loop for a in v collect (if (atom a) a (deep-sort a))))

(defun deep (w)
  (sort w #'< :key #'(lambda (a) (if (atom a) a (sum a)))))

(defun sum (w)
  (if w (if (atom w) w (+ (sum (car w)) (sum (cdr w)))) 0))

> (deep-sort '(9 2 (2 1) 5 (3 6 2) 12 (6 (3 1))))
(2 (1 2) 5 9 ((1 3) 6) (2 3 6) 12)
```

Решение 4.302.6

```
(defun deep-sort (w)
  (loop for a in (deep w) collect (if (atom a) a (deep-sort a))))

(defun deep (w)
  (sort w #'< :key #'(lambda (a) (if (atom a) a (sum a)))))

(defun sum (w)
```

```
(if w (if (atom w) w (+ (sum (car w)) (sum (cdr w)))) 0))

> (deep-sort '(9 2 (2 1) 5 (3 6 2) 12 (6 (3 1))))
(2 (1 2) 5 9 ((1 3) 6) (2 3 6) 12)
```

Задача 4.303 sum-elt.lisp

Дан список двухэлементных числовых списков произвольной длины. Определить функцию, возвращающую суммы элементов в этих списках.

Решение 4.303.1

```
(defun sum-elm (w)
  (mapcar #'(lambda (a) (eval (cons '+ a))) w))

> (sum-elm '((1 1) (1 1)))
(2 2)
```

Решение 4.303.2

```
(defun sum-elm (w) (mapcar #'(lambda (a) (reduce #'+ a)) w))

> (sum-elm '((1 1) (1 1)))
(2 2)
```

Решение 4.303.3

```
(defun sum-elm (w) (loop for a in w collect (reduce #'+ a)))

> (sum-elm '((1 1) (1 1)))
(2 2)
```

Решение 4.303.4

```
(defun sum-elm (w)
  (cond ((null w) nil)
        (t (cons (reduce #'+ (car w))
                  (sum-elm (cdr w))))))

> (sum-elm '((1 1) (1 1)))
(2 2)
```

Решение 4.303.5

```
(defun sum-elm (w)
  (when w (cons (reduce #'+ (car w)) (sum-elm (cdr w)))))

> (sum-elm '((1 1) (1 1)))
(2 2)
```

Решение 4.303.6

```
(defun sum-elm (w)
  (mapcar #'(lambda (a) (+ (car a) (cadr a))) w))

> (sum-elm '((1 1) (1 1)))
(2 2)
```

Задача 4.304 even/n-odd.lisp

Вычислить среднее арифметическое четных и сумму нечетных элементов одноуровневого списка.

Решение 4.304.1

```
(defun even/n-odd (w &optional (even 0) (n 0) (odd 0))
  (cond ((null w) (list (unless (zerop n) (/ even n)) odd))
        ((evenp (car w)) (even/n-odd (cdr w)
                                       (+ even (car w))
                                       (1+ n)
                                       odd))
        ((even/n-odd (cdr w)
                      even
                      n
                      (+ odd (car w)))))

> (even/n-odd '(1 2 3 4 5 6 7))
(4 16)
> (even/n-odd '(1 3 5 7))
(NIL 16)
```

Решение 4.304.2

```
(defun even/n-odd (w)
  (loop for a in w
        if (evenp a) collect a into even
        else sum a into odd
        finally (return (values
                          (when even
                            (/ (reduce #'+ even)
                                (length even)))
                          odd))))

> (even/n-odd '(1 2 3 4 5 6 7))
(4 16)
> (even/n-odd '(1 3 5 7))
(NIL 16)
```

Задача 4.305 sum-list.lisp

Определить функцию, которая из исходного списка формирует список-результат: первый элемент - сумма всех элементов, второй - сумма элементов хвоста и т. д.

Решение 4.305.1

```
(defun sum-list (w)
  (when w (cons (reduce #' + w) (sum-list (cdr w)))))

> (sum-list '(1 2 3 4 5 6 7 8 9 10))
(55 54 52 49 45 40 34 27 19 10)
```

Решение 4.305.2

```
(defun sum-list (w) (maplist #'(lambda (d) (reduce #' + d)) w))

> (sum-list '(1 2 3 4 5 6 7 8 9 10))
(55 54 52 49 45 40 34 27 19 10)
```

Решение 4.305.3

```
(defun sum-list (w) (loop for d on w collect (reduce #' + d)))

> (sum-list '(1 2 3 4 5 6 7 8 9 10))
(55 54 52 49 45 40 34 27 19 10)
```

Задача 4.306 >postfix.lisp

Определить функцию, которая из префиксной формы создает постфиксную: (аргумент1 аргумент2 ... аргументN функция).

Решение 4.306.1

```
(defun shift> (w) (when w (nconc (cdr w) (list (car w)))))

(defun >postfix (w)
  (loop for a in w collect (if (atom a) a (shift> a)) into b
    finally (return (shift> b))))

> (>postfix '(+))
(+)
> (>postfix '(+ 1 2 3))
(1 2 3 +)
> (>postfix '(+ 4 (* 5 d) 7))
(4 (5 D *) 7 +)
```

Задача 4.307 sine-adjacent.lisp

Определить функцию, которая удаляет из списка идущие подряд элементы.

Решение 4.307.1

```
(defun sine-adjacent (w &optional b)
  (cond ((null w) nil)
```

```
((eq b (car w)) (sine-adjacent (cdr w) b))
((cons (car w) (sine-adjacent (cdr w) (car w))))))
```

```
> (sine-adjacent '(1 2 3 3 4 5 4 4 3 1 1))
(1 2 3 4 5 4 3 1)
```

Решение 4.307.2

```
(defun sine-adjacent (w &optional b)
  (when w (if (eq b (car w))
              (sine-adjacent (cdr w) b)
              (cons (car w)
                    (sine-adjacent (cdr w) (car w))))))
```

```
> (sine-adjacent '(1 2 3 3 4 5 4 4 3 1 1))
(1 2 3 4 5 4 3 1)
```

Решение 4.307.3

```
(defun sine-adjacent (w &optional b ac)
  (cond ((null w) (reverse ac))
        ((eq b (car w)) (sine-adjacent (cdr w) b ac))
        ((sine-adjacent (cdr w) (car w) (cons (car w) ac)))))
```

```
> (sine-adjacent '(1 2 3 3 4 5 4 4 3 1 1))
(1 2 3 4 5 4 3 1)
```

Решение 4.307.4

```
(defun sine-adjacent (w &optional ac)
  (cond ((null w) (reverse ac))
        ((eq (car w) (cadr w)) (sine-adjacent (cdr w) ac))
        ((sine-adjacent (cdr w) (cons (car w) ac)))))
```

```
> (sine-adjacent '(1 2 3 3 4 5 4 4 3 1 1))
(1 2 3 4 5 4 3 1)
```

Решение 4.307.5

```
(defun sine-adjacent (w)
  (when w (if (eq (car w) (cadr w))
              (sine-adjacent (cdr w))
              (cons (car w)
                    (sine-adjacent (cdr w))))))
```

```
> (sine-adjacent '(1 2 3 3 4 5 4 4 3 1 1))
(1 2 3 4 5 4 3 1)
```

Решение 4.307.6

```
(defun sine-adjacent (w)
  (cons (car w)
        (loop for a in w
```

```

        for b in (cdr w)
        unless (eq a b) collect b)))

> (sine-adjacent '(1 2 3 3 4 5 4 4 3 1 1))
(1 2 3 4 5 4 3 1)

```

Задача 4.308 *sort-flatten.lisp*

Определить функцию, которая выстраивает атомы в многоуровневом числовом списке *w* в один уровень по возрастанию.

Решение 4.308.1

```

(defun flatten (w)
  (mapcan #'(lambda (a) (if (atom a) `(,a) (flatten a))) w))

(defun sort-flatten (w) (sort (flat w) #'<))

> (flatten '(7 ((3 5 (((4) 6 1) 0)) 2)))
(7 3 5 4 6 1 0 2)
> (sort-flatten '(7 ((3 5 (((4) 6 1) 0)) 2)))
(0 1 2 3 4 5 6 7)

```

Решение 4.308.2

```

(defun flatten (w)
  (if (atom w) `(,w) (mapcan #'flatten w)))

(defun sort-flatten (w) (sort (flat w) #'<))

> (flatten '(7 ((3 5 (((4) 6 1) 0)) 2)))
(7 3 5 4 6 1 0 2)
> (sort-flatten '(7 ((3 5 (((4) 6 1) 0)) 2)))
(0 1 2 3 4 5 6 7)

```

Задача 4.309 *most-frequent.lisp*

Определить функцию, возвращающую список наиболее часто встречающихся элементов списка.

Решение 4.309.1

```

(defun most-frequent (w
  &aux
  (v (remove-duplicates w))
  (m (loop for a in v
    maximize
    (count a w :test #'equalp))))
  (remove-if
    #'(lambda (a) (< (count a w :test #'equalp) m)) v))

```

```
> (most-frequent '(a b c a b))
(A B)
> (most-frequent '((a) (b) c (a) (b)))
((A) (B) (A) (B))
```

Задача 4.310 *cleave.lisp*

Есть список списков: ((1 2 3 4) (5 6 7 8) (9 10 11 12)) и список с индексами: ((0 2) (1 3) (0 1)), где указано какие элементы нужно выбрать. Вернуть два списка - выбранных: ((1 3) (6 8) (9 10)) и оставшихся элементов: ((2 4) (5 7) (11 12)).

Решение 4.310.1

```
(defun cleave (w v)
  (loop for e in (mapcar
    #'(lambda (m n)
      (loop for a in m for i upfrom 0
        if (find i n) collect a into x
        else collect a into z
        finally (return (list x z))))
    w v)
    collect (car e) into f
    collect (cadr e) into g
    finally (return (values f g))))

> (cleave '((1 2 3 4) (5 6 7 8) (9 10 11 12)) '((0 2) (1 3) (0 1)))
((1 3) (6 8) (9 10))
((2 4) (5 7) (11 12))
```

Задача 4.311 *compress.lisp*

Дан список: ((1 r) (2 r) (3 i) (4 i) (5 x) (6 e) (7 e) (8 e) (9 z)). Ужать его в такой: ((1 2 R) (3 4 I) (5 5 X) (6 8 E) (9 9 Z)).

Решение 4.311.1

```
(defun compress (w
  &optional
  b
  ac
  acc
  &aux
  (a (car w))
  (d (cdr w)))
  (cond ((null w) (reverse (cons (reverse (cons b
    (cons (car ac)
    (last ac))))
    acc)))
    ((null b) (compress d
    (cadr a))
```



```
> (compress '((1 r) (2 r) (3 i) (4 i) (5 x) (6 e) (7 e) (8 e) (9 z)))
((1 2 R) (3 4 I) (5 X) (6 8 E) (9 Z))
```

Задача 4.312 deep-atoms.lisp

Написать программу замены элементов списков первого уровня на элементы списка самого низкого уровня в цикл. Например, `((a)) (b) ((c) e) ((d)))` -> `((C) (D) (C) (D))`.

Решение 4.312.1

```
(defun max-depth (w)
  (cond ((null w) 0)
        ((atom (car w)) (max-depth (cdr w)))
        ((max (1+ (max-depth (car w))) (max-depth (cdr w))))))

(defun dive-elms (n w &aux (a (car w)) (d (cdr w)))
  (cond ((null w) nil)
        ((= n 0) w)
        ((atom a) (dive-elms n d))
        ((nconc (dive-elms (1- n) a) (dive-elms n d)))))

(defun exchange (w v &optional (z v))
  (cond ((null w) nil)
        ((null z) (exchange w v v))
        ((cons (car z) (exchange (cdr w) v (cdr z))))))

(defun deep-atoms (w)
  (mapcar #'list (exchange w (dive-elms (max-depth w) w))))

> (deep-atoms '((a)) (b) ((c) e) ((d)))
((C) (D) (C) (D))
```

Задача 4.313 add-matrices.lisp

Сложить две целочисленные матрицы.

Решение 4.313.1

```
(defun add-lists (w v)
  (cond ((null w) nil)
        (t (cons (+ (car w) (car v))
                   (add-lists (cdr w) (cdr v))))))

(defun add-matrices (w v)
  (cond ((null w) nil)
        (t (cons (add-lists (car w) (car v))
                   (add-matrices (cdr w) (cdr v))))))

> (add-matrices '((1 2 3) (4 5 6) (7 8 9)) '((1 2 3) (4 5 6) (7 8 9)))
```

```
((2 4 6) (8 10 12) (14 16 18))
```

Решение 4.313.2

```
(defun add-lists (w v)
  (when w (cons (+ (car w) (car v))
                (add-lists (cdr w) (cdr v)))))

(defun add-matrices (w v)
  (when w (cons (add-lists (car w) (car v))
                (add-matrices (cdr w) (cdr v)))))

> (add-matrices '((1 2 3) (4 5 6) (7 8 9)) '((1 2 3) (4 5 6) (7 8 9)))
((2 4 6) (8 10 12) (14 16 18))
```

Решение 4.313.3

```
(defun add-matrices (w v)
  (loop for a in w for b in v
        collect (loop for c in a for d in b
                      collect (+ c d))))

> (add-matrices '((1 2 3) (4 5 6) (7 8 9)) '((1 2 3) (4 5 6) (7 8 9)))
((2 4 6) (8 10 12) (14 16 18))
```

Решение 4.313.4

```
(defun add-matrices (w v)
  (loop for a in w for b in v
        collect (mapcar #'(+ a b))))

> (add-matrices '((1 2 3) (4 5 6) (7 8 9)) '((1 2 3) (4 5 6) (7 8 9)))
((2 4 6) (8 10 12) (14 16 18))
```

Решение 4.313.5

```
(defun add-matrices (w v)
  (mapcar #'(lambda (a b) (mapcar #'(+ a b)) w v))

> (add-matrices '((1 2 3) (4 5 6) (7 8 9)) '((1 2 3) (4 5 6) (7 8 9)))
((2 4 6) (8 10 12) (14 16 18))
```

Задача 4.314 *in-depth.lisp*

Написать программу формирования списка, состоящего из всех списков, которые не являются списками самого высокого уровня и списками самого низкого уровня для заданного списка.

Решение 4.314.1

```
(defun max-depth (w)
  (cond ((null w) 0)
```

```

((atom (car w)) (max-depth (cdr w)))
(max (1+ (max-depth (car w))) (max-depth (cdr w)))))

(defun dive (n w &aux (a (car w)) (d (cdr w)))
  (cond ((null w) nil)
        ((= n 0) (list w))
        ((atom a) (dive n d))
        ((nconc (dive (1- n) a) (dive n d)))))

(defun in-depth (w &aux (m (max-depth w)))
  (loop for a from 2 to (1- m) nconc (dive a w)))

> (max-depth '(a (b (c) ((1))) d (e (f (g) h))))
3
> (dive 0 '(a (b (c) ((1))) d (e (f (g) h))))
((A (B (C) ((1))) D (E (F (G) H))))
> (dive 1 '(a (b (c) ((1))) d (e (f (g) h))))
((B (C) ((1))) (E (F (G) H)))
> (dive 2 '(a (b (c) ((1))) d (e (f (g) h))))
((C) ((1)) (F (G) H))
> (dive 3 '(a (b (c) ((1))) d (e (f (g) h))))
((1) (G))
> (dive 4 '(a (b (c) ((1))) d (e (f (g) h))))
NIL
> (in-depth '(a (b (c) ((1))) d (e (f (g) h))))
((C) ((1)) (F (G) H))

```

Задача 4.315 peak-numbers.lisp

Find peak elements and return its indexes (A peak element is an element that is greater than its neighbors).

Вернуть номера пиковых элементов числового списка (пиковым является элемент, который больше, чем соседние элементы).

Решение 4.315.1

```

(defun peak-numbers (w &optional (n 1))
  (cond ((null (cddr w)) nil)
        ((and (> (cadr w) (car w))
              (> (cadr w) (caddr w)))
         (cons n (peak-numbers (cdr w) (1+ n))))
        ((peak-numbers (cdr w) (1+ n)))))

> (peak-numbers '(0 1 0 1 2 3 1))
(1 5)

```

Решение 4.315.2

```

(defun peak-numbers (w &optional (n 1) &aux (d (cdr w)))
  (cond ((null (cdr d)) nil)
        ((and (> (car d) (car w))
              (> (car d) (cadr d)))
         (cons n (peak-numbers (cdr w) (1+ n) d))))

```

```
(> (car d) (cadr d)))
(cons n (peak-numbers d (1+ n))))
((peak-numbers d (1+ n))))

> (peak-numbers '(0 1 0 1 2 3 1))
(1 5)
```

Задача 4.316 *maximum-product-sequence.lisp*

Find the contiguous subarray within an array (containing at least one number) which has the largest product. For example, given the array $[2,3,-2,4]$, the contiguous subarray $[2,3]$ has the largest product = 6.

Дан числовой список. Найти непрерывную последовательность (последовательности), произведение элементов которой является наибольшим.

Решение 4.316.1

```
(defun accs (w &optional (acc (list (car w))))
  (if (cdr w)
      (cons (reverse acc)
            (accs (cdr w) (cons (cadr w) acc)))
      (list (reverse acc))))

(defun maximum-product-sequence (w
                                &aux
                                (v (loop for d on w
                                           nconc (accs d)))
                                (z (mapcar
                                    #'(lambda (a)
                                        (list a
                                              (reduce #'* a)))
                                      v))
                                (m (reduce #'max z :key #'cadr)))
  (mapcar #'car (remove-if-not #'(lambda (a) (= (cadr a) m)) z)))

> (maximum-product-sequence '(1 2 3 0 1 3 4 8 -4 -6 0 3 4))
((1 3 4 8 -4 -6) (3 4 8 -4 -6))
> (maximum-product-sequence '(2 3 -2 4))
((2 3))
```

Решение 4.316.2

```
(defun accs (w &optional (acc (list (car w))) &aux (d (cdr w)))
  (if d
      (cons (reverse acc)
            (accs d (cons (car d) acc)))
      (list (reverse acc))))

(defun maximum-product-sequence (w
                                &aux
                                (v (loop for d on w
```

```

                nconc (accs d)))
(z (mapcar
   #'(lambda (a)
        `(,a ,(reduce #'* a)))
   v))
(m (reduce #'max z :key #'cadr)))
(mapcar #'car (remove-if-not #'(lambda (a) (= (cadr a) m)) z)))

> (maximum-product-sequence '(1 2 3 0 1 3 4 8 -4 -6 0 3 4))
((1 3 4 8 -4 -6) (3 4 8 -4 -6))
> (maximum-product-sequence '(2 3 -2 4))
((2 3))

```

Задача 4.317 *annex-<.lisp*

Определить функцию для преобразования списка по принципу: '(a b c d e) -> (A (B (C (D (E)))))

Решение 4.317.1

```

(defun annex-< (w)
  (reduce #'list (butlast w) :from-end t :initial-value (last w)))

> (annex-< '(a b c d e))
(A (B (C (D (E)))))

```

Решение 4.317.2

```

(defun annex-< (w)
  (if (cdr w) (cons (car w) (list (annex-< (cdr w)))) w))

> (annex-< '(a b c d e))
(A (B (C (D (E)))))

```

Решение 4.317.3

```

(defun annex-< (w &aux (d (cdr w)))
  (if d (cons (car w) `(,(annex-< d))) w))

> (annex-< '(a b c d e))
(A (B (C (D (E)))))

```

Задача 4.318 *dump-top-level-duplicates.lisp*

Определить функцию, удаляющую из списка все повторяющиеся элементы (атомы и списки) на верхнем уровне.

Решение 4.318.1

```

(defun dump-top-level-duplicates (w)
  (cond ((null w) nil)

```

```

((member (car w) (cdr w) :test #'equalp)
 (dump-top-level-duplicates (remove (car w) (cdr w)
                                     :test #'equalp)))
((cons (car w) (dump-top-level-duplicates (cdr w))))))

> (dump-top-level-duplicates '(1 (2 2 3) 4 (3 2 3) (8 9) 5 1 (8 9) (2
2 3)))
(4 (3 2 3) 5)

```

Решение 4.318.2

```

(defun dump-top-level-duplicates (w)
  (when w (if (member (car w) (cdr w) :test #'equalp)
              (dump-top-level-duplicates (remove (car w) (cdr w)
                                                  :test #'equalp))
              (cons (car w) (dump-top-level-duplicates (cdr w))))))

> (dump-top-level-duplicates '(1 (2 2 3) 4 (3 2 3) (8 9) 5 1 (8 9) (2
2 3)))
(4 (3 2 3) 5)

```

Решение 4.318.3

```

(defun dump-top-level-duplicates (w
                                  &optional ac
                                  &aux (a (car w)) (d (cdr w)))
  (cond ((null w) (reverse ac))
        ((member a d :test #'equalp) (dump-top-level-duplicates
                                       (remove a d :test #'equalp)
                                       ac))
        (t (dump-top-level-duplicates d (cons a ac)))))

> (dump-top-level-duplicates '(1 (2 2 3) 4 (3 2 3) (8 9) 5 1 (8 9) (2
2 3)))
(4 (3 2 3) 5)

```

Решение 4.318.4

```

(defun dump-top-level-duplicates (w)
  (loop for a in w
        when (= (count a w :test #'equalp) 1)
        collect a))

> (dump-top-level-duplicates '(1 (2 2 3) 4 (3 2 3) (8 9) 5 1 (8 9) (2
2 3)))
(4 (3 2 3) 5)

```

Задача 4.319 *listt.lisp*

Для произвольного списка построить список той же длины, состоящий из атомов `t` и `nil` в зависимости от того, является ли соответствующий элемент исходного списка списком или атомом соответственно

Решение 4.319.1

```
(defun listt (w)
  (when w (cons (listp (car w))
                 (listt (cdr w)))))

> (listt '((a) a))
(T NIL)
```

Решение 4.319.2

```
(defun listt (w) (loop for a in w collect (listp a)))

> (listt '((a) a))
(T NIL)
```

Решение 4.319.3

```
(defun listt (w) (mapcar #'listp w))

> (listt '((a) a))
(T NIL)
```

Задача 4.320 *max-number.lisp*

Для произвольного списка построить список той же длины, состоящий из атомов `t` и `nil` в зависимости от того, является ли соответствующий элемент исходного списка списком или атомом соответственно

Решение 4.320.1

```
(defun flat (w &optional acc)
  (cond ((null w) acc)
        ((atom w) (cons w acc))
        ((flat (car w) (flat (cdr w) acc)))))

(defun max-number (w
                  &aux
                    (a (length (car w)))
                    (v (flat w))
                    (m (reduce #'max v))
                    (p (position m v)))
  (list m (floor p a) (rem p a)))

> (max-number '((1 2 3 4) (-5 8 3 5)))
(8 1 1)
> (max-number '((1 2 3) (-1 -2 5) (3 4 2)))
(5 1 2)
```

Решение 4.320.2

```

(defun flat (w)
  (loop for a in w
        if (listp a) append (flat a)
        else collect a))

(defun max-number (w
                  &aux
                    (a (length (car w)))
                    (v (flat w))
                    (m (reduce #'max v))
                    (p (position m v)))
  (values m (floor p a) (rem p a)))

> (max-number '((1 2 3 4) (-5 8 3 5)))
8
1
1
> (max-number '((1 2 3) (-1 -2 5) (3 4 2)))
5
1
2

```

Задача 4.321 *dive.lisp*

Из заданного списка (a (b (c) d ((e ((f))))) получив список (f b (a)), используя базовые функции cons, car, cdr.

Решение 4.321.1

```

(defun dive (w)
  (cons (caaar (cdaadr (cddadr w)))
        (cons (caadr w)
              (cons (cons (car w) nil)
                    nil))))

> (dive '(a (b (c) d ((e ((f)))))
(F B (A))

```

Задача 4.322 *rearrange.lisp*

Поменять местами элементы списка:

```

(a1 a2 a3 a4)
(a3 a1 a2 a4).

```

Решение 4.322.1

```

(defun rearrange (w) (rotatef (nth 1 w) (nth 0 w) (nth 2 w)) w)

> (rearrange '(a1 a2 a3 a4))
(A3 A1 A2 A4)

```


Задача 4.323 random-product.lisp

Увеличить каждый элемент списка ((a) (b)) ((c) (d)) ... в произвольное число раз.

Решение 4.323.1

```
(defun random-product (w &optional (n (random 1000)))
  (mapcar #'(lambda (a) `((, (* (caar a) n)) (, (* (caadr a) n)))) w))

> (random-product '(((1) (2)) ((3) (4))))
((993) (1986)) ((2979) (3972)))
> (random-product '(((1) (2)) ((3) (4))))
((174) (348)) ((522) (696)))
```

Задача 4.324 infixlisp

Напишите программу, которая преобразует префиксную запись арифметического выражения в инфиксную.

Решение 4.324.1

```
(defun f (w)
  (when w (if (atom w) w (infix w))))

(defun infix (w &optional ac (p (car w)) (d (cdr w)))
  (if (cdr d)
      (infix (cdr d) (nconc ac (list (f (car d)) p)) p (cdr d))
      (nconc ac (list (f (car d))))))

> (infix '(* (+ 3 4 6) (- 1 2 3) 8 9))
((3 + 4 + 6) * (1 - 2 - 3) * 8 * 9)
```

Задача 4.325 plusp>min.lisp

Дан одноуровневый числовой список. Определите функцию замены положительных элементов на минимальный.

Решение 4.325.1

```
(defun plusp>min (w &optional (m (reduce #'min w)) &aux (a (car w)))
  (cond ((null w) nil)
        (t (cons (if (plusp a) m a) (plusp>min (cdr w) m)))))

> (plusp>min '(1 2 3 -1 4 -5))
(-5 -5 -5 -1 -5 -5)
```

Решение 4.325.2

```
(defun plusp>min (w &optional (m (reduce #'min w)) &aux (a (car w)))
  (when w (cons (if (plusp a) m a) (plusp>min (cdr w) m))))
```

```
> (plusp>min '(1 2 3 -1 4 -5))
(-5 -5 -5 -1 -5 -5)
```

Решение 4.325.3

```
(defun plusp>min (w &aux (m (reduce #'min w)))
  (mapcar #'(lambda (a) (if (plusp a) m a)) w))
```

```
> (plusp>min '(1 2 3 -1 4 -5))
(-5 -5 -5 -1 -5 -5)
```

Решение 4.325.4

```
(defun plusp>min (w &aux (m (reduce #'min w)))
  (substitute-if m #'plusp w))
```

```
> (plusp>min '(1 2 3 -1 4 -5))
(-5 -5 -5 -1 -5 -5)
```

Решение 4.325.5

```
(defun plusp>min (w)
  (substitute-if (reduce #'min w) #'plusp w))
```

```
> (plusp>min '(1 2 3 -1 4 -5))
(-5 -5 -5 -1 -5 -5)
```

Задача 4.326 *plusp>min-deep.lisp*

Дан многоуровневый числовой список. Определите функцию замены положительных элементов на минимальный.

Решение 4.326.1

```
(defun plusp>min-deep (w
  &optional (m (reduce #'min (flat w)))
  &aux (a (car w)) (d (cdr w)))
  (cond ((null w) nil)
        ((listp a) (cons (plusp>min-deep a m) (plusp>min-deep d m)))
        (t (cons (if (plusp a) m a) (plusp>min-deep d m)))))
```

```
(defun flat (w &optional acc)
  (cond ((null w) acc)
        ((atom w) (cons w acc))
        ((flat (car w) (flat (cdr w) acc)))))
```

```
> (plusp>min-deep '(1 (2 (3)) ((-1) 4) ((-5))))
(-5 (-5 (-5)) ((-1) -5) ((-5)))
```

Решение 4.326.2

```
(defun plusp>min-deep (w
  &optional (m (reduce #'min (flat w)))
  &aux (a (car w)) (d (cdr w)))
  (cond ((null w) nil)
        ((listp a) (cons (plusp>min-deep a m) (plusp>min-deep d m)))
        (t (cons (if (plusp a) m a) (plusp>min-deep d m)))))

(defun flat (w)
  (loop for a in w
        if (listp a) append (flat a)
        else collect a))

> (plusp>min-deep '(1 (2 (3)) ((-1) 4) ((-5))))
(-5 (-5 (-5)) ((-1) -5) ((-5)))
```

Решение 4.326.3

```
(defun plusp>min-deep (w
  &optional (m (reduce #'min (flat w)))
  &aux (a (car w)) (d (cdr w)))
  (cond ((null w) nil)
        ((listp a) (cons (plusp>min-deep a m) (plusp>min-deep d m)))
        (t (cons (if (plusp a) m a) (plusp>min-deep d m)))))

(defun flat (w)
  (mapcan #'(lambda (a) (if (listp a) (flat a) `(,a))) w))

> (plusp>min-deep '(1 (2 (3)) ((-1) 4) ((-5))))
(-5 (-5 (-5)) ((-1) -5) ((-5)))
```

Решение 4.326.4

```
(defun plusp>min-deep (w
  &optional (m (reduce #'min (flat w)))
  &aux (a (car w)))
  (when w (cons (if (listp a)
                    (plusp>min-deep a m)
                    (if (plusp a) m a))
                (plusp>min-deep (cdr w) m))))

(defun flat (w)
  (mapcan #'(lambda (a) (if (listp a) (flat a) `(,a))) w))

> (plusp>min-deep '(1 (2 (3)) ((-1) 4) ((-5))))
(-5 (-5 (-5)) ((-1) -5) ((-5)))
```

Задача 4.327 *abs-average.lisp*

Дан многоуровневый числовой список. Определите функцию замены положительных элементов на минимальный.

Решение 4.327.1

```

(defun sum (w) (if (null w) 0 (+ (car w) (sum (cdr w)))))

(defun our-length (w) (if (null w) 0 (1+ (our-length (cdr w)))))

(defun average (w) (float (/ (sum w) (our-length w))))

(defun abs-average (w &optional (m (average w)))
  (cond ((null w) nil)
        ((> (abs (car w)) m) (cons (* (car w) -1)
                                     (abs-average (cdr w) m)))
        (t (cons (car w) (abs-average (cdr w) m)))))

> (abs-average '(1 2 3 4))
(1 2 -3 -4)

```

Решение 4.327.2

```

(defun average (w) (float (/ (reduce #'+ w) (length w))))

(defun abs-average (w &optional (m (average w)) &aux (a (car w)))
  (cond ((null w) nil)
        ((> (abs a) m) (cons (* a -1) (abs-average (cdr w) m)))
        ((cons a (abs-average (cdr w) m)))))

> (abs-average '(1 2 3 4))
(1 2 -3 -4)

```

Решение 4.327.3

```

(defun average (w) (float (/ (reduce #'+ w) (length w))))

(defun abs-average (w &optional (m (average w)) &aux (a (car w)))
  (when w (if (> (abs a) m)
              (cons (* a -1) (abs-average (cdr w) m))
              (cons a (abs-average (cdr w) m)))))

> (abs-average '(1 2 3 4))
(1 2 -3 -4)

```

Решение 4.327.4

```

(defun average (w) (float (/ (reduce #'+ w) (length w))))

(defun abs-average (w &optional (m (average w)) &aux (a (car w)))
  (when w (cons (if (> (abs a) m) (* a -1) a)
                (abs-average (cdr w) m))))

> (abs-average '(1 2 3 4))
(1 2 -3 -4)

```

Решение 4.327.5

```

(defun average (w)
  (loop for a in w
        sum a into s
        count a into c
        finally (return (float (/ s c)))))

(defun abs-average (w &aux (m (average w)))
  (loop for a in w collect (if (> (abs a) m) (* a -1) a)))

> (abs-average '(1 2 3 4))
(1 2 -3 -4)

```

Решение 4.327.6

```

(defun average (w) (float (/ (reduce #'+ w) (length w))))

(defun abs-average (w &aux (m (average w)))
  (mapcar #'(lambda (a) (if (> (abs a) m) (* a -1) a)) w))

> (abs-average '(1 2 3 4))
(1 2 -3 -4)

```

Решение 4.327.7

```

(defun abs-average (w
  &aux
  (m (float (/ (reduce #'+ w)
                (length w)))))
  (mapcar #'(lambda (a)
    (if (> (abs a) m) (* a -1) a)) w))

> (abs-average '(1 2 3 4))
(1 2 -3 -4)

```

Решение 4.327.8

```

(defun abs-average (w
  &aux
  (m (float (/ (reduce #'+ w)
                (length w)))))
  (loop for a in w
        collect (if (> (abs a) m) (* a -1) a)))

> (abs-average '(1 2 3 4))
(1 2 -3 -4)

```

Задача 4.328 reverse-top.lisp

Определите функцию с одним аргументом: списком списков. Функция должна вернуть список перевернутых списков.

Решение 4.328.1

```
(defun reverse-top (w)
  (when w (cons (reverse (car w))
                 (reverse-top (cdr w))))))
```

```
> (reverse-top '((a b)((c) d) (e g)))
((B A) (D (C)) (G E))
```

Решение 4.328.2

```
(defun reverse-top (w) (loop for a in w collect (reverse a)))
```

```
> (reverse-top '((a b)((c) d) (e g)))
((B A) (D (C)) (G E))
```

Решение 4.328.3

```
(defun reverse-top (w)
  (mapcar #'(lambda (a) (if (atom a) a (reverse a))) w))
```

```
> (reverse-top '((a b)((c) d) (e g)))
((B A) (D (C)) (G E))
```

Решение 4.328.4

```
(defun reverse-top (w) (mapcar #'reverse w))
```

```
> (reverse-top '((a b)((c) d) (e g)))
((B A) (D (C)) (G E))
```

Задача 4.329 row-sums.lisp

Определите функцию, которая получает список целых чисел и выдает список, содержащий в качестве n -го элемента суммы первых n членов исходного списка. Например, $(3\ 2\ 6\ 5\ 1) > (3\ 5\ 11\ 16\ 17)$.

Решение 4.329.1

```
(defun row-sums (w) (cdr (sums w)))

(defun sums (w &optional (s 0))
  (cond ((null w) (list s))
        ((cons s (sums (cdr w) (+ s (car w)))))))
```

```
> (row-sums '(3 2 6 5 1))
(3 5 11 16 17)
```

Решение 4.329.2

```
(defun row-sums (w) (cdr (sums w)))

(defun sums (w &optional (s 0))
```

```
(if w (cons s (sums (cdr w) (+ s (car w)))) (list s)))

> (row-sums '(3 2 6 5 1))
(3 5 11 16 17)
```

Решение 4.329.3

```
(defun row-sums (w)
  (cdr (labels ((sum (w s)
                 (cond ((null w) (list s))
                       ((cons s (sums (cdr w)
                                       (+ s (car w))))))
                 (sum w 0))))))

> (row-sums '(3 2 6 5 1))
(3 5 11 16 17)
```

Решение 4.329.4

```
(defun row-sums (w)
  (cdr (labels ((sum (w s)
                 (if w
                     (cons s (sums (cdr w)
                                     (+ s (car w))))
                     (list s))))
        (sum w 0))))

> (row-sums '(3 2 6 5 1))
(3 5 11 16 17)
```

Задача 4.330 *clear-duplicates.lisp*

Определите функцию удаляющую из списка все элементы, входящие в список не один раз, кроме последнего вхождения.

Решение 4.330.1

```
(defun clear-duplicates (w &aux (a (car w)) (d (cdr w)))
  (when w (if (member a d)
              (clear-duplicates d)
              (cons a (clear-duplicates d)))))

> (clear-duplicates '(1 2 3 a 1 b 2 c 3))
(A 1 B 2 C 3)
```

Решение 4.330.2

```
(defun clear-duplicates (w &aux (a (car w)) (d (cdr w)))
  (when w (if (member a d)
              (clear-duplicates d)
              (cons a (clear-duplicates d)))))
```

```
> (clear-duplicates '(1 2 3 a 1 b 2 c 3))
(A 1 B 2 C 3)
```

Решение 4.330.3

```
(defun clear-duplicates (w &optional ac &aux (a (car w)) (d (cdr w)))
  (cond ((null w) (reverse ac))
        ((member a d) (clear-duplicates d ac))
        ((clear-duplicates d (cons a ac)))))
```

```
> (clear-duplicates '(1 2 3 a 1 b 2 c 3))
(A 1 B 2 C 3)
```

Решение 4.330.4

```
(defun clear-duplicates (w &optional ac &aux (a (car w)) (d (cdr w)))
  (cond ((null w) (reverse ac))
        ((clear-duplicates d (if (member a d) ac (cons a ac)))))
```

```
> (clear-duplicates '(1 2 3 a 1 b 2 c 3))
(A 1 B 2 C 3)
```

Решение 4.330.5

```
(defun clear-duplicates (w
                        &optional
                        ac
                        &aux
                        (a (car w))
                        (d (cdr w)))
  (if w
      (clear-duplicates d (if (member a d)
                              ac
                              (cons a ac)))
      (reverse ac)))
```

```
> (clear-duplicates '(1 2 3 a 1 b 2 c 3))
(A 1 B 2 C 3)
```

Решение 4.330.6

```
(defun clear-duplicates (w) (clear (reverse w)))

(defun clear (w &optional ac)
  (if w (clear (cdr w) (adjoin (car w) ac)) ac))
> (clear-duplicates '(1 2 3 a 1 b 2 c 3))
(A 1 B 2 C 3)
```

Решение 4.330.7

```
(defun clear-duplicates (w)
  (labels ((clear (v ac)
             (if v (clear (cdr v) (adjoin (car v) ac)) ac)))
```



```
(clear (reverse w) nil)))

> (clear-duplicates '(1 2 3 a 1 b 2 c 3))
(A 1 B 2 C 3)
```

Задача 4.331 bubble-sort.lisp

Реализовать сортировку методом пузырька.

Решение 4.331.1

```
(defun bubble-sort (w)
  (do ((i (- (length w) 1) (- i 1)))
      ((= i 0))
      (do ((j 0 (+ j 1)))
          ((= j i))
          (when (< (nth i w) (nth j w))
            (let ((tmp (nth i w)))
              (setf (nth i w) (nth j w) (nth j w) tmp))))))
  w)

> (bubble-sort '(3 2 1))
(1 2 3)
```

Задача 4.332 drop-core.lisp

Написать функцию, которая удаляет из списка нечетной длины элемент, стоящий в середине списка.

Решение 4.332.1

```
(defun drop-core (w &aux (n (length w)))
  (if (oddp n) (drop w (floor n 2)) w))

(defun drop (w n)
  (if (zerop n) (cdr w) (cons (car w) (drop (cdr w) (1- n)))))

> (drop-core '(1 2 3 4 5 6))
(1 2 3 4 5 6)
> (drop-core '(1 2 3 4 5))
(1 2 4 5)
> (drop-core '(1))
NIL
> (drop-core '())
NIL
```

Решение 4.332.2

```
(defun drop-core (w &aux (n (length w)) (z (floor n 2)))
  (if (oddp n)
      (nconc (subseq w 0 z)
```

```

        (subseq w (1+ z)))
      w))

> (drop-core '(1 2 3 4 5 6))
(1 2 3 4 5 6)
> (drop-core '(1 2 3 4 5))
(1 2 4 5)
> (drop-core '(1))
NIL
> (drop-core '())

```

Решение 4.332.3

```

(defun drop-core (w &aux (n (length w)))
  (if (oddp n) (drop w (floor n 2))
      w))

(defun drop (w n)
  (loop for a in w for b upfrom 0
        unless (= b n) collect a))

> (drop-core '(1 2 3 4 5 6))
(1 2 3 4 5 6)
> (drop-core '(1 2 3 4 5))
(1 2 4 5)
> (drop-core '(1))
NIL
> (drop-core '())

```

Задача 4.333 front-end-pairs.lisp

Дан список. Определить функцию, которая возвращает список пар: первый элемент с последним, второй с предпоследним и т. д.

Решение 4.333.1

```

(defun front-end-pairs (w
  &aux
    (n (length w))
    (m (if (oddp n)
            (ceiling n 2)
            (/ n 2))))
  (when w (mapcar #'list
    (subseq w 0 m)
    (reverse (nthcdr (1- m) w))))))

> (front-end-pairs '(1 2 3 4 5))
((1 5) (2 4) (3 3))
> (front-end-pairs '(1 2 3 4))
((1 4) (2 3))
> (front-end-pairs '(1 2 3))
((1 3) (2 2))

```

```
> (front-end-pairs '(1 2))
((1 2))
> (front-end-pairs '(1))
((1 1))
```

Задача 4.334 *frequent-column.lisp*

Дана матрица. Найти элемент с максимальной частотностью. Заменить элементы столбцов, в которых содержится такой элемент, на этот элемент.

Решение 4.334.1

```
(defun transpose (m) (apply #'mapcar #'list m))

(defun frequent-column (w
                        &aux
                        (n (length w))
                        (f (reduce #'append w))
                        (u (remove-duplicates f))
                        (v (loop for a in u
                                collect (list (count a f) a)))
                        (m (cadar (sort v #'> :key #'car))))
  (transpose
   (loop for a in (transpose w)
         if (find m a) collect (make-list n :initial-element m)
         else collect a)))

> (frequent-column '((a b c d) (b c c d) (a a a b)))
((A A A D) (A A A D) (A A A B))
```

Задача 4.335 *jam.lisp*

Создать из списка следующую последовательность, складывая попарно числа: (2 4 1 5 6 3) => ((2 4 1 5 6 3) (6 5 6 11 9) (11 11 17 20) (22 28 37) (50 65) (115)).

Решение 4.335.1

```
(defun jam (w)
  (if (cdr w) (cons w (jam (mapcar #'+ w (cdr w)))) `(),w)))

> (jam '(2 4 1 5 6 3))
((2 4 1 5 6 3) (6 5 6 11 9) (11 11 17 20) (22 28 37) (50 65) (115))
```

Задача 4.336 *onion.lisp*

Определить функцию, преобразующую исходный список, в список «луковицу». Например: вход: (3 2 1 2 3), выход: (3 (2 (1) 2) 3).

Решение 4.336.1

```
(defun onion (w)
  (if (null (cdr w)) w
      (list (car w)
            (onion (butlast (cdr w)))
            (car (last w)))))
```

```
> (onion '(1 2 3 2 1))
(1 (2 (3) 2) 1)
> (onion '(3 2 1 2 3))
(3 (2 (1) 2) 3)
```

Решение 4.336.2

```
(defun onion (w)
  (if (cdr w) (list (car w)
                    (onion (butlast (cdr w)))
                    (car (last w)))
      w))
```

```
> (onion '(1 2 3 2 1))
(1 (2 (3) 2) 1)
> (onion '(3 2 1 2 3))
(3 (2 (1) 2) 3)
```

Если в списке *w* два или больше элементов (*cdr w*), то функция *onion* возвращает список из первого элемента, вызова той же *onion* (это и есть рекурсия, когда функция вызывает саму себя), куда передается обрезанный с обеих сторон список (*butlast (cdr w)*) и последнего элемента (*car (last w)*). Если в списке остался лишь один элемент, то он возвращается в последней строке - *w*.

Задача 4.337 *aver-sine.lisp*

Дан список чисел *a1...an*. Получить список чисел *b1...bn*, где *bi* - среднее арифметическое всех членов последовательности *a1...an*, кроме *ai*.

Решение 4.337.1

```
(defun aver-sine (w
                  &optional
                  (s (reduce #'+ w))
                  (n (1- (length w))))
  (cond ((null w) nil)
        ((cons (float (/ (- s (car w)) n))
                (aver-sine (cdr w) s n)))))
```

```
> (aver-sine '(1 2 3 4 5))
(3.5 3.25 3.0 2.75 2.5)
```

Решение 4.337.2

```
(defun aver-sine (w
                  &optional
                  (s (reduce #' + w))
                  (n (1- (length w))))
  (when w (cons (float (/ (- s (car w)) n))
                (aver-sine (cdr w) s n))))

> (aver-sine '(1 2 3 4 5))
(3.5 3.25 3.0 2.75 2.5)
```

Решение 4.337.3

```
(defun aver-sine (w
                  &optional
                  (s (reduce #' + w))
                  (n (1- (length w))))
  (mapcar #'(lambda (a) (float (/ (- s a) n))) w))

> (aver-sine '(1 2 3 4 5))
(3.5 3.25 3.0 2.75 2.5)
```

Решение 4.337.4

```
(defun aver-sine (w
                  &optional
                  (s (reduce #' + w))
                  (n (1- (length w))))
  (loop for a in w collect (float (/ (- s a) n))))

> (aver-sine '(1 2 3 4 5))
(3.5 3.25 3.0 2.75 2.5)
```

Решение 4.337.5

```
(defun aver-sine (w
                  &optional
                  (s (reduce #' + w))
                  (n (1- (length w))))
  (when (cdr w) (loop for a in w collect (/ (- s a) n))))

> (aver-sine '(1 2 3 4 5 6 7))
(9/2 13/3 25/6 4 23/6 11/3 7/2)
```

Задача 4.338 count-more.lisp

Определите функцию, которая по данному списку формирует список его элементов, встречающихся в нем более n раз.

Решение 4.338.1

```
(defun count-more (n w &optional ac &aux (a (car w)))
  (cond ((null w) (reverse ac))
```

```

      ((> (count a w) n) (count-more n (cdr w) (adjoin a ac)))
      ((count-more n (cdr w) ac))))

> (count-more 1 '(1 2 3 1 2 4 5 6 3))
(1 2 3)
> (count-more 2 '(1 2 3 1 2 4 5 6 3))
NIL

```

Задача 4.339 half.lisp

Дан список чисел. Если максимальный элемент списка находится на первом месте, а минимум - на втором, то уменьшить все элементы списка в два раза.

Решение 4.339.1

```

(defun half (w)
  (if (and (cdr w)
           (equalp (car w) (reduce #'max w))
           (equalp (cadr w) (reduce #'min w)))
      (mapcar #'(lambda (a) (/ a 2)) w)
      w))

> (half '(10.0))
(10.0)
> (half '(10 10.0 10.0))
(5 5.0 5.0)

```

Решение 4.339.2

```

(defun half (w)
  (if (and (cdr w)
           (= (car w) (reduce #'max w))
           (= (cadr w) (reduce #'min w)))
      (mapcar #'(lambda (a) (/ a 2)) w)
      w))

> (half '(10.0))
(10.0)
> (half '(10 10.0 10.0))
(5 5.0 5.0)

```

Решение 4.339.3 * без функции mapcar

```

(defun half (w)
  (if (and (cdr w)
           (= (car w) (reduce #'max w))
           (= (cadr w) (reduce #'min w)))
      (div w 2)
      w))

(defun div (w n) (when w (cons (/ (car w) n) (div (cdr w) n))))

```

```
> (half '(10.0))
(10.0)
> (half '(10 10.0 10.0))
(5 5.0 5.0)
```

Задача 4.340 max-length-seq.lisp

Дан список чисел, Найти самую длинную убывающую или растущую последовательность.

Решение 4.340.1

```
(defparameter *w* (loop repeat 1000000 collect (random 1000000)))

(defun max-length-seq (w &aux (a (m w '<=)) (b (m w '>=)))
  (if (>= (length a) (length b)) a b))

(defun s (w p &optional (b (car w)) &aux (a (car w)))
  (when w (when (funcall p b a) (cons a (s (cdr w) p a)))))

(defun m (w p &optional a &aux (v (s w p)))
  (if w (m (cdr w) p (if (> (length v) (length a)) v a)) a))

> (max-length-seq '(8 4 2 3 4 5 2))
(2 3 4 5)
> (max-length-seq '(8 4 2 3 2))
(8 4 2)
> (time (max-length-seq *w*))
Real time: 3.656209 sec.
Run time: 3.619223 sec.
Space: 27489568 Bytes
GC: 7, GC time: 0.2964019 sec.
(834341 778338 759739 542349 453988 258028 130867 109354 105127 85714)
```

Решение 4.340.2

```
(defparameter *w* (loop repeat 1000000 collect (random 1000000)))

(defun max-length-seq (w &aux (a (m w '<=)) (b (m w '>=)))
  (if (>= (length a) (length b)) a b))

(defun s (w p &optional (b (car w)) &aux (a (car w)))
  (when (and w (funcall p b a)) (cons a (s (cdr w) p a)))))

(defun m (w p &optional a &aux (v (s w p)))
  (if w (m (cdr w) p (if (> (length v) (length a)) v a)) a))

> (max-length-seq '(8 4 2 3 4 5 2))
(2 3 4 5)
> (max-length-seq '(8 4 2 3 2))
(8 4 2)
```

```
> (time (max-length-seq *w*))
Real time: 3.600206 sec.
Run time: 3.5256226 sec.
Space: 27489568 Bytes
GC: 7, GC time: 0.2496016 sec.
(834341 778338 759739 542349 453988 258028 130867 109354 105127 85714)
```

Решение 4.340.3

```
(defparameter *w* (loop repeat 1000000 collect (random 1000000)))

(defun max-length-seq (w &aux (a (m w '<=)) (b (m w '>=)))
  (if (>= (length a) (length b)) a b))

(defun s (w p &optional ac (b (car w)) &aux (a (car w)))
  (if (and w (funcall p b a))
      (s (cdr w) p (cons a ac) a)
      (nreverse ac)))

(defun m (w p &optional a &aux (v (s w p)))
  (if w (m (cdr w) p (if (> (length v) (length a)) v a)) a))

> (max-length-seq '(8 4 2 3 4 5 2))
(2 3 4 5)
> (max-length-seq '(8 4 2 3 2))
(8 4 2)
> (time (max-length-seq *w*))
Real time: 3.8662212 sec.
Run time: 3.8064244 sec.
Space: 27489568 Bytes
GC: 7, GC time: 0.2808018 sec.
(834341 778338 759739 542349 453988 258028 130867 109354 105127 85714)
```

Решение 4.340.4

```
(defparameter *w* (loop repeat 1000000 collect (random 1000000)))

(defun max-length-seq (w &aux (a (m w '<=)) (b (m w '>=)))
  (if (>= (length a) (length b)) a b))

(defun s (w p &optional ac (b (car w)) &aux (a (car w)))
  (if (and w (funcall p b a))
      (s (cdr w) p (push a ac) a)
      (nreverse ac)))

(defun m (w p &optional a &aux (v (s w p)))
  (if w (m (cdr w) p (if (> (length v) (length a)) v a)) a))

> (max-length-seq '(8 4 2 3 4 5 2))
(2 3 4 5)
> (max-length-seq '(8 4 2 3 2))
(8 4 2)
> (time (max-length-seq *w*))
```



```

Real time: 3.8502202 sec.
Run time: 3.8376245 sec.
Space: 27489568 Bytes
GC: 7, GC time: 0.2340015 sec.
(834341 778338 759739 542349 453988 258028 130867 109354 105127 85714)

```

Решение 4.340.5 (* для растущих: автор - helter, www.cyberforum.ru)

```

(defparameter *w* (loop repeat 1000000 collect (random 1000000)))

(defun longfinder (test)
  (let ((longest)
        (long-len 0)
        (long-pos)
        (temp)
        (temp-len 0)
        (temp-pos)
        (pos 0))
    (lambda (el)
      (cond ((or (null temp) (funcall test (car temp) el))
             (push el temp)
             (incf temp-len))
            ((setf temp (list el))
             (setf temp-len 1)
             (setf temp-pos pos)))
      (when (> temp-len long-len)
        (setf longest temp)
        (setf long-len temp-len)
        (setf long-pos temp-pos))
      (incf pos)
      (values longest long-len long-pos))))

(defun max-length-seq ()
  (time
   (let ((f (longfinder #'<=)) longest long-len long-pos)
     (dolist (el *w* (values (nreverse longest) long-len long-pos))
       (setf (values longest long-len long-pos) (funcall f el))))))

```

```

> (max-length-seq)
Real time: 1.2800732 sec.
Run time: 1.2792082 sec.
Space: 8000080 Bytes
GC: 2, GC time: 0.0780005 sec.
(216291 325954 397023 596428 602798 603906 718373 783547 809859)
9
795048

```

Решение 4.340.6

```

(defparameter *w* (loop repeat 1000000 collect (random 1000000)))

(defun max-length-seq (w &aux (a (seq w '<=)) (b (seq w '>=)))
  (if (>= (length a) (length b)) a b)

```

```
(defun seq (w p)
  (loop with b = (car w) and v and z and n = 0 and m = 0
    for a in w if (funcall p b a) do (setf b a) (incf n)
    (setf v (cons a v)) else if (< m n) do (setf b a)
    (setf m n) (setf n 1) (setf z v) (setf v (list a))
    else do (setf b a) (setf v (list a)) (setf n 1)
    finally (return (if (> n m) (nreverse v) (nreverse z)))))
```

* CLISP:

```
> (time (max-length-seq *w*))
```

Real time: 1.1120636 sec.

Run time: 1.092007 sec.

Space: 16000000 Bytes

GC: 4, GC time: 0.156001 sec.

```
(834341 778338 759739 542349 453988 258028 130867 109354 105127 85714)
```

* SBCL:

```
> (time (max-length-seq *w*))
```

Evaluation took:

1.493 seconds of real time

1.497610 seconds of total run time (1.326009 user, 0.171601 system)

[Run times consist of 0.359 seconds GC time, and 1.139 seconds non-GC time.]

100.33% CPU

1,490,254,629 processor cycles

16,011,928 bytes consed

```
(149015 426512 488005 500559 563153 588156 782054 804240 865935
911191)
```

Решение 4.340.7

```
(defparameter *w* (loop repeat 10000000 collect (random 1000000)))
```

```
(defun max-length-seq (w &aux (a (seq w '<=)) (b (seq w '>=)))
  (if (>= (length a) (length b)) a b))
```

```
(defun seq (w p)
  (loop with b = (car w) and v and z and n = 0 and m = 0
    for a in w if (funcall p b a) do (setf b a) (incf n)
    (setf v (cons a v)) else if (< m n) do (setf b a)
    (setf m n) (setf n 1) (setf z v) (setf v (list a))
    else do (setf b a) (setf v (list a)) (setf n 1)
    finally (return (if (> n m) (nreverse v) (nreverse z)))))
```

* CLISP:

```
> (time (max-length-seq *w*))
```

Real time: 10.087618 sec.

Run time: 10.030865 sec.

Space: 160000000 Bytes

GC: 8, GC time: 1.3104084 sec.

```
(998733 964178 873281 843105 817805 794879 760122 730057 289221 285153
236462 230293)
```

```
* SBCL:
> (time (max-length-seq *w*))
Evaluation took:
  12.309 seconds of real time
  12.246078 seconds of total run time (11.310072 user, 0.936006 sys-
tem)
  [ Run times consist of 1.794 seconds GC time, and 10.453 seconds
non-GC time. ]
  99.49% CPU
  12,289,870,923 processor cycles
  160,076,152 bytes consed
(42992 80559 376000 423167 441165 484334 488868 601420 848775 870043
988607)
```

Решение 4.340.8

```
(defparameter *w* (loop repeat 10000000 collect (random 1000000)))

(defun s (p)
  (if (eq p '<=) '>= '<=))

(defun max-length-seq (w)
  (when (cdr w)
    (loop with b = (car w) and v and z and n = 0 and m = 0
      and p = (if (<= (car w) (cadr w)) '<= '>=)
      for a in w if (funcall p b a) do (setf b a) (incf n)
      (setf v (cons a v))
      else if (< m n) do (setf m n) (setf n 2) (setf z v)
      (setf v (list a b)) (setf b a) (setf p (s p))
      else do (setf b a) (setf v (list a)) (setf n 1)
      (setf p (s p))
      finally (return (if (> n m) (nreverse v) (nreverse z))))))

> (max-length-seq '(10 9 8 7 6 5 4 3 2 1 2 3 4 5 4 3 2 1 0 -1))
(10 9 8 7 6 5 4 3 2 1)
> (max-length-seq '(4 3 2 1 2 3 4 5 4 3 2 1 0 -1))
(5 4 3 2 1 0 -1)
> (max-length-seq '(4 3 2 1 2 3 4 5 4 3))
(1 2 3 4 5)
> (max-length-seq '(1))
NIL
> (max-length-seq nil)
NIL
```

```
* CLISP:
> (time (max-length-seq *w*))
Real time: 7.3926153 sec.
Run time: 7.3944473 sec.
Space: 80000064 Bytes
GC: 4, GC time: 1.1544074 sec.
(964178 873281 843105 817805 794879 760122 730057 289221 285153 236462
230293)
```

```

* SBCL:
> (time (max-length-seq *w*))
Evaluation took:
  5.850 seconds of real time
  5.834437 seconds of total run time (5.834437 user, 0.000000 system)
  [ Run times consist of 0.032 seconds GC time, and 5.803 seconds non-
GC time. ]
  99.73% CPU
  5,838,287,880 processor cycles
  80,045,512 bytes consed
(379806 418120 459598 477939 641898 712335 740597 790079 835458)

```

Решение 4.340.9

```

(defparameter *w* (loop repeat 10000000 collect (random 1000000)))

(defun s (p)
  (if (eq p '<=) '>= '<=))

(defun max-length-seq (w)
  (when (cdr w)
    (loop with b = (car w) and v and z and n = 0 and m = 0
      and p = (if (<= (car w) (cadr w)) '<= '>=)
      for a in w
      if (funcall p b a) do
        (setf b a v (cons a v)) (incf n)
      else if (< m n) do
        (setf m n n 2 z v v (list a b) b a p (s p))
      else do
        (setf b a v (list a) n 1 p (s p))
      finally (return (if (> n m) (nreverse v)
                          (nreverse z))))))

> (max-length-seq '(10 9 8 7 6 5 4 3 2 1 2 3 4 5 4 3 2 1 0 -1))
(10 9 8 7 6 5 4 3 2 1)
> (max-length-seq '(4 3 2 1 2 3 4 5 4 3 2 1 0 -1))
(5 4 3 2 1 0 -1)
> (max-length-seq '(4 3 2 1 2 3 4 5 4 3))
(1 2 3 4 5)
> (max-length-seq '(1))
NIL
> (max-length-seq nil)
NIL

```

```

* CLISP:
> (time (max-length-seq *w*))
Real time: 7.5794334 sec.
Run time: 7.5036483 sec.
Space: 80000048 Bytes
GC: 2, GC time: 0.9204059 sec.
(304587 320915 386605 500230 572204 616790 871531 912490 929658)

```

```
* SBCL:
> (time (max-length-seq *w*))
Evaluation took:
  6.502 seconds of real time
  6.489642 seconds of total run time (6.364841 user, 0.124801 system)
  [ Run times consist of 0.031 seconds GC time, and 6.459 seconds non-
GC time. ]
  99.82% CPU
  6,490,538,502 processor cycles
  80,040,352 bytes consed
(379806 418120 459598 477939 641898 712335 740597 790079 835458)
```

Задача 4.341 make-odd.lisp

Дан список положительных целых чисел. Определить функцию, которая возвращает список, превращая четные числа списка в нечетные, добавляя к ним единицу. Если число уже нечетное, то оно остается без изменений.

Решение 4.341.1

```
(defun make-odd (w)
  (cond ((null w) nil)
        ((evenp (car w)) (cons (1+ (car w)) (make-odd (cdr w))))
        (t (cons (car w) (make-odd (cdr w)))))

> (make-odd '(1 2 3 4 5 6))
(1 3 3 5 5 7)
```

Решение 4.341.2

```
(defun make-odd (w &aux (a (car w)) (d (cdr w)))
  (when w (if (evenp a)
              (cons (1+ a) (make-odd d))
              (cons a (make-odd d)))))

> (make-odd '(1 2 3 4 5 6))
(1 3 3 5 5 7)
```

Решение 4.341.3

```
(defun make-odd (w &aux (a (car w)))
  (when w (cons (+ (if (evenp a) 1 0) a) (make-odd (cdr w)))))

> (make-odd '(1 2 3 4 5 6))
(1 3 3 5 5 7)
```

Решение 4.341.4

```
(defun make-odd (w &aux (a (car w)) (d (cdr w)))
  (when w (cons (+ (if (evenp a) 1 0) a) (make-odd d)))))

> (make-odd '(1 2 3 4 5 6))
```

```
(1 3 3 5 5 7)
```

Решение 4.341.5

```
(defun make-odd (w)
  (mapcar #'(lambda (a) (+ (if (evenp a) 1 0) a)) w))

> (make-odd '(1 2 3 4 5 6))
(1 3 3 5 5 7)
```

Решение 4.341.6

```
(defun make-odd (w)
  (loop for a in w collect (+ (if (evenp a) 1 0) a)))

> (make-odd '(1 2 3 4 5 6))
(1 3 3 5 5 7)
```

Задача 4.342 *more-than-one.lisp*

Написать функцию, которая сохраняет дубликаты элементов списка в список. Например, `(f '(1 2 3 1 2 3 5))`, в результате должен получиться список `(1 2 3)`.

Решение 4.342.1

```
(defun more-than-one (w &optional (v (remove-duplicates w)))
  (cond ((null v) nil)
        ((= (count (car v) w) 1) (more-than-one w (cdr v)))
        ((cons (car v) (more-than-one w (cdr v)))))

> (more-than-one '(1 2 3 1 2 3 5))
(1 2 3)
> (more-than-one '(6 1 2 7 1 2 5))
(1 2)
```

Решение 4.342.2

```
(defun more-than-one (w &optional (v (remove-duplicates w)))
  (when v (if (= (count (car v) w) 1)
              (more-than-one w (cdr v))
              (cons (car v) (more-than-one w (cdr v))))))

> (more-than-one '(1 2 3 1 2 3 5))
(1 2 3)
> (more-than-one '(6 1 2 7 1 2 5))
(1 2)
```

Решение 4.342.3

```
(defun more-than-one (w &optional (v (remove-duplicates w)))
  (loop for a in v when (> (count a w) 1) collect a))
```

```
> (more-than-one '(1 2 3 1 2 3 5))
(1 2 3)
> (more-than-one '(6 1 2 7 1 2 5))
(1 2)
```

Решение 4.342.4

```
(defun more-than-one (w &optional (v (remove-duplicates w)))
  (remove-if #'(lambda (a) (= (count a w) 1)) v))

> (more-than-one '(1 2 3 1 2 3 5))
(1 2 3)
> (more-than-one '(6 1 2 7 1 2 5))
(1 2)
```

Решение 4.342.5

```
(defun more-than-one (w &optional (v (remove-duplicates w)) ac)
  (cond ((null v) (nreverse ac))
        ((= (count (car v) w) 1) (more-than-one w (cdr v) ac))
        ((more-than-one w (cdr v) (cons (car v) ac)))))

> (more-than-one '(1 2 3 1 2 3 5))
(1 2 3)
> (more-than-one '(6 1 2 7 1 2 5))
(1 2)
```

Решение 4.342.6

```
(defun more-than-one (w &optional (v (remove-duplicates w)) ac)
  (if v
      (more-than-one w (cdr v) (if (= (count (car v) w) 1)
                                     ac
                                     (cons (car v) ac)))
      (nreverse ac)))

> (more-than-one '(1 2 3 1 2 3 5))
(1 2 3)
> (more-than-one '(6 1 2 7 1 2 5))
(1 2)
```

Задача 4.343 square.lisp

Определить функцию, которая конструирует список из квадратов элементов данного списка целых чисел.

Решение 4.343.1

```
(defun square (w)
  (cond ((null w) nil)
        (t (cons (* (car w) (car w)) (square (cdr w))))))
```

```
> (square '(2 3))
(4 9)
```

Решение 4.343.2

```
(defun square (w)
  (when w (cons (* (car w) (car w)) (square (cdr w))))))

> (square '(2 3))
(4 9)
```

Решение 4.343.3

```
(defun square (w &optional ac)
  (cond ((null w) (nreverse ac))
        (t (square (cdr w) (cons (* (car w) (car w)) ac)))))

> (square '(2 3))
(4 9)
```

Решение 4.343.4

```
(defun square (w &optional ac)
  (if w
      (square (cdr w) (cons (* (car w) (car w)) ac))
      (nreverse ac)))

> (square '(2 3))
(4 9)
```

Решение 4.343.5

```
(defun square (w &optional ac &aux (a (car w)) (d (cdr w)))
  (if w (square d (cons (* a a) ac)) (nreverse ac)))

> (square '(2 3))
(4 9)
```

Решение 4.343.6

```
(defun square (w) (mapcar #'(lambda (a) (* a a)) w))

> (square '(2 3))
(4 9)
```

Решение 4.343.7 (автор - vlisp, www.cyberforum.ru)

```
(defun square (w) (mapcar #'* w w))

> (square '(2 3))
(4 9)
```


Решение 4.343.8

```
(defun square (w)
  (loop for a in w collect (* a a )))

> (square '(2 3))
(4 9)
```

Задача 4.344 *elms-positions.lisp*

Определить функцию, возвращающую позиции элементов одноуровневого списка.

Решение 4.344.1

```
(defun elm-positions (e w)
  (loop for a in w for b from 0
        when (equalp a e) collect b))

(defun elms-positions (w)
  (loop for a in (remove-duplicates w)
        collect (cons a (elm-positions a w))))

> (elms-positions '(a s d s a d a f a))
((S 1 3) (D 2 5) (F 7) (A 0 4 6 8))
```

Решение 4.344.2

```
(defun elm-positions (e w)
  (loop for a in w for b from 0
        when (equalp a e) collect b))

(defun elms-positions (w)
  (mapcar #'(lambda (a) (cons a (elm-positions a w)))
          (remove-duplicates w)))

> (elms-positions '(a s d s a d a f a))
((S 1 3) (D 2 5) (F 7) (A 0 4 6 8))
```

Решение 4.344.3

```
(defun elms-positions (w)
  (loop for a in (remove-duplicates w)
        collect (cons a (loop for e in w for b from 0
                              when (equalp a e) collect b))))

> (elms-positions '(a s d s a d a f a))
((S 1 3) (D 2 5) (F 7) (A 0 4 6 8))
```

Решение 4.344.4

```
(defun elms-positions (w)
```

```
(mapcar #'(lambda (a)
  (cons a (loop for e in w for b from 0
    when (equalp a e) collect b)))
(remove-duplicates w)))

> (elms-positions '(a s d s a d a f a))
((S 1 3) (D 2 5) (F 7) (A 0 4 6 8))
```

Задача 4.345 thirds.lisp

While any text editor can save your files,
only Emacs can save your soul.

Per Abrahamsen

Определить функцию, возвращающую каждый третий элемент списка.

Решение 4.345.1

```
(defun thirds (w &aux (n 0))
  (mapcan #'(lambda (a) (incf n)
    (when (= n 3) (setf n 0) (list a)))
    w))

> (thirds '(1 2 3 4 5 6))
(3 6)
```

Решение 4.345.2

```
(defun thirds (w)
  (when (cddr w) (cons (caddr w) (thirds (cddddr w)))))

> (thirds '(1 2 3 4 5 6))
(3 6)
```

Решение 4.345.3

```
(defun thirds (w)
  (loop for a in w for b from 1
    when (zerop (rem b 3)) collect a))

> (thirds '(1 2 3 4 5 6))
(3 6)
```

Решение 4.345.4

```
(defun thirds (w &optional ac)
  (if (cddr w)
    (thirds (cddddr w) (cons (caddr w) ac))
    (nreverse ac)))

> (thirds '(1 2 3 4 5 6))
```

```
(3 6)
```

Решение 4.345.5

```
(defun thirds (w)
  (loop for (a b c) on w by #'cdddr
        when c collect c))

> (thirds '(1 2 3 4 5 6))
(3 6)
```

Задача 4.346 double.lisp

Проверить список - если в списке не содержится данный элемент, то вернуть исходный список, если в списке содержится только один нужный символ, то его нужно продублировать, если больше одного, то вернуть nil.

Решение 4.346.1

```
(defun double (w g &aux (n (count g w)))
  (if (zerop n) w (when (= n 1) (d w g))))

(defun d (w g)
  (cond ((null w) nil)
        ((equalp (car w) g) (cons g w))
        ((cons (car w) (d (cdr w) g)))))

> (double '(a b c c d e) 'f)
(A B C C D E)
> (double '(a b c d e) 'c)
(A B C C D E)
> (double '(a b c c d e) 'c)
NIL
> (double '() 'c)
NIL
```

Решение 4.346.2

```
(defun double (w g &aux (n (count g w)))
  (if (zerop n) w (when (= n 1) (d w g))))

(defun d (w g)
  (when w (cons (car w)
                 (if (equalp (car w) g)
                     w
                     (d (cdr w) g)))))

> (double '(a b c c d e) 'f)
(A B C C D E)
> (double '(a b c d e) 'c)
(A B C C D E)
> (double '(a b c c d e) 'c)
NIL
```

```
NIL
> (double '() 'c)
NIL
```

Решение 4.346.3

```
(defun double (w g &aux (n (count g w)))
  (if (zerop n) w (when (= n 1) (d w g))))

(defun d (w g &optional ac)
  (cond ((null w) (nreverse ac))
        ((equalp (car w) g) (nconc (nreverse ac)
                                     (cons g w)))
        ((d (cdr w) g (cons (car w) ac)))))

> (double '(a b c c d e) 'f)
(A B C C D E)
> (double '(a b c d e) 'c)
(A B C C D E)
> (double '(a b c c d e) 'c)
NIL
> (double '() 'c)
NIL
```

Решение 4.346.4

```
(defun double (w g &aux (n (count g w)))
  (if (zerop n) w (when (= n 1) (d w g))))

(defun d (w g)
  (loop for a in w
        if (equalp a g) nconc (list a a)
        else nconc (list a)))

> (double '(a b c c d e) 'f)
(A B C C D E)
> (double '(a b c d e) 'c)
(A B C C D E)
> (double '(a b c c d e) 'c)
NIL
> (double '() 'c)
NIL
```

Решение 4.346.5

```
(defun double (w g &aux (n (count g w)))
  (if (zerop n) w (when (= n 1) (d w g))))

(defun d (w g)
  (mapcan #'(lambda (a) (if (equalp a g)
                           (list a a)
                           (list a)))
          w))
```

```

> (double '(a b c c d e) 'f)
(A B C C D E)
> (double '(a b c d e) 'c)
(A B C C D E)
> (double '(a b c c d e) 'c)
NIL
> (double '() 'c)
NIL

```

Решение 4.346.6

```

(defun double (w g &aux (n (count g w)))
  (if (zerop n) w (when (= n 1) (d w g))))

(defun d (w g)
  (mapcan #'(lambda (a) (if (equalp a g) `((a ,a) `(,a))) w))

> (double '(a b c c d e) 'f)
(A B C C D E)
> (double '(a b c d e) 'c)
(A B C C D E)
> (double '(a b c c d e) 'c)
NIL
> (double '() 'c)
NIL

```

Решение 4.346.7

```

(defun double (w g &aux (n (count g w)))
  (if (zerop n) w (when (= n 1) (d w g))))

(defun d (w g &aux (p (position g w)))
  (append (subseq w 0 (1+ p)) (subseq w p)))

> (double '(a b c c d e) 'f)
(A B C C D E)
> (double '(a b c d e) 'c)
(A B C C D E)
> (double '(a b c c d e) 'c)
NIL
> (double '() 'c)
NIL

```

Решение 4.346.8

```

(defun double (w g &aux (n (count g w)))
  (if (zerop n) w (when (= n 1) (d w g))))

(defun d (w g &aux (p (position g w)))
  (nconc (subseq w 0 (1+ p)) (nthcdr p w)))

> (double '(a b c c d e) 'f)

```

```
(A B C C D E)
> (double '(a b c d e) 'c)
(A B C C D E)
> (double '(a b c c d e) 'c)
NIL
> (double '() 'c)
NIL
```

Задача 4.347 double-third.lisp

Продублировать каждый третий элемент списка.

Решение 4.347.1

```
(defun double-third (w &optional (n 2))
  (cond ((null w) nil)
        ((zerop n) (cons (car w) (double-third w 3)))
        ((cons (car w) (double-third (cdr w) (1- n))))))

> (double-third '(1 2 3 4 5 6 7 8 9 10))
(1 2 3 3 4 5 6 6 7 8 9 9 10)
> (double-third '(1 2 3))
(1 2 3 3)
> (double-third '(1 2))
(1 2)
> (double-third '(1))
(1)
> (double-third '())
NIL
```

Задача 4.348 double-third-us.lisp

Продублировать каждый третий элемент списка *w*, если тот равен *u*.

Решение 4.348.1

```
(defun double-third-us (w u &optional (n 2) &aux (a (car w)))
  (cond ((null w) nil)
        ((and (zerop n) (equalp a u))
         (cons a (double-third-us w u 2)))
        ((zerop n) (cons a (double-third-us (cdr w) u 2)))
        ((cons a (double-third-us (cdr w) u (1- n))))))

> (double-third-us '(1 2 3 4 5 6 7 8 9 10) 3)
(1 2 3 3 4 5 6 7 8 9 10)
> (double-third-us '(1 2 3 4 5 6 7 8 9 10) 8)
(1 2 3 4 5 6 7 8 9 10)
> (double-third-us '(1 2 3 4 5 6 7 8 9 10) 9)
(1 2 3 4 5 6 7 8 9 9 10)
```

Решение 4.348.2

```

(defun double-third-us (w u &optional (n 2) &aux (a (car w)))
  (cond ((null w) nil)
        ((zerop n) (if (equalp a u)
                        (cons a (double-third-us w u 2))
                        (cons a (double-third-us (cdr w) u 2))))
        ((cons a (double-third-us (cdr w) u (1- n))))))

> (double-third-us '(1 2 3 4 5 6 7 8 9 10) 3)
(1 2 3 3 4 5 6 7 8 9 10)
> (double-third-us '(1 2 3 4 5 6 7 8 9 10) 8)
(1 2 3 4 5 6 7 8 9 10)
> (double-third-us '(1 2 3 4 5 6 7 8 9 10) 9)
(1 2 3 4 5 6 7 8 9 9 10)

```

Задача 4.349 *sqrt-them-all.lisp*

The people I've found most deeply knowledgeable about (Common) Lisp are those who have spent a lot of time working on a compiler. All of the following people have demonstrated great wisdom for the language. SBCL: Paul Khuong, Nikodemus Siivola, Christophe Rhodes, Stas Boukarev. CCL: Gary Byers. LW: Martin Simmons. Allegro: Duane Rettig. There are certainly lots more people beyond this list, but it's certainly a notable group.

Robert Smith

Продублировать каждый третий элемент списка *w*, если тот равен *u*.

Решение 4.349.1

```

(defun sqrt-them-all (w)
  (when w (cons (floor (newton (car w) .00005))
                 (sqrt-them-all (cdr w)))))

(defun newton (x e &optional (z 1))
  (if (> (abs (- x (* z z))) e)
      (newton x e (* (+ z (/ x z)) .5))
      z))

> (sqrt-them-all '(4 9 16 32))
(2.0 3.0 4.0000005 5.656854)

```

Решение 4.349.2

```

(defun sqrt-them-all (w &optional ac)
  (if w
      (sqrt-them-all (cdr w)
                      (cons (sqrt (car w))
                            ac))))

```

```

                                ac))
      (nreverse ac)))

> (sqrt-them-all '(4 9 16 25 49))
(2 3 4 5 7)

```

Решение 4.349.3

```

(defun sqrt-them-all (w &aux ac)
  (dolist (a (reverse w) ac)
    (setf ac (cons (sqrt a) ac))))

> (sqrt-them-all '(4 9 16 25 49))
(2 3 4 5 7)

```

Решение 4.349.4

```

(defun sqrt-them-all (w &aux ac)
  (dolist (a (reverse w) ac)
    (push (sqrt a) ac)))

> (sqrt-them-all '(4 9 16 25 49))
(2 3 4 5 7)

```

Решение 4.349.5

```

(defun sqrt-them-all (w)
  (loop for a in w collect (sqrt a)))

> (sqrt-them-all '(4 9 16 25 49))
(2 3 4 5 7)

```

Решение 4.349.6

```

(defun sqrt-them-all (w) (mapcar #'(lambda (a) (sqrt a)) w))

> (sqrt-them-all '(4 9 16 25 49))
(2 3 4 5 7)

```

Задача 4.350 *deep-reverse.lisp*

Определить функцию реверсирования произвольного списка (включая под-списки).

Решение 4.350.1

```

(defun deep-reverse (w &optional ac &aux (a (car w)) (d (cdr w)) )
  (cond ((null w) ac)
        ((atom a) (deep-reverse d (cons a ac)))
        ((deep-reverse d (cons (deep-reverse a) ac)))))

> (deep-reverse '(1 (2 3) 4))

```



```
(4 (3 2) 1)
```

Решение 4.350.2

```
(defun deep-reverse (w &optional ac &aux (a (car w)))
  (if w
      (deep-reverse (cdr w)
                     (cons (if (atom a)
                               a
                               (deep-reverse a))
                           ac))
      ac))

> (deep-reverse '(1 (2 3) 4))
(4 (3 2) 1)
```

Задача 4.351 change.lisp

В произвольном списке все отрицательные элементы умножить на -2, все 0 заменить на (1 1).

Решение 4.351.1

```
(defun change (w)
  (loop for a in w
        if (atom a)
        collect (cond ((minusp a) (* a -2))
                      ((zerop a) '(1 1))
                      (t a))
        else collect (change a)))

> (change '(-1 1 0 2 -3))
(2 1 (1 1) 2 6)
> (change '(-1 (-1 (-1 0 1) 1) 2))
(2 (2 (2 (1 1) 1) 1) 2)
```

Решение 4.351.2

```
(defun change (w)
  (loop for a in w
        collect (cond ((listp a) (change a))
                      ((minusp a) (* a -2))
                      ((zerop a) '(1 1))
                      (t a))))

> (change '(-1 1 0 2 -3))
(2 1 (1 1) 2 6)
> (change '(-1 (-1 (-1 0 1) 1) 2))
(2 (2 (2 (1 1) 1) 1) 2)
```



```
w))
```

```
> (intermittent '(1 2 3 4 5 6 7))
4
```

Задача 4.354 *percentage.lisp*

Задана матрица размера 5*5, состоящая из чисел от 0 до 9. Определить функцию, возвращающую их процентное содержание.

Решение 4.354.1

```
(defun percentage (w
                  &aux
                    (v (loop for a in w nconc a))
                    (n (length v)))
  (loop for a in (sort (remove-duplicates v) #'<)
        collect `(,a ,(* (float (/ (count a v) n)) 100))))

> (percentage '((1 2 3 4 5) (0 4 5 2 3) (3 3 4 4 5) (5 5 0 0 1) (4 5 2
3 1)))
((0 12.0) (1 12.0) (2 12.0) (3 20.0) (4 20.0) (5 24.0))
```

Задача 4.355 *main-sum.lisp*

Задана матрица A размера 6*7 в каждой строке матрицы элементу главной диагонали присвоить сумму положительных элементов строки без элемента главной диагонали.

Решение 4.355.1

```
(defun main-sum (w &optional (n 0) &aux (a (car w)))
  (when w (cons (insert a n (reduce
                        #'(+
                          (remove-if
                           #'minusp
                           (drop a n))))
                  (main-sum (cdr w) (1+ n)))))

(defun insert (w n z &optional(m 0))
  (cond ((null w) nil)
        ((= n m) (cons z (cdr w)))
        ((cons (car w) (insert (cdr w) n z (1+ m)))))

(defun drop (w n)
  (if (zerop n) (cdr w)
      (cons (car w) (drop (cdr w) (1- n)))))

> (main-sum '((-1 2 3) (-4 5 6) (-7 8 9)))
((5 2 3) (-4 6 6) (-7 8 8))
```

Задача 4.356 result.lisp

Сводка выполнения плана содержит сведения: наименование, шифр, единица измерения, план выпуска, фактически выпущено, отклонение от плана. Описать и заполнить запись для пяти позиций сводки. Предусмотреть возможность выдачи данных по введённому наименованию.

Решение 4.356.1

```
(defstruct sum name code unit plan impl devi)

(defun res (n)
  (when (> n 0) (cons (make-sum :name (read)
                                  :code (read)
                                  :unit (read)
                                  :plan (read)
                                  :impl (read)
                                  :devi (read))
                      (res (1- n)))))

(defun result (w name)
  (loop for a in w
        when (equalp (sum-name a) name) collect a))

> (setf summary (res 5))
wheat
1
ton
1000
1100
100
barley
2
ton
800
700
-100
rye
3
ton
400
400
0
maize
4
ton
500
600
100
rice
5
ton
```

```

2000
1500
-500
(#S(SUM :NAME WHEAT :CODE 1 :UNIT TON :PLAN 1000 :IMPL 1100 :DEVI 100)
 #S(SUM :NAME BARLEY :CODE 2 :UNIT TON :PLAN 800 :IMPL 700 :DEVI -100)
 #S(SUM :NAME RYE :CODE 3 :UNIT TON :PLAN 400 :IMPL 400 :DEVI 0)
 #S(SUM :NAME MAIZE :CODE 4 :UNIT TON :PLAN 500 :IMPL 600 :DEVI 100)
 #S(SUM :NAME RICE :CODE 5 :UNIT TON :PLAN 2000 :IMPL 1500 :DEVI -
500))
> (result summary 'rice)
(#S(SUM :NAME RICE :CODE 5 :UNIT TON :PLAN 2000 :IMPL 1500 :DEVI -
500))

```

Задача 4.357 max-span.lisp

In my opinion, the number one feature that makes Lisp attractive, is the uniform syntax. Everything looks the same, and there are very little things to remember about syntax. Because of the uniform syntax, manipulating programs in Lisp is very easy that it's a language feature. Using macros, you build an entirely new language that would suit your problem better. So when you're solving a new problem, you got two options. One option is to break down the problem so that it fits the language, which is what you normally do when you're using other languages. The second option is to mold Lisp to make it suit your problem better, this is not very easy in other languages.

Mostafa Hany Goma

Даны отрезки. Определите интервалы, которые покрывается наибольшим количеством отрезков и количество отрезков, покрывающих эти интервалы.

Решение 4.357.1

```

(defun max-span (w
  &aux
    (v (reduce #'append w))
    (n (reduce #'min v))
    (m (reduce #'max v))
    (x (loop for a in w collect (line n m a)))
    (z (apply #'mapcar #'list x))
    (p (pair z))
    (s (loop for a in p
      collect
        (mapcar #'list (car a) (cadr a))))
    (i (loop for a in s

```

```

                collect
                (loop for e in a
                      when (every #'numberp e)
                        collect e)))
            (r (reduce #'max (mapcar #'length i))))
(values (loop for a in i
              when (= (length a) r)
                collect (car a))
      r))

(defun line (n m w)
  (loop for a from n to m
        if (<= (first w) a (second w)) collect a
        else collect 'a))

(defun pair (w)
  (when (cdr w) (cons (list (car w) (cadr w))
                      (pair (cdr w)))))

> (max-span '((1 8) (1 9) (2 3) (2 4) (2 5) (6 7) (6 8) (6 9)))
((2 3) (6 7))
5

```

Задача 4.358 conditional-mid-edges-change.lisp

Определить функцию, которая меняет местами первый и последний элементы списка, оставляя остальные на своих местах, если элементы списка разные, иначе функция возвращает список без первого и последнего элементов.

Решение 4.358.1

```

(defun edges-equal (w)
  (equalp (car w) (car (last w))))

(defun conditional-mid-edges-change (w)
  (if (edges-equal w)
      (butlast (cdr w))
      (append (last w) (butlast (cdr w)) (list (car w)))))

> (conditional-mid-edges-change '(a b c d))
(D B C A)
> (conditional-mid-edges-change '(a b c a))
(B C)

```

Решение 4.358.2

```

(defun edges-equal (w)
  (equalp (car w) (car (last w))))

(defun conditional-mid-edges-change (w &aux (v (butlast (cdr w))))
  (if (edges-equal w) v

```

```

      (append (last w) v (list (car w))))))

> (conditional-mid-edges-change '(a b c d))
(D B C A)
> (conditional-mid-edges-change '(a b c a))
(B C)

Решение 4.358.3

(defun edges-equal (w)
  (equalp (car w) (car (last w))))

(defun rotate (w) (rotatef (car w) (nth (1- (length w)) w) w))

(defun conditional-mid-edges-change (w)
  (if (edges-equal w)
      (butlast (cdr w))
      (rotate w)))

> (conditional-mid-edges-change '(a b c d))
(D B C A)
> (conditional-mid-edges-change '(a b c a))
(B C)

```

Задача 4.359 pos+.lisp

Определить функцию, которая принимает одноуровневый список чисел и возвращает новый список чисел, основанный на исходном, в котором n -й элемент списка увеличен на номер позиции в списке (нумерация с нуля).

Решение 4.359.1

```

(defun pos+ (w &optional (n 0))
  (when w (cons (+ n (car w)) (pos+ (cdr w) (1+ n)))))

> (pos+ '(0 0 0 0))
(0 1 2 3)
> (pos+ '(1 2 3 4))
(1 3 5 7)
> (pos+ '())
NIL

```

Решение 4.359.2

```

(defun pos+ (w &optional ac (n 0))
  (if w
      (pos+ (cdr w) (cons (+ n (car w)) ac) (1+ n))
      (nreverse ac)))

> (pos+ '(0 0 0 0))
(0 1 2 3)
> (pos+ '(1 2 3 4))

```

```
(1 3 5 7)
> (pos+ '())
NIL
```

Решение 4.359.3

```
(defun pos+ (w)
  (loop for a in w for b from 0 collect (+ a b)))

> (pos+ '(0 0 0 0))
(0 1 2 3)
> (pos+ '(1 2 3 4))
(1 3 5 7)
> (pos+ '())
NIL
```

Задача 4.360 matrix-main.lisp

Дана квадратная матрица чисел. Найти в ней наименьший элемент в секторе над главной диагональю и наименьший элемент в секторе под главной диагональю. Наибольшим из них заменить элементы главной диагонали.

Решение 4.360.1

```
(defun matrix-main (w
  &optional
  (n 0)
  (m (max (main-diagonal-above-min w)
    (main-diagonal-below-min w)))
  &aux
  (a (car w)))
  (when w (cons (progn (setf (nth n a) m) a)
    (matrix-main (cdr w) (1+ n) m))))

(defun main-diagonal-above-min (w)
  (loop for a in w
    for b from 1 to (1- (length (car w)))
    minimize (apply #'min (subseq a b))))

(defun main-diagonal-below-min (w)
  (loop for a in (cdr w)
    for b upfrom 1
    minimize (apply #'min (subseq a 0 b))))

> (matrix-main '((1 2 3 4) (5 6 -7 8) (9 10 11 12) (13 -14 15 16)))
((-7 2 3 4) (5 -7 -7 8) (9 10 -7 12) (13 -14 15 -7))
> (matrix-main '((1 2 3 4) (5 6 7 8) (9 10 11 12) (13 -14 15 16)))
((2 2 3 4) (5 2 7 8) (9 10 2 12) (13 -14 15 2))
```

Решение 4.360.2

```
(defun matrix-main (w &optional
```



```

      (n 0)
      (m (max (uni-min w t)
              (uni-min (cdr w))))
      &aux (a (car w)))
    (when w (cons (progn (setf (nth n a) m) a)
                  (matrix-main (cdr w) (1+ n) m))))

(defun uni-min (w &optional f)
  (loop for a in w
        for b from 1 to (1- (length (car w)))
        minimize (reduce #'min (if f
                                     (subseq a b)
                                     (subseq a 0 b))))))

> (matrix-main '((1 2 3 4) (5 6 -7 8) (9 10 11 12) (13 -14 15 16)))
((-7 2 3 4) (5 -7 -7 8) (9 10 -7 12) (13 -14 15 -7))
> (matrix-main '((1 2 3 4) (5 6 7 8) (9 10 11 12) (13 -14 15 16)))
((2 2 3 4) (5 2 7 8) (9 10 2 12) (13 -14 15 2))

```

Задача 4.361 *quine.lisp*

Написать функцию, возвращающую точную копию тела функции.

Решение 4.361.1

```

(defun quine ()
  ((lambda (x) (list x (list 'quote x)))
   '(lambda (x) (list x (list 'quote x)))))

> (quine)
((LAMBDA (X) (LIST X (LIST 'QUOTE X))) '(LAMBDA (X) (LIST X (LIST
'QUOTE X))))
> ((lambda (x) (list x (list 'quote x)))
   '(lambda (x) (list x (list 'quote x)))))
((LAMBDA (X) (LIST X (LIST 'QUOTE X))) '(LAMBDA (X) (LIST X (LIST
'QUOTE X))))

```

Задача 4.362 *sum-square.lisp*

Определить функцию, возвращающую сумму квадратов числового списка.

Решение 4.362.1

```

(defun sum-square (w)
  (if w (+ (expt (car w) 2) (sum-square (cdr w))) 0))

> (sum-square '(1 2 3 4 5 6 7 8 9 10))
385

```

Решение 4.362.2

```
(defun sum-square (w)
  (loop for a in w sum (expt a 2)))

> (sum-square '(1 2 3 4 5 6 7 8 9 10))
385
```

Задача 4.363 first-min.lisp

Дан список чисел. Определить функцию, меняющую первое число с первым наименьшим числом в этом списке.

Решение 4.363.1

```
(defun first-min (w
                  &optional (z (car w)) (m (min-number w)) f
                  &aux (a (car w)))
  (cond ((null w) nil)
        ((= a m) (cons z (cdr w)))
        (f (cons a (first-min (cdr w) z m t)))
        ((cons m (first-min (cdr w) z m t)))))

(defun min-number (w &optional x &aux (a (car w)))
  (if w (min-number (cdr w) (if (and x (<= x a)) x a)) x))

> (first-min '(2 2 3 4 1 1))
(1 2 3 4 2 1)
```

Решение 4.363.2

```
(defun first-min (w
                  &optional (z (car w)) (m (reduce #'min w)) f
                  &aux (a (car w)))
  (cond ((null w) nil)
        ((= a m) (cons z (cdr w)))
        (f (cons a (first-min (cdr w) z m t)))
        ((cons m (first-min (cdr w) z m t)))))

> (first-min '(2 2 3 4 1 1))
(1 2 3 4 2 1)
```

Решение 4.363.3

```
(defun first-min (w
                  &optional (z (car w)) (m (reduce #'min w)) f
                  &aux (a (car w)))
  (cond ((null w) nil)
        ((= a m) (cons z (cdr w)))
        ((cons (if f a m) (first-min (cdr w) z m t)))))

> (first-min '(2 2 3 4 1 1))
(1 2 3 4 2 1)
```

Решение 4.363.4

```
(defun first-min (w
                  &optional (z (car w)) (m (reduce #'min w)) f
                  &aux (a (car w)))
  (when w (if (= a m) (cons z (cdr w))
              (cons (if f a m) (first-min (cdr w) z m t)))))

> (first-min '(2 2 3 4 1 1))
(1 2 3 4 2 1)
```

Задача 4.364 first-min-all.lisp

Дан список чисел. Определить функцию, меняющую первое число с наименьшими числами в этом списке.

Решение 4.364.1

```
(defun first-min-all (w
                      &optional (z (car w)) (m (reduce #'min w)) f
                      &aux (a (car w)))
  (cond ((null w) nil)
        ((= a m) (cons z (first-min-all (cdr w) z m t)))
        (f (cons a (first-min-all (cdr w) z m t)))
        ((cons m (first-min-all (cdr w) z m t)))))

> (first-min-all '(2 2 3 4 1 1))
(1 2 3 4 2 2)
```

Решение 4.364.2

```
(defun first-min-all (w
                      &optional (z (car w)) (m (reduce #'min w)) f
                      &aux (a (car w)))
  (when w (cons (cond ((= a m) z)
                  (f a)
                  (m))
                (first-min-all (cdr w) z m t))))

> (first-min-all '(2 2 3 4 1 1))
(1 2 3 4 2 2)
```

Задача 4.365 ns.lisp

Дан список из чисел и символов. Определить функцию, которая делит список на два подсписка так, чтобы в первый список вошли только числа, а во второй – только символы.

Решение 4.365.1

```
(defun ns (w &optional n s &aux (a (car w)))
```

```

(cond ((null w) (list (nreverse n) (nreverse s)))
      ((numberp a) (ns (cdr w) (cons a n) s))
      ((ns (cdr w) n (cons a s)))))

> (ns '(a 1 b 2 c 3 d 4))
((1 2 3 4) (A B C D))

```

Задача 4.366 *plural.lisp*

Множественное число большинства английских существительных получается путем добавления буквы *s* к форме единственного числа. Но если существительное заканчивается буквой *y*, следующей за согласной, множественное число образуется путем замены буквы *y* на сочетание *ies*: если же существительное заканчивается буквой *o*, следующей за согласной, множественное число образуется путем добавления сочетания *es*. Определите функцию *(f w)*, вычисляющую множественное число для английского слова, представленного списком букв *w*.

Решение 4.366.1

```

(defparameter *consonant* '(#\b #\c #\d #\f #\g #\h #\j #\k #\l #\m
#\n #\p #\q #\r #\s #\t #\v #\w #\x #\y #\z))

(defun plural (w)
  (cond ((null w) nil)
        ((and (char= (car (last w)) #\y)
               (member (car (last w 2)) *consonant*))
         (nconc (butlast w) '(#\i #\e #\s)))
        ((and (char= (car (last w)) #\o)
               (member (car (last w 2)) *consonant*))
         (nconc w '(#\e #\s)))
        ((nconc w '(#\s)))))

> (plural '(\g #\u #\n))
(#\g #\u #\n #\s)
> (plural '(\h #\e #\r #\o))
(#\h #\e #\r #\o #\e #\s)
> (plural '(\s #\k #\y))
(#\s #\k #\i #\e #\s)

```

Задача 4.367 *inside-out.lisp*

Определить функцию, разбиения простых подсписков списка на атомы и превращения атомов списка в списки.

Решение 4.367.1

```

(defun inside-out (w)
  (loop for a in w
        if (atom a) collect (list a)
        else if (some #'listp a) collect a

```

```
else nconc a))
```

```
> (inside-out '((a b) (c (d e)) f))
(A B (C (D E)) (F))
> (inside-out '(a (b (c) d) e (((f)))))
((A) (B (C) D) (E) (((F))))
```

Решение 4.367.2

```
(defun inside-out (w &aux (a (car w)))
  (cond ((null w) nil)
        ((atom a) (cons (list a) (inside-out (cdr w))))
        ((some #'listp a) (cons a (inside-out (cdr w))))
        ((nconc a (inside-out (cdr w)))))

> (inside-out '((a b) (c (d e)) f))
(A B (C (D E)) (F))
> (inside-out '(a (b (c) d) e (((f)))))
((A) (B (C) D) (E) (((F))))
```

Решение 4.367.3

```
(defun inside-out (w &aux (a (car w)))
  (when w
    (if (and (listp a) (every #'atom a))
        (nconc a (inside-out (cdr w)))
        (cons (if (atom a) (list a) a)
                (inside-out (cdr w))))))

> (inside-out '((a b) (c (d e)) f))
(A B (C (D E)) (F))
> (inside-out '(a (b (c) d) e (((f)))))
((A) (B (C) D) (E) (((F))))
```

Решение 4.367.4

```
(defun inside-out (w)
  (loop for a in w
        if (and (listp a) (every #'atom a)) nconc a
        else collect (if (atom a) (list a) a)))

> (inside-out '((a b) (c (d e)) f))
(A B (C (D E)) (F))
> (inside-out '(a (b (c) d) e (((f)))))
((A) (B (C) D) (E) (((F))))
```

Задача 4.368 max-frequent.lisp

В целочисленном списке найти числа, которые повторяются максимальное количество раз.

Решение 4.368.1

```
(defun max-frequent (w
  &aux
    (v (remove-duplicates w))
    (z (loop for a in v
      collect (list (count a w) a)))
    (m (reduce #'max z :key #'car)))
  (loop for a in z when (= (car a) m) collect (cadr a)))

> (max-frequent '(1 2 3 1 2))
(1 2)
```

Задача 4.369 max-product-3.lisp

Дан список целых чисел. Определить функцию, возвращающую три числа из этого списка, произведение которых является наибольшим.

Решение 4.369.1

```
(defun max-product-3 (w
  &aux
    (n1 (first w))
    (n2 (second w))
    (n3 (third w))
    (n4 (fourth w))
    (n5 (fifth w)))
  (cond ((null n3) 'less-than-3-items)
        ((null n4) (list n1 n2 n3))
        ((null n5) (let* ((x1 (* n1 n2 n3))
                          (x2 (* n1 n3 n4))
                          (x3 (* n1 n2 n4))
                          (x4 (* n2 n3 n4))
                          (mx (max x1 x2 x3 x4)))
          (cond ((= x1 mx) (list n1 n2 n3))
                ((= x2 mx) (list n1 n3 n4))
                ((= x3 mx) (list n1 n2 n4))
                ((= x4 mx) (list n2 n3 n4))))))
  (t (let ((v (sort (list n1 n2 n3 n4 n5) #>)))
    (max-product-3- (nthcdr 5 w)
      (first v)
      (second v)
      (third v)
      (fourth v)
      (fifth v))))))

(defun max-product-3- (w m1 m2 m3 m4 m5
  &aux
    (a (car w))
    (d (cdr w)))
  (cond ((null w) (if (>= (* m1 m2 m3) (* m1 m4 m5))
    (list m1 m2 m3)
    (list m1 m4 m5)))
```

```

((> a m1) (max-product-3- d a m1 m2 m4 m5))
((> a m2) (max-product-3- d m1 a m2 m4 m5))
((> a m3) (max-product-3- d m1 m2 a m4 m5))
((< a m4) (max-product-3- d m1 m2 m3 a m4))
((< a m5) (max-product-3- d m1 m2 m3 m4 a))
(t (max-product-3- d m1 m2 m3 m4 m5)))

> (max-product-3 '(-10 -10 1 3 2))
(3 -10 -10)
> (max-product-3 '(-1 -2 3 4))
(-1 -2 4)
> (max-product-3 '(-1 -2 3 4 5))
(5 4 3)
> (max-product-3 '(-1 -2))
LESS-THAN-3-ITEMS

```

Решение 4.369.2

```

(defun max-product-3 (w
  &aux
    (n1 (first w))
    (n2 (second w))
    (n3 (third w))
    (n4 (fourth w))
    (n5 (fifth w)))
  (cond ((null n3) 'less-than-3-items)
        ((null n4) (list n1 n2 n3))
        ((null n5) (let* ((x1 (* n1 n2 n3))
                          (x2 (* n1 n3 n4))
                          (x3 (* n1 n2 n4))
                          (x4 (* n2 n3 n4))
                          (mx (max x1 x2 x3 x4)))
                    (cond ((= x1 mx) (list n1 n2 n3))
                          ((= x2 mx) (list n1 n3 n4))
                          ((= x3 mx) (list n1 n2 n4))
                          ((= x4 mx) (list n2 n3 n4))))))
  (t (let ((v (sort (list n1 n2 n3 n4 n5) #'>)))
      (max-product-3- (nthcdr 5 w)
        (first v)
        (second v)
        (third v)
        (fourth v)
        (fifth v))))))

(defun max-product-3- (w m1 m2 m3 m4 m5)
  (loop for a in w do
    (cond ((> a m1) (setf m3 m2) (setf m2 m1) (setf m1 a))
          ((> a m2) (setf m3 m2) (setf m2 a))
          ((> a m3) (setf m3 a))
          ((< a m4) (setf m5 m4) (setf m4 a))
          ((< a m5) (setf m5 a)))
    finally (return (if (>= (* m1 m2 m3) (* m1 m4 m5))
      (list m1 m2 m3)

```

```

                                (list m1 m4 m5))))))
> (max-product-3 '(-10 -10 1 3 2))
(3 -10 -10)
> (max-product-3 '(-1 -2 3 4))
(-1 -2 4)
> (max-product-3 '(-1 -2 3 4 5))
(5 4 3)
> (max-product-3 '(-1 -2))
LESS-THAN-3-ITEMS

```

Задача 4.370 `_remove-duplicates.lisp`

Most papers in computer science describe how their author learned what someone else already knew.

Peter Landin

Определите функцию, удаляющую повторные вхождения элементов в список.

Решение 4.370.1

```

(defun _remove-duplicates (w)
  (cond ((null w) nil)
        ((member (car w) (cdr w)) (_remove-duplicates (cdr w)))
        ((cons (car w) (_remove-duplicates (cdr w))))))

> (_remove-duplicates '(1 1 1 a a))
(1 A)

```

Решение 4.370.2

```

(defun _remove-duplicates (w)
  (when w (if (member (car w) (cdr w))
              (_remove-duplicates (cdr w))
              (cons (car w) (_remove-duplicates (cdr w))))))

> (_remove-duplicates '(1 1 1 a a))
(1 A)

```

Решение 4.370.3

```

(defun _remove-duplicates (w &aux (a (car w)) (d (cdr w)))
  (when w (if (member a d)
              (_remove-duplicates d)
              (cons a (_remove-duplicates d)))))

> (_remove-duplicates '(1 1 1 a a))
(1 A)

```

Решение 4.370.4


```
(defun _remove-duplicates (w)
  (loop for a in w
        unless (member a ac)
        collect a into ac
        finally (return (nreverse ac))))
```

```
> (_remove-duplicates '(1 1 1 a a))
(1 A)
```

Решение 4.370.5

```
(defun _remove-duplicates (w &aux ac)
  (loop for a in w unless (member a ac) do (push a ac)
        finally (return (nreverse ac))))
```

```
> (_remove-duplicates '(1 1 1 a a))
(1 A)
```

Решение 4.370.6

```
(defun _remove-duplicates (w &aux ac)
  (loop for a in w do (pushnew a ac)
        finally (return (nreverse ac))))
```

```
> (_remove-duplicates '(1 1 1 a a))
(1 A)
```

Решение 4.370.7

```
(defun _remove-duplicates (w &optional ac &aux (a (car w)))
  (if w
      (_remove-duplicates (cdr w)
                           (if (member a ac)
                               ac
                               (cons a ac)))
      (nreverse ac)))
```

```
> (_remove-duplicates '(1 1 1 a a 1))
(1 A)
```

Решение 4.370.8

```
(defun _remove-duplicates (w
                           &optional ac
                           &aux (a (car w)) (d (cdr w)))
  (if w
      (_remove-duplicates d
                           (if (member a d)
                               ac
                               (cons a ac)))
      (nreverse ac)))
```

```
> (_remove-duplicates '(1 1 1 a a 1))
```

(A 1)

Задача 4.371 max-days-off-month.lisp

Выходные и праздничные дни заданы списком пар (день, месяц). Определить, в каком месяце больше всего нерабочих дней.

Решение 4.371.1

```
(defun max-days-off-month (w
  &aux
  (v (loop for a in w
    collect (cadr a)))
  (z (loop for a in (remove-duplicates v)
    collect (list a (count a v))))
  (m (loop for a in z maximize (cadr a))))
  (remove-if-not #'(lambda (a) (= (cadr a) m)) z))

> (max-days-off-month '((1 1) (2 1) (3 1) (23 2) (8 3) (1 5) (2 5) (9
5)))
((1 3) (5 3))
```

Решение 4.371.2

```
(defun max-days-off-month (w
  &aux
  (v (loop for a in w
    collect (cadr a)))
  (z (loop for a in (remove-duplicates v)
    collect (list a (count a v))))
  (m (loop for a in z maximize (cadr a))))
  (loop for a in (remove-if-not #'(lambda (a) (= (cadr a) m)) z)
    collect (car a)))

> (max-days-off-month '((1 1) (2 1) (3 1) (23 2) (8 3) (1 5) (2 5) (9
5)))
(1 5)
```

Решение 4.371.3

```
(defun max-days-off-month (w
  &aux
  (v (mapcar #'cadr w))
  (z (loop for a in (remove-duplicates v)
    collect (list a (count a v))))
  (m (loop for a in z maximize (cadr a))))
  (mapcar #'car (loop for a in z when (= (cadr a) m) collect a)))

> (max-days-off-month '((1 1) (2 1) (3 1) (23 2) (8 3) (1 5) (2 5) (9
5)))
(1 5)
```

Решение 4.371.4

```
(defun max-days-off-month (w
                          &aux
                          (v (mapcar #'cadr w))
                          (z (loop for a in (remove-duplicates v)
                                collect (list a (count a v))))
                          (m (loop for a in z maximize (cadr a))))
  (loop for a in z when (= (cadr a) m) collect (car a)))

> (max-days-off-month '((1 1) (2 1) (3 1) (23 2) (8 3) (1 5) (2 5) (9
5)))
(1 5)
```

Решение 4.371.5

```
(defun max-days-off-month (w &aux (v (mapcar #'cadr w))
                          (z (loop for a in
                                (remove-duplicates v)
                                collect `(,a ,(count a v)))))
  (loop for a in z
        when (= (cadr a) (loop for a in z maximize (cadr a)))
        collect (car a)))

> (max-days-off-month '((1 1) (2 1) (3 1) (23 2) (8 3) (1 5) (2 5) (9
5)))
(1 5)
```

Задача 4.372 edge-backward-pair.lisp

Дан список '(a b c d e f g h). Определить функцию, возвращающую ((h a) (g b) (f c) (e d)).

Решение 4.372.1

```
(defun edge-backward-pair (w)
  (when w (cons (list (car (last w)) (car w))
                (edge-backward-pair (butlast (cdr w))))))

> (edge-backward-pair '(a b c d e f g h))
((H A) (G B) (F C) (E D))
```

Задача 4.373 shift-pair.lisp

Определить функцию, которая циклически сдвигает список w на 2 позиции вправо и группирует по 2 элемента в подписки.

Решение 4.373.1

```
(defun <r (w n) (append (last w n) (butlast w n)))
```

```

(defun pair (lst)
  (if (null lst) nil
      (cons (list (car lst) (cadr lst))
            (pair (cddr lst)))))

(defun shift-pair (w n) (pair (<r w n)))

> (shift-pair '(a b c d e f g h) 2)
((G H) (A B) (C D) (E F))

```

Решение 4.373.2

```

(defun <r (w n) (nconc (last w n) (butlast w n)))

(defun pair (w)
  (when w (cons `((, (car w) , (cadr w)) (pair (cddr w)))))

(defun shift-pair (w n) (pair (<r w n)))

> (shift-pair '(a b c d e f g h) 2)
((G H) (A B) (C D) (E F))

```

Решение 4.373.3

```

(defun <r (w n) (nconc (last w n) (butlast w n)))

(defun pair (w)
  (when w (cons (subseq w 0 2) (pair (cddr w)))))

(defun shift-pair (w n) (pair (<r w n)))

> (shift-pair '(a b c d e f g h) 2)
((G H) (A B) (C D) (E F))

```

Решение 4.373.4

```

(defun shift-pair (w n)
  (labels ((pair (v)
            (when v (cons (subseq v 0 2)
                          (pair (cddr v))))))
    (pair (<r w n))))

> (shift-pair '(a b c d e f g h) 2)
((G H) (A B) (C D) (E F))

```

Решение 4.373.5

```

(defun shift-pair (w n)
  (labels ((pair (v)
            (when v (cons (list (car v) (cadr v))
                          (pair (cddr v)))))
    (pair (<r w n))))

```

```
> (shift-pair '(a b c d e f g h) 2)
((G H) (A B) (C D) (E F))
```

Задача 4.374 edge-sum.lisp

Определить функцию, формирующую список, состоящий из сумм первого и последнего, второго и предпоследнего элементов числового списка и т. д. Каждый элемент должен участвовать в сложении не более одного раза.

Решение 4.374.1

```
(defun edge-sum (w
                  &aux
                    (m (ceiling (length w) 2))
                    (v (subseq w 0 m))
                    (z (subseq w m)))
  (sum v (reverse z)))

(defun sum (w v &aux (a (car v)))
  (when w (cons (+ (car w) (if a a 0))
                (sum (cdr w) (cdr v)))))

> (edge-sum '(1 -2 -3 4 5 6 -7 8 9))
(10 6 -10 10 5)
> (edge-sum '(1 -2 -3 4 5 100 6 -7 8 9))
(10 6 -10 10 105)
```

Решение 4.374.2

```
(defun edge-sum (w &aux (m (ceiling (length w) 2)))
  (sum (subseq w 0 m) (reverse (subseq w m))))

(defun sum (w v &aux (a (car v)))
  (when w (cons (+ (car w) (if a a 0))
                (sum (cdr w) (cdr v)))))

> (edge-sum '(1 -2 -3 4 5 6 -7 8 9))
(10 6 -10 10 5)
> (edge-sum '(1 -2 -3 4 5 100 6 -7 8 9))
(10 6 -10 10 105)
```

Решение 4.374.3

```
(defun edge-sum (w &aux (m (ceiling (length w) 2)))
  (labels ((sum (z v &aux (a (car v)))
            (when z (cons (+ (car z) (if a a 0))
                          (sum (cdr z) (cdr v))))))
    (sum (subseq w 0 m) (reverse (subseq w m)))))

> (edge-sum '(1 -2 -3 4 5 6 -7 8 9))
(10 6 -10 10 5)
> (edge-sum '(1 -2 -3 4 5 100 6 -7 8 9))
```

```
(10 6 -10 10 105)
```

Задача 4.375 twone.lisp

Определить функцию, заменяющую в исходном списке два подряд идущих одинаковых элемента одним.

Решение 4.375.1

```
(defun twone (w &aux (a (car w)) (d (cdr w)) )
  (when w (if (equalp a (car d))
              (cons a (twone (cdr d)))
              (cons a (twone d)))))
```

```
> (twone '(a a a b b n n n n))
(A A B N N)
```

Решение 4.375.2

```
(defun twone (w &aux (a (car w)) (d (cdr w)) )
  (cond ((null w) nil)
        ((equalp a (car d)) (cons a (twone (cdr d))))
        ((cons a (twone d)))))
```

```
> (twone '(a a a b b n n n n))
(A A B N N)
```

Решение 4.375.3

```
(defun twone (w &aux (a (car w)) (d (cdr w)) )
  (when w (cons a (twone (if (equalp a (car d)) (cdr d) d)))))
```

```
> (twone '(a a a b b n n n n))
(A A B N N)
```

Решение 4.375.4

```
(defun twone (w &optional ac &aux (a (car w)) (d (cdr w)) )
  (if w (twone (if (equalp a (car d)) (cdr d) d) (cons a ac))
      (nreverse ac)))
```

```
> (twone '(a a a b b n n n n))
(A A B N N)
```

Решение 4.375.5

```
(defun twone (w)
  (when w ((lambda (a d)
              (cons a (twone (if (equalp a (car d)) (cdr d) d)))
            (car w) (cdr w)))))
```

```
> (twone '(a a a b b n n n n))
```

```
(A A B N N)
```

Решение 4.375.6

```
(defun twone (w)
  (labels ((won (v ac &aux (a (car v)) (d (cdr v)))
            (if v (won (if (equalp a (car d)) (cdr d) d) (cons a ac))
                (nreverse ac))))
    (won w nil)))

> (twone '(a a a b b n n n n))
(A A B N N)
```

Задача 4.376 negative-zero.lisp

Определить функцию, заменяющую все отрицательные элементы списка значением 0.

Решение 4.376.1

```
(defun negative-zero (w &aux (a (car w)))
  (cond ((null w) nil)
        ((minusp a) (cons 0 (negative-zero (cdr w))))
        ((cons a (negative-zero (cdr w)))))

> (negative-zero '(-1 20 -3 40))
(0 20 0 40)
```

Решение 4.376.2

```
(defun negative-zero (w &aux (a (car w)))
  (when w (cons (if (minusp a) 0 a) (negative-zero (cdr w)))))

> (negative-zero '(-1 20 -3 40))
(0 20 0 40)
```

Решение 4.376.3

```
(defun negative-zero (w)
  (loop for a in w
        collect (if (minusp a) 0 a)))

> (negative-zero '(-1 20 -3 40))
(0 20 0 40)
```

Решение 4.376.4

```
(defun negative-zero (w) (substitute-if 0 #'minusp w))

> (negative-zero '(-1 20 -3 40))
(0 20 0 40)
```

Задача 4.377 drop-middle.lisp

Задан список длина которого кратная трем. Получить список в котором удалена вторая треть.

Решение 4.377.1

```
(defun drop-middle (w
                   &aux
                     (m (length w))
                     (n (/ m 3))
                     (z (+ n n 1)))
  (loop for a in w
        for b from 1 to m
        unless (> z b n)
        collect a))

> (drop-middle '(1 2 3 4 5 6 7 8 9))
(1 2 3 7 8 9)
```

Задача 4.378 ppos.lisp

Дан список чисел. Сформировать список, содержащий номера позиций положительных элементов исходного списка.

Решение 4.378.1

```
(defun ppos (w &optional (n 0))
  (cond ((null w) nil)
        ((plusp (car w)) (cons n (ppos (cdr w) (1+ n))))
        ((ppos (cdr w) (1+ n)))))

> (ppos '(-1 1 1 -1))
(1 2)
```

Решение 4.378.2

```
(defun ppos (w &optional (n 0))
  (when w (if (plusp (car w)) (cons n (ppos (cdr w) (1+ n)))
              (ppos (cdr w) (1+ n)))))

> (ppos '(-1 1 1 -1))
(1 2)
```

Решение 4.378.3

```
(defun ppos (w)
  (loop for a in w
        for n from 0
        when (plusp a) collect n))

> (ppos '(-1 1 1 -1))
```


(1 2)

Задача 4.379 group-same.lisp

Определить функцию, которая группирует элементы списка (a a a c b b a c c b) в подписки следующим образом: ((a a a) (c) (b b) (a) (c c) (b)).

Решение 4.379.1

```
(defun group-same (w &optional ac acc &aux (a (car w)))
  (cond ((null w) (when acc (reverse (cons ac acc)))))
  ((equal a (car ac)) (group-same (cdr w) (cons a ac) acc))
  (ac (group-same (cdr w) (list a) (cons ac acc)))
  ((group-same (cdr w) (cons a ac) acc))))

> (group-same '(a a a c b b a c c b))
((A A A) (C) (B B) (A) (C C) (B))
```

Решение 4.379.2

```
(defun group-same (w)
  (labels ((g (w ac acc &aux (a (car w)))
    (cond ((null w) (when acc (reverse (cons ac acc)))))
    ((equal a (car ac)) (g (cdr w) (cons a ac) acc))
    (ac (g (cdr w) (list a) (cons ac acc)))
    ((g (cdr w) (cons a ac) acc)))))
    (g w nil nil)))

> (group-same '(a a a c b b a c c b))
((A A A) (C) (B B) (A) (C C) (B))
```

Задача 4.380 eval-select.lisp

Определить функцию, которая обрабатывает список списков так, что при этом вычисляются только те подписки, которые начинаются с символа * или +.

Решение 4.380.1

```
(defun eval-select (w &aux (a (car w)))
  (cond ((null w) nil)
        ((and (listp a) (or (eq (car a) '*) (eq (car a) '+)))
         (cons (eval a) (eval-select (cdr w)))))
        ((cons a (eval-select (cdr w)))))

> (eval-select '(1 (+ 2 3) (+) (* 4 5) (*) '(6 7 8)))
(1 5 0 20 1 '(6 7 8))
```

Решение 4.380.2

```
(defun eval-select (w &aux (a (car w)))
  (when w
    (cons (if (and (listp a)
                  (or (eq (car a) '*)
                      (eq (car a) '+)))
          (eval a)
          a)
          (eval-select (cdr w))))))

> (eval-select '(1 (+ 2 3) (+) (* 4 5) (*) '(6 7 8)))
(1 5 0 20 1 '(6 7 8))
```

Решение 4.380.3

```
(defun eval-select (w)
  (loop for a in w collect
    (if (and (listp a)
            (or (eq (car a) '*)
                (eq (car a) '+)))
        (eval a)
        a)))

> (eval-select '(1 (+ 2 3) (+) (* 4 5) (*) '(6 7 8)))
(1 5 0 20 1 '(6 7 8))
```

Задача 4.381 sum.lisp

Определить функцию, которая возвращает список сумм подсписков многоуровневого списка, например (f '((2 5 7 (8 5 2 (1 2))))) => (14 (15 (3))).

Решение 4.381.1

```
(defun sum (w &optional (ac 0) acc &aux (a (car w)))
  (cond ((null w) (cons ac (nreverse acc)))
        ((atom a) (sum (cdr w) (+ a ac) acc))
        ((sum (cdr w) ac (cons (sum a) acc)))))

> (sum '(1 2 3 (4 5 6) 7 (4 5) 8 9 (8 9)))
(30 (15) (9) (17))
```

Решение 4.381.2

```
(defun sum (w)
  (cons (loop for a in w
    when (atom a) sum a
    (loop for a in w
      when (listp a) collect (sum a))))

> (sum '(1 2 3 (4 5 6) 7 (4 5) 8 9 (8 9)))
(30 (15) (9) (17))
```

Задача 4.382 level.lisp

Дан многоуровневый список. Определить функцию, возвращающую список, содержащий информацию о количестве подписков на каждом уровне вложенности ((<уровень> <количество подписков>)...).

Решение 4.382.1

```
(defun dive-count (n w &aux (a (car w)) (d (cdr w)))
  (cond ((null w) 0)
        ((= n 0) (count-if #'listp w))
        ((atom a) (dive-count n d))
        ((+ (dive-count (1- n) a) (dive-count n d)))))

(defun max-depth (w)
  (if (atom w)
      -1
      (max (1+ (max-depth (car w)))
            (max-depth (cdr w)))))

(defun level (w &aux (n (1- (max-depth w))))
  (loop for a from 0 to n
        collect (list a (dive-count a w))))

> (level '(a b (c d) ((e f ((g h) (i j) k)))))
((0 2) (1 1) (2 1) (3 2))
```

Решение 4.382.2

```
(defun dive-count (n w &aux (a (car w)) (d (cdr w)))
  (cond ((null w) 0)
        ((= n 0) (count-if #'listp w))
        ((atom a) (dive-count n d))
        ((+ (dive-count (1- n) a) (dive-count n d)))))

(defun max-depth (w)
  (if (atom w)
      -1
      (max (1+ (max-depth (car w)))
            (max-depth (cdr w)))))

(defun level (w &optional (n (max-depth w)) (m 0))
  (when (< m n) (cons (list m (dive-count m w))
                       (level w n (1+ m)))))

> (level '(a b (c d) ((e f ((g h) (i j) k)))))
((0 2) (1 1) (2 1) (3 2))
```

Решение 4.382.3

```
(defun dive-count (n w &aux (a (car w)) (d (cdr w)))
  (cond ((null w) 0)
```

```

      ((= n 0) (count-if #'listp w))
      ((atom a) (dive-count n d))
      ((+ (dive-count (1- n) a) (dive-count n d))))))

(defun max-depth (w)
  (if (atom w)
      -1
      (max (1+ (max-depth (car w)))
            (max-depth (cdr w)))))

(defun level (w &optional ac (n (max-depth w)) (m 0))
  (if (< m n)
      (level w
              (cons (list m (dive-count m w))
                    ac)
              n
              (1+ m))
      (reverse ac)))

> (level '(a b (c d) ((e f ((g h) (i j) k)))))
((0 2) (1 1) (2 1) (3 2))

```

Решение 4.382.4

```

(defun level (w
              &optional
              ac
              (n (labels
                  ((max-depth (v)
                     (if (atom v)
                         -1
                         (max (1+ (max-depth (car v)))
                              (max-depth (cdr v))))))
                    (max-depth w)))
              (m 0))
  (if (< m n)
      (level w
              (cons (list m
                          (labels
                           ((dive-count (n
                                           v
                                           &aux
                                           (a (car v))
                                           (d (cdr v)))
                           (cond ((null v) 0)
                                ((= n 0) (count-if #'listp v))
                                ((atom a) (dive-count n d))
                                ((+ (dive-count (1- n) a)
                                    (dive-count n d))))))
                          (dive-count m w)))
                    ac)
              n
              (1+ m))

```

```
(reverse ac)))

> (level '(a b (c d) ((e f ((g h) (i j) k)))))
((0 2) (1 1) (2 1) (3 2))
```

Задача 4.383 list-v.lisp

Определите функцию, которая из данного одноуровневого списка строит список списков его элементов, например, (a b) -> ((a) (b)).

Решение 4.383.1

```
(defun list-v (w) (when w (cons (list (car w)) (list-v (cdr w)))))

> (list-v '(a b c))
((A) (B) (C))
```

Решение 4.383.2

```
(defun list-v (w) (when w (cons `',(car w) (list-v (cdr w)))))

> (list-v '(a b c))
((A) (B) (C))
```

Решение 4.383.3

```
(defun list-v (w) (loop for a in w collect (list a)))

> (list-v '(a b c))
((A) (B) (C))
```

Решение 4.383.4

```
(defun list-v (w) (loop for a in w collect `(,a)))

> (list-v '(a b c))
((A) (B) (C))
```

Задача 4.384 next.lisp

Определить функцию переставляющую элементы списка таким образом, чтобы одинаковые элементы оказались рядом. Например, (1 2 1 4 3 1 2 4) -> (1 1 1 2 2 4 4 3).

Решение 4.384.1

```
(defun next (w &aux (a (car w)))
  (when w (nconc (remove-if-not #'(lambda (e) (equal a e)) w)
    (next (remove a w)))))

> (next '(1 2 1 4 3 1 2 4))
```

```
(1 1 1 2 2 4 4 3)
```

Решение 4.384.2

```
(defun next (w &aux (a (car w)))
  (when w (nconc (nx w a when) (next (nx w a unless)))))

(defmacro nx (w a p)
  `(loop for e in ,w ,p (equal ,a e) collect e))

> (next '(1 2 1 4 3 1 2 4))
(1 1 1 2 2 4 4 3)
```

Решение 4.384.3

```
(defun next (w
  &aux (a (car w))
  (z (loop for e in w
    if (equal a e) collect e into v
    else collect e into z
    finally (return (list v z)))
  (v (car z))
  (u (cadr z)))
  (when w (nconc v (next u))))

> (next '(1 2 1 4 3 1 2 4))
(1 1 1 2 2 4 4 3)
```

Решение 4.384.4

```
(defun next (w
  &aux
  (z (loop for a in w
    if (equal (car w) a) collect a into v
    else collect a into u
    finally (return (list v u)))))
  (when w (nconc (car z) (next (cadr z)))))

> (next '(1 2 1 4 3 1 2 4))
(1 1 1 2 2 4 4 3)
```

Решение 4.384.5

```
(defun next (w)
  (let ((z (loop for a in w
    if (equal (car w) a) collect a into v
    else collect a into u
    finally (return (list v u)))))
    (when w (nconc (car z) (next (cadr z)))))

> (next '(1 2 1 4 3 1 2 4))
(1 1 1 2 2 4 4 3)
```

Задача 4.385 =sum.lisp

В колоде карт есть n карт (n — чётное число). На каждой карте написано целое положительное число. В новую карточную игру будут играть $n/2$ человек. Каждому из игроков перед началом игры будет роздано ровно по две карты, причём каждая карта из колоды будет роздана ровно одному игроку. Перед вами стоит задача найти такой способ раздачи карт, чтобы у каждого из игроков сумма чисел, написанных на полученных им картах, совпадала. Гарантируется, что входные данные таковы, что это всегда возможно. В первой строке входных данных записано целое положительное число n ($2 \leq n \leq 100$) — количество карт. Гарантируется, что n чётно. Во второй строке следует последовательность из n целых положительных чисел $a_1, a_2, \dots, a_n$ ($1 \leq a_i \leq 100$), где a_i равно числу, написанному на i -й карте. Выведите $n/2$ пар целых чисел, по одной паре в строке — номера карт, которые нужно отдать каждому игроку. Каждая карта должна быть роздана ровно одному игроку. Карты нумеруются в том же порядке, в котором описываются во входных данных, начиная с единицы. Гарантируется, что входные данные таковы, что ответ всегда существует. Если возможных правильных ответов несколько, разрешается вывести любой из них.

Решение 4.385.1

```
(defun =sum (w)
  (pairs (loop for a in w
              for b from 1
              collect `(,a ,b))
    (/ (reduce #' + w) (length w) 1/2)))

(defun pairs (w s &aux (d (cdr w)))
  (when w
    (let ((z (find (- s (caar w)) d :key #'car)))
      (cons (list (cadar w) (cadr z))
            (pairs (remove z d :test #'equal :count 1)
                  s))))))

> (=sum '(5 4 6 1 2 0 7 3 4 8 4 4))
((1 8) (2 9) (3 5) (4 7) (6 10) (11 12))
```

Решение 4.385.2

```
(defun =sum (w
  &aux
  (v (sort
    (loop for a in w
        for b from 1
        collect `(,a ,b))
    #'< :key #'car))
  (n (/ (length w) 2)))
  (mapcar #'(lambda (a b) (list (cadr a) (cadr b)))
    (subseq v 0 n)
    (reverse (subseq v n)))))
```

```
> (=sum '(5 4 6 1 2 0 7 3 4 8 4 4))
((1 8) (2 9) (3 5) (4 7) (6 10) (11 12))
```

Решение 4.385.3

```
(defun =sum (w
             &aux
             (v (sort
                  (loop for a in w
                        for b from 1
                        collect `(,a ,b))
                  #'< :key #'car))
             (n (/ (length w) 2)))
  (loop for a in (subseq v 0 n)
        for b in (reverse (nthcdr n v))
        collect `(, (cadr a) , (cadr b))))

> (=sum '(5 4 6 1 2 0 7 3 4 8 4 4))
((1 8) (2 9) (3 5) (4 7) (6 10) (11 12))
```

Задача 4.386 rep.lisp

Определить функцию, которая возвращает повторяющиеся элементы списка.

Решение 4.386.1

```
(defun rep (w &optional ac &aux (a (car w)))
  (cond ((null w) (nreverse ac))
        ((member a (cdr w)) (rep (cdr w) (adjoin a ac)))
        (t (rep (cdr w) ac))))

> (rep '(0 1 1 2 2 2))
(1 2)
```

Решение 4.386.2

```
(defun rep (w &optional ac &aux (a (car w)))
  (cond ((null w) (nreverse ac))
        (t (rep (cdr w)
                  (if (member a (cdr w))
                      (adjoin a ac)
                      ac)))))

> (rep '(0 1 1 2 2 2))
(1 2)
```

Решение 4.386.3

```
(defun rep (w &optional ac &aux (a (car w)))
  (if w
      (rep (cdr w)
            (if (member a (cdr w))
```



```

      (adjoin a ac)
    ac))
  (nreverse ac)))

```

```

> (rep '(0 1 1 2 2 2))
(1 2)

```

Решение 4.386.4

```

(defun rep (w)
  (loop for a in (remove-duplicates w)
        when (> (count a w) 1) collect a))

```

```

> (rep '(0 1 1 2 2 2))
(1 2)

```

Задача 4.387 code-names.lisp

Определить структуру, заплнить её данными и вернуть результат запроса.

Решение 4.387.1

```

(defstruct account checking name history)

(defun client (n)
  (when (> n 0)
    (cons (make-account
            :checking (read)
            :name (read)
            :history (list (read) (read) (read) (read)))
          (client (1- n)))))

(defun names (w code)
  (loop for a in w
        when (equalp (caddr (account-history a)) code)
        collect (account-name a)))

(defun code-names (w) (names w (read)))

```

```

> (setf account (client 3))
1111
Jennifer
123
0
1000
1
2222
Michelle
456
1
1000
0

```

```

3333
Suzanne
789
1
2000
1
(#S (ACCOUNT :CHECKING 1111 :NAME JENNIFER :HISTORY (123 0 1000 1))
  #S (ACCOUNT :CHECKING 2222 :NAME MICHELLE :HISTORY (456 1 1000 0))
  #S (ACCOUNT :CHECKING 3333 :NAME SUZANNE :HISTORY (789 1 2000 1)))
> (code-names account)
0
(MICHELLE)
> (code-names account)
1
(JENNIFER SUZANNE)

```

Задача 4.388 *spec-fluency-names.lisp*

Определить структуру, заплнить её данными и вернуть результат запроса.

Решение 4.388.1

```

(defstruct head ssn name speciality cert experience fluency)

(defun heads (n)
  (when (> n 0)
    (cons (make-head
            :ssn (read)
            :name (read)
            :speciality (read)
            :cert (read)
            :experience (read)
            :fluency (read))
          (heads (1- n)))))

(defun names (w s f)
  (loop for a in w
        when (and (equalp (head-speciality a) s)
                  (equalp (head-fluency a) f))
        collect (head-name a)))

(defun spec-fluency-names (w) (names w (read) (read)))

> (setf agency (heads 3))
111-22-3333
Carmen
Biochemistry
Certificate
4
A1
123-45-6789
Jason

```

```

Law
Certificate
7
C1
333-22-4444
Brian
Law
Certificate
6
C1
(#S(HEAD :SSN |111-22-3333| :NAME CARMEN :SPECIALITY BIOCHEMISTRY
    :CERT CERTIFICATE :EXPERIENCE 4 :FLUENCY A1)
 #S(HEAD :SSN |123-45-6789| :NAME JASON :SPECIALITY LAW :CERT CERTIFI-
CATE
    :EXPERIENCE 7 :FLUENCY C1)
 #S(HEAD :SSN |333-22-4444| :NAME BRIAN :SPECIALITY LAW :CERT CERTIFI-
CATE
    :EXPERIENCE 6 :FLUENCY C1))
> (spec-fluency-names agency)
Biochemistry
A1
(CARMEN)
> (spec-fluency-names agency)
Law
C1
(JASON BRIAN)

```

Задача 4.389 code-surnames.lisp

Определить структуру, заполнить её данными и вернуть результат запроса.

Решение 4.389.1

```

(defstruct visitor number insurer surname code reason fee)

(defun visitors (n)
  (when (> n 0)
    (cons (make-visitor
              :number (read)
              :insurer (read)
              :surname (read)
              :code (read)
              :reason (read)
              :fee (read))
          (visitors (1- n)))))

(defun surnames (w input)
  (loop for a in w
        when (equalp (visitor-code a) input)
        collect (visitor-surname a)))

(defun code-surnames (w) (surnames w (read)))

```

```

> (setf clinic (visitors 3))
111
MA
Willis
0
pain
200
112
MB
Walsh
1
pain
200
113
MC
Wolfgang
2
pain
200
(#S(VISITOR :NUMBER 111 :INSURER MA :SURNAME WILLIS :CODE 0 :REASON
PAIN :FEE 200)
 #S(VISITOR :NUMBER 112 :INSURER MB :SURNAME WALSH :CODE 1 :REASON
PAIN :FEE 200)
 #S(VISITOR :NUMBER 113 :INSURER MC :SURNAME WOLFGANG :CODE 2 :REASON
PAIN :FEE 200))
> (code-surnames clinic)
0
(WILLIS)
> (code-surnames clinic)
1
(WALSH)
> (code-surnames clinic)
2
(WOLFGANG)

```

Задача 4.390 *ma.lisp*

Определить функцию, возвращающую в одномерном массиве количества положительных элементов каждой строки матрицы.

Решение 4.390.1

```

(defun ma (w
           &aux
           (v (loop for a in w collect
                    (loop for e in a
                        when (plusp e) count e))))
  (make-array (list (length v)) :initial-contents v))

> (ma '((1 2 3 4 5) (9 -5 8 -7 0) (-1 -2 -3 -4 0) (1 2 -3 4 5) (-1 2 3
-4 5)))

```

```
#(5 2 0 4 3)
```

Решение 4.390.2

```
(defun ma (w)
  (map 'array #'identity
    (loop for a in w collect
      (loop for e in a
        when (plusp e) count e))))

> (ma '((1 2 3 4 5) (9 -5 8 -7 0) (-1 -2 -3 -4 0) (1 2 -3 4 5) (-1 2 3
-4 5)))
#(5 2 0 4 3)
```

Решение 4.390.3

```
(defun ma (w
  &aux
    (v (loop for a in w collect
      (loop for e in a
        when (plusp e) count e))))
  (make-array (length v) :initial-contents v))

> (ma '((1 2 3 4 5) (9 -5 8 -7 0) (-1 -2 -3 -4 0) (1 2 -3 4 5) (-1 2 3
-4 5)))
#(5 2 0 4 3)
```

Задача 4.391 an.lisp

Определить функцию, которая после каждого элемента списка вставляет его порядковый номер.

Решение 4.391.1

```
(defun an (w &optional (n 0))
  (cond ((null w) nil)
        (t (cons (car w) (cons n (an (cdr w) (1+ n)))))))

> (an '(a b c))
(A 0 B 1 C 2)
```

Решение 4.391.2

```
(defun an (w &optional (n 0))
  (when w (cons (car w) (cons n (an (cdr w) (1+ n))))))

> (an '(a b c))
(A 0 B 1 C 2)
```

Решение 4.391.3

```
(defun an (w)
```

```

(loop for a in w for b from 0
      nconc (list a b)))

> (an '(a b c))
(A 0 B 1 C 2)

```

Задача 4.392 *plane-.lisp*

Определить функцию, которая возвращает список, получающийся после удаления из заданного списка всех элементов, представляющих собой одноуровневые списки.

Решение 4.392.1

```

(defun plane- (w &aux (a (car w)))
  (cond ((null w) nil)
        ((atom a) (cons a (plane- (cdr w))))
        ((every #'atom a) (plane- (cdr w)))
        (t (cons a (plane- (cdr w))))))

> (plane- '(1 (2 (3)) 4 (5) 6))
(1 (2 (3)) 4 6)

```

Решение 4.392.2

```

(defun plane- (w &aux (a (car w)))
  (cond ((null w) nil)
        ((or (atom a) (some #'listp a)) (cons a (plane- (cdr w))))
        (t (plane- (cdr w)))))

> (plane- '(1 (2 (3)) 4 (5) 6))
(1 (2 (3)) 4 6)

```

Решение 4.392.3

```

(defun plane- (w &aux (a (car w)))
  (when w (if (or (atom a) (some #'listp a))
              (cons a (plane- (cdr w)))
              (plane- (cdr w)))))

> (plane- '(1 (2 (3)) 4 (5) 6))
(1 (2 (3)) 4 6)

```

Решение 4.392.4

```

(defun plane- (w)
  (loop for a in w
        when (or (atom a) (some #'listp a))
        collect a))

> (plane- '(1 (2 (3)) 4 (5) 6))
(1 (2 (3)) 4 6)

```

Решение 4.392.5

```
(defun plane- (w)
  (remove-if-not #'(lambda (a) (or (atom a) (some #'listp a))) w))

> (plane- '(1 (2 (3)) 4 (5) 6))
(1 (2 (3)) 4 6)
```

Задача 4.393 *mx-string.lisp*

Дан двумерный массив. Определить функцию, которая возвращает список чаще всего встречающихся атомов типа «строка».

Решение 4.393.1

```
(defun mx-string
  (a
   &aux
    (w (remove-if-not #'stringp (flatten (2d-array-list a))))
    (v (remove-duplicates w :test #'string-equal))
    (z (loop for e in v
              collect (list e (count e w :test #'string-equal))))
    (m (loop for x in z maximize (cadr x))))
  (loop for e in z when (= (cadr e) m) collect (car e)))

(defun 2d-array-list (a)
  (loop for i below (array-dimension a 0)
        collect (loop for j below (array-dimension a 1)
                      collect (aref a i j))))

(defun flatten (w &optional acc)
  (cond ((null w) acc)
        ((atom w) (cons w acc))
        ((flatten (car w) (flat (cdr w) acc)))))

> (mx-string #2A(("aa" "bb" "cc") ("aa" "bb" "dd")))
("aa" "bb")
```

Решение 4.393.2

```
(defun mx-string
  (a
   &aux
    (w (remove-if-not #'stringp (flatten (2d-array-list a))))
    (v (remove-duplicates w :test #'string-equal))
    (z (loop for e in v
              collect (list e (count e w :test #'string-equal))))
    (m (loop for x in z maximize (cadr x))))
  (loop for e in z when (= (cadr e) m) collect (car e)))

(defun 2d-array-list (a)
```

```

(loop for i below (array-total-size a)
  collect (row-major-aref a i)))

(defun flatten (w)
  (loop for a in w if (listp a)
    nconc (flatten a) else collect a))

> (mx-string #2A(("aa" "bb" "cc") ("aa" "bb" "dd")))
("aa" "bb")

```

Решение 4.393.3

```

(defun mx-string
  (a
   &aux
    (w (remove-if-not #'stringp (flatten (2d-array-list a))))
    (v (remove-duplicates w :test #'string-equal))
    (z (loop for e in v
      collect (list e (count e w :test #'string-equal))))
    (m (loop for x in z maximize (cadr x))))
  (loop for e in z when (= (cadr e) m) collect (car e)))

(defun 2d-array-list (a)
  (loop for i below (array-total-size a)
    collect (row-major-aref a i)))

(defun flatten (w)
  (mapcan #'(lambda (a) (if (atom a) `(),a) (flatten a))) w))

> (mx-string #2A(("aa" "bb" "cc") ("aa" "bb" "dd")))
("aa" "bb")

```

Решение 4.393.4

```

(defun mx-string
  (a
   &aux
    (w (remove-if-not #'stringp (flatten (2d-array-list a))))
    (v (remove-duplicates w :test #'string-equal))
    (z (loop for e in v
      collect (list e (count e w :test #'string-equal))))
    (m (loop for x in z maximize (cadr x))))
  (loop for e in z when (= (cadr e) m) collect (car e)))

(defun 2d-array-list (a)
  (loop for i below (array-total-size a)
    collect (row-major-aref a i)))

(defun flatten (w)
  (typecase w
    (cons (mapcan #'flatten w))
    (t `(),w)))

```



```
> (mx-string #2A(("aa" "bb" "cc") ("aa" "bb" "dd")))
("aa" "bb")
```

Решение 4.393.5

```
(defun mx-string
  (a
   &aux
    (w (remove-if-not #'stringp (flatten (2d-array-list a))))
    (v (remove-duplicates w :test #'string-equal))
    (z (loop for e in v
              collect (list e (count e w :test #'string-equal))))
    (m (loop for x in z maximize (cadr x))))
  (loop for e in z when (= (cadr e) m) collect (car e)))

(defun 2d-array-list (a)
  (loop for i below (array-total-size a)
        collect (row-major-aref a i)))

(defun flatten (w)
  (if (atom w) `(,w) (mapcan #'flatten w)))

> (mx-string #2A(("aa" "bb" "cc") ("aa" "bb" "dd")))
("aa" "bb")
```

Решение 4.393.6

```
(defun mx-string (a
                  &aux
                   (b (make-array
                       (apply #'* (array-dimensions a))
                       :displaced-to a))
                   (table (make-hash-table :test 'equal))
                   acc)
  (loop for e across b do (incf (gethash e table 0)))
  (maphash #'(lambda (e n) (push `(,e ,n) acc)) table)
  (loop for a in acc
        when (= (loop for e in acc maximize (cadr e)) (cadr a))
        collect (car a)))

> (mx-string #2A(("aa" "bb" "cc") ("aa" "bb" "dd")))
("aa" "bb")
```

Решение 4.393.7

```
(defun mx-string (a
                  &aux
                   (b (remove-if-not
                       #'stringp
                       (make-array
                        (apply #'* (array-dimensions a))
                        :displaced-to a))))
```

```

                (table (make-hash-table :test 'equal))
                acc)
(loop for e across b do (incf (gethash e table 0)))
(maphash #'(lambda (e n) (push `(,e ,n) acc)) table)
(loop for a in acc
  when (= (loop for e in acc maximize (cadr e)) (cadr a))
    collect (car a)))

> (mx-string #2A(("aa" "bb" "cc") ("aa" "bb" "dd")))
("aa" "bb")

```

Решение 4.393.8

```

(defun mx-string (a
                  &aux
                    (b (remove-if-not
                        #'stringp
                        (make-array
                          (apply #'* (array-dimensions a))
                          :displaced-to a)))
                    (table (make-hash-table :test 'equal)))
  (loop for e across b do (incf (gethash e table 0)))
  (loop for k being the hash-keys in table using (hash-value v)
    when (= v (loop for v being the hash-values of table maximize
                    v))
      collect k))

> (mx-string #2A(("aa" "bb" "cc") ("aa" "bb" "dd")))
("aa" "bb")

```

Решение 4.393.9

```

(defun mx-string (a &aux (h (make-hash-table :test #'equal)))
  (loop for e across (remove-if-not
                      #'stringp
                      (make-array
                        (apply #'* (array-dimensions a))
                        :displaced-to a)) do (incf (gethash e h 0)))
  (loop for k being the hash-keys in h using (hash-value v)
    when (= v (loop for v being the hash-values of h maximize v))
      collect k))

> (mx-string #2A(("aa" "bb" "cc") ("aa" "bb" "dd")))
("aa" "bb")

```

Задача 4.394 clear-square.lisp

Определить функцию, которая удаляет из числового списка все совпадающие с заданным числовым атомом элементы и возвращает в качестве значения список из квадратов оставшихся элементов.

Решение 4.394.1

```
(defun clear-square (a w)
  (cond ((null w) nil)
        ((= (car w) a) (clear-square a (cdr w)))
        (t (cons (expt (car w) 2) (clear-square a (cdr w))))))
```

```
> (clear-square 1 '(1 2 3))
(4 9)
```

Решение 4.394.2

```
(defun clear-square (a w &optional ac)
  (cond ((null w) (nreverse ac))
        ((= (car w) a) (clear-square a (cdr w) ac))
        (t (clear-square a (cdr w) (cons (expt (car w) 2) ac)))))
```

```
> (clear-square 1 '(1 2 3))
(4 9)
```

Решение 4.394.3

```
(defun clear-square (a w &optional ac)
  (if w (clear-square a (cdr w) (if (= (car w) a) ac
                                     (cons (expt (car w) 2) ac)))
      (nreverse ac)))
```

```
> (clear-square 1 '(1 2 3))
(4 9)
```

Решение 4.394.4

```
(defun clear-square (a w)
  (loop for e in w
        unless (= e a) collect (* e e)))
```

```
> (clear-square 1 '(1 2 3))
(4 9)
```

Решение 4.394.5

```
(defun clear-square (a w)
  (loop for e in w
        unless (= e a) collect (expt e 2)))
```

```
> (clear-square 1 '(1 2 3))
(4 9)
```

Задача 4.395 rev.lisp

Определить функцию, которая удаляет из списка все подписки с нечетными порядковыми номерами, а элементы всех подписков с четными порядковыми номерами записать в обратном порядке.

Решение 4.395.1

```
(defun rev (w)
  (loop for a in w for b from 1
        when (evenp b) collect (reverse a)))

> (rev '((1 0 1 0) (2 0 2 0) (3 0 3 0) (4 0 4 0) (5 0 5 0)))
((0 2 0 2) (0 4 0 4))
```

Решение 4.395.2

```
(defun rev (w)
  (when (cdr w) (cons (reverse (cadr w)) (rev (cddr w)))))

> (rev '((1 0 1 0) (2 0 2 0) (3 0 3 0) (4 0 4 0) (5 0 5 0)))
((0 2 0 2) (0 4 0 4))
```

Решение 4.395.3

```
(defun rev (w)
  (loop for a in (cdr w) by #'cddr
        collect (reverse a)))

> (rev '((1 0 1 0) (2 0 2 0) (3 0 3 0) (4 0 4 0) (5 0 5 0)))
((0 2 0 2) (0 4 0 4))
```

Задача 4.396 m2p3.lisp

Дан список целых чисел. Определить функцию, возвращающую его копию, в которой отрицательные элементы умножены на 2, а положительные – на 3.

Решение 4.396.1

```
(defun m2p3 (w)
  (when w (cons (* (if (minusp (car w)) 2 3)
                 (car w))
                (m2p3 (cdr w)))))

> (m2p3 '(3 -2 4 -1 -5 -9 2 6))
(9 -4 12 -2 -10 -18 6 18)
```

Задача 4.397 pairs.lisp

Определить функцию, которая по списку `(a1 a2...aN)` строит сложный список `((a1 a2) (a2 a3) ...)`.

Решение 4.397.1

```
(defun pairs (w)
  (when (cdr w) (cons (list (car w) (cadr w))
```

```
(pairs (cdr w))))
```

```
> (pairs '(a b c d e))
((A B) (B C) (C D) (D E))
> (pairs '(a))
NIL
```

Задача 4.398 *plus-minus.lisp*

Вывести в виде отдельных списков положительные и отрицательные элементы заданного списка.

Решение 4.398.1

```
(defun plus-minus (w)
  (loop for i in w
        if (plusp i) collect i into p
        else if (minusp i) collect i into m
        finally (return (values p m))))

> (plus-minus '(2 1 -5 5 -2 0 -9))
(2 1 5)
(-5 -2 -9)
```

Решение 4.398.2

```
(defun plus-minus (w)
  (loop for i in w
        if (plusp i) collect i into p
        else if (minusp i) collect i into m
        finally (return (list p m))))

> (plus-minus '(2 1 -5 5 -2 0 -9))
((2 1 5) (-5 -2 -9))
```

Решение 4.398.3

```
(defun plus-minus (w)
  (values (remove-if-not #'plusp w)
          (remove-if-not #'minusp w)))

> (plus-minus '(2 1 -5 5 -2 0 -9))
(2 1 5)
(-5 -2 -9)
```

Решение 4.398.4

```
(defun plus-minus (w)
  (list (remove-if-not #'plusp w)
        (remove-if-not #'minusp w)))

> (plus-minus '(2 1 -5 5 -2 0 -9))
```

```
((2 1 5) (-5 -2 -9))
```

Задача 4.399 glue-edges.lisp

Определить функцию для преобразования одноуровневого списка в двухуровневый, в котором каждый подсписок – это пара элементов исходного списка, соответственно, первый и последний элементы, второй и предпоследний и т.д. Последний подсписок может состоять из одного элемента.

Решение 4.399.1

```
(defun glue-edges (w)
  (when w (if (cdr w)
              (cons (cons (car w) (last w))
                    (glue-edges (cdr (butlast w))))
              (list w))))

> (glue-edges '(1 2 3 4 5))
((1 5) (2 4) (3))
```

Задача 4.400 arrange.lisp

Преобразовать исходный список, расположив вначале неотрицательные элементы, затем отрицательные. Порядок неотрицательных элементов изменить на обратный, а порядок отрицательных элементов оставить прежним.

Решение 4.400.1

```
(defun arrange (w &optional n m)
  (cond ((null w) (append m (reverse n)))
        ((minusp (car w)) (arrange (cdr w) (cons (car w) n) m))
        ((arrange (cdr w) n (cons (car w) m)))))

> (arrange '(-1 0 3 -2 4))
(4 3 0 -1 -2)
```

Решение 4.400.2

```
(defun arrange (w
                &optional n m
                &aux (a (car w)) (d (cdr w)))
  (cond ((null w) (append m (reverse n)))
        ((minusp a) (arrange d (cons a n) m))
        ((arrange d n (cons a m)))))

> (arrange '(-1 0 3 -2 4))
(4 3 0 -1 -2)
```

Решение 4.400.3

```
(defun arrange (w
```

```

      &optional n m
      &aux (a (car w)) (d (cdr w)))
  (if w (if (minusp a) (arrange d (cons a n) m)
            (arrange d n (cons a m)))
    (append m (reverse n))))

> (arrange '(-1 0 3 -2 4))
(4 3 0 -1 -2)

```

Задача 4.401 *nums.lisp*

Определить функцию, создающую список только из числовых элементов исходного списка, учитывая подсписки.

Решение 4.401.1

```

(defun nums (w &optional ac)
  (cond ((null w) ac)
        ((atom w) (if (numberp w) (cons w ac) ac))
        ((nums (car w) (nums (cdr w) ac)))))

> (nums '(a 1 (2 a) (((3 4 a)) 5 a)))
(1 2 3 4 5)

```

Решение 4.401.2

```

(defun nums (w)
  (loop for a in w
        if (listp a) nconc (nums a)
        else if (numberp a) collect a))

> (nums '(a 1 (2 a) (((3 4 a)) 5 a)))
(1 2 3 4 5)

```

Задача 4.402 *grow.lisp*

Определить функцию, которая из произвольного списка строит новый список: (1 2 3 4 5 6 7 8 9 10) → ((1) (2 3) (4 5 6) (7 8 9 10)).

Решение 4.402.1

```

(defun grow (w
             &optional ac (n 1) (m n)
             &aux (a (car w)) (d (cdr w)))
  (cond ((null w) (when ac (list (nreverse ac))))
        ((zerop m) (cons (nreverse ac)
                          (grow d (cons a nil) (1+ n) n)))
        (t (grow d (cons a ac) n (1- m)))))

> (grow '(1 2 3 4 5 6 7 8 9 10))
((1) (2 3) (4 5 6) (7 8 9 10))

```

Решение 4.402.2

```
(defun grow (w
  &optional acc ac (n 1) (m n)
  &aux (a (car w)) (d (cdr w)))
  (cond ((null w) (when ac (nconc (nreverse acc)
                                   (list (nreverse ac)))))
        ((zerop m) (grow d
                          (cons (nreverse ac) acc)
                          (cons a nil)
                          (1+ n)
                          n))
        (t (grow d acc (cons a ac) n (1- m)))))

> (grow '(1 2 3 4 5 6 7 8 9 10))
((1) (2 3) (4 5 6) (7 8 9 10))
```

Решение 4.402.3

```
(defun grow (w &optional acc ac (n 1) (m n))
  (cond ((null w) (when ac (nconc (nreverse acc)
                                   (list (nreverse ac)))))
        ((zerop m) (grow (cdr w)
                          (cons (nreverse ac) acc)
                          (cons (car w) nil)
                          (1+ n)
                          n))
        (t (grow (cdr w) acc (cons (car w) ac) n (1- m)))))

> (grow '(1 2 3 4 5 6 7 8 9 10))
((1) (2 3) (4 5 6) (7 8 9 10))
```

Решение 4.402.4

```
(defun grow (w &optional (n 1) &aux (z (> n (length w))))
  (when w (cons (if z w (subseq w 0 n))
                (grow (if z nil (subseq w n)) (1+ n)))))

> (grow '(1 2 3 4 5 6 7 8 9 10))
((1) (2 3) (4 5 6) (7 8 9 10))
```

Решение 4.402.5

```
(defun grow (w &optional (n 1) &aux (z (> n (length w))))
  (when w (cons (if z w (subseq w 0 n))
                (grow (unless z (subseq w n)) (1+ n)))))

> (grow '(1 2 3 4 5 6 7 8 9 10))
((1) (2 3) (4 5 6) (7 8 9 10))
```


Задача 4.403 level-number-of-lists.lisp

Определить функцию, возвращающую список, содержащий информацию о количестве подсписков на каждом уровне вложенности.

Решение 4.403.1

```
(defun dive-count (n w &aux (a (car w)) (d (cdr w)))
  (cond ((null w) 0)
        ((= n 0) (count-if #'listp w))
        ((atom a) (dive-count n d))
        ((+ (dive-count (1- n) a) (dive-count n d)))))

(defun max-depth (w)
  (if (atom w)
      -1
      (max (1+ (max-depth (car w)))
           (max-depth (cdr w)))))

(defun level-number-of-lists (w &aux (n (1- (max-depth w))))
  (loop for a from 0 to n
        collect (list a (dive-count a w))))

> (level-number-of-lists '(a b (c d) ((e f ((g h) (i j) k)))))
((0 2) (1 1) (2 1) (3 2))
```

Решение 4.403.2

```
(defun dive-count (n w &aux (a (car w)) (d (cdr w)))
  (cond ((null w) 0)
        ((= n 0) (count-if #'listp w))
        ((atom a) (dive-count n d))
        ((+ (dive-count (1- n) a) (dive-count n d)))))

(defun max-depth (w)
  (if (atom w)
      -1
      (max (1+ (max-depth (car w)))
           (max-depth (cdr w)))))

(defun level-number-of-lists (w &optional (n (max-depth w)) (m 0))
  (when (< m n) (cons (list m (dive-count m w))
                       (level-number-of-lists w n (1+ m)))))

> (level-number-of-lists '(a b (c d) ((e f ((g h) (i j) k)))))
((0 2) (1 1) (2 1) (3 2))
```

Решение 4.403.3 * вариант «одна функция»

```
(defun level (w
              &optional
              ac
```

```

(n (labels ((max-depth (v)
  (if (atom v)
    -1
    (max (1+ (max-depth (car v)))
        (max-depth (cdr v))))))
  (max-depth w)))
(m 0))
(if (< m n)
  (level w
    (cons (list m
      (labels ((dive-count (n
        v
        &aux
        (a (car v))
        (d (cdr v)))
        (cond ((null v) 0)
              ((= n 0) (count-if #'listp v))
              ((atom a) (dive-count n d))
              ((+ (dive-count (1- n) a)
                  (dive-count n d))))))
        (dive-count m w)))
      ac)
    n
    (1+ m))
  (reverse ac)))

> (level '(a b (c d) ((e f ((g h) (i j) k)))))
((0 2) (1 1) (2 1) (3 2))

```

Решение 4.403.4

```

(defun dive-count (n w &aux (a (car w)) (d (cdr w)))
  (cond ((null w) 0)
        ((= n 0) (count-if #'listp w))
        ((atom a) (dive-count n d))
        ((+ (dive-count (1- n) a) (dive-count n d)))))

(defun max-depth (w)
  (if (atom w)
    -1
    (max (1+ (max-depth (car w)))
        (max-depth (cdr w)))))

(defun level-number-of-lists (w &optional ac (n (max-depth w)) (m 0))
  (if (< m n)
    (level-number-of-lists w
      (cons (list m (dive-count m w))
            ac)
      n
      (1+ m))
    (reverse ac)))

> (level-number-of-lists '(a b (c d) ((e f ((g h) (i j) k)))))

```

```
((0 2) (1 1) (2 1) (3 2))
```

Задача 4.404 *ssort.lisp*

Определить функцию, которая сортирует список, собрав вместе элементы одних типов (целые числа, действительные числа, символы, списки).

Решение 4.404.1

```
(defun ssort (w &optional v i r s &aux (a (car w)))
  (cond ((null w)
        (nconc (reverse i) (reverse r) (reverse s) (reverse v)))
        ((listp a) (ssort (cdr w) (cons a v) i r s))
        ((integerp a) (ssort (cdr w) v (cons a i) r s))
        ((realp a) (ssort (cdr w) v i (cons a r) s))
        (t (ssort (cdr w) v i r (cons a s)))))

> (ssort '(1 56.7 (99 6 0) p 89 30.8 (12 89 6) 0 gh))
(1 89 0 56.7 30.8 P GH (99 6 0) (12 89 6))
```

Задача 4.405 *sum-m-p.lisp*

Определить функцию, вычисляющую суммы положительных и отрицательных элементов одноуровневого числового списка.

Решение 4.405.1

```
(defun sum-m-p (w &optional (m 0) (p 0))
  (cond ((null w) (list m p))
        ((minusp (car w)) (sum-m-p (cdr w) (+ m (car w)) p))
        (t (sum-m-p (cdr w) m (+ p (car w))))))

> (sum-m-p '(-3 -2 -1 0 1 2 3))
(-6 6)
```

Решение 4.405.2

```
(defun sum-m-p (w)
  (sum w 0 0))

(defun sum (w m p)
  (cond ((null w) (list m p))
        ((minusp (car w)) (sum (cdr w) (+ m (car w)) p))
        (t (sum (cdr w) m (+ p (car w))))))

> (sum-m-p '(-3 -2 -1 0 1 2 3))
(-6 6)
```

Решение 4.405.3

```
(defun sum-m-p (w) (sum w 0 0))
```

```
(defun sum (w m p &aux (a (car w)))
  (if w
    (if (minusp a)
      (sum (cdr w) (+ m a) p)
      (sum (cdr w) m (+ p a)))
    (values m p)))
```

```
> (sum-m-p '(-3 -2 -1 0 1 2 3))
-6
6
```

Решение 4.405.4

```
(defun sum-m-p (w)
  (sum w 0 0))

(defun sum (w m p &aux (a (car w)))
  (if w (let ((z (minusp a)))
    (sum (cdr w) (if z (+ m a) m) (if z p (+ p a))))
    (values m p)))
```

```
> (sum-m-p '(-3 -2 -1 0 1 2 3))
-6
6
```

Решение 4.405.5

```
(defun sum-m-p (w)
  (list (sum w #'minusp) (sum w #'plusp)))

(defun sum (w f &optional (n 0) &aux (a (car w)))
  (if w (sum (cdr w) f (if (funcall f a) (+ n a) n)) n))
```

```
> (sum-m-p '(-3 -2 -1 0 1 2 3))
(-6 6)
```

Решение 4.405.6

```
(defun sum-m-p (w)
  (values (sum w #'minusp) (sum w #'plusp)))

(defun sum (w f &optional (n 0) &aux (a (car w)))
  (if w (sum (cdr w) f (if (funcall f a) (+ n a) n)) n))
```

```
> (sum-mp '(-3 -2 -1 0 1 2 3))
-6
6
```

Решение 4.405.7

```
(defun sum-m-p (w)
  (values (sum w #'minusp) (sum w #'plusp)))
```

```
(defun sum (w f)
  (loop for a in w when (funcall f a) sum a))
```

```
> (sum-m-p '(-3 -2 -1 0 1 2 3))
-6
6
```

Решение 4.405.8

```
(defun sum-m-p (w)
  (values (reduce #' + (remove-if #'plusp w))
          (reduce #' + (remove-if #'minusp w))))
```

```
> (sum-m-p '(-3 -2 -1 0 1 2 3))
-6
6
```

Решение 4.405.9

```
(defun sum-m-p (w)
  (values (sum w #'minusp) (sum w #'plusp)))
```

```
(defun sum (w f)
  (reduce #' + (remove-if f w)))
```

```
> (sum-m-p '(-3 -2 -1 0 1 2 3))
6
-6
```

Решение 4.405.10

```
(defun sum-m-p (w)
  (values (sum w #'minusp) (sum w #'plusp)))
```

```
(defun sum (w f)
  (eval (cons ' + (loop for a in w unless (funcall f a) collect a))))
```

```
> (summp '(-3 -2 -1 0 1 2 3))
6
-6
```

Решение 4.405.11

```
(defun sum-m-p (w)
  (labels ((sum (v m p)
            (cond ((null v) (list m p))
                  ((minusp (car v)) (sum (cdr v) (+ m (car v)) p))
                  (t (sum (cdr v) m (+ p (car v)))))))
    (sum w 0 0)))
```

```
> (sum-m-p '(-3 -2 -1 0 1 2 3))
(-6 6)
```

Решение 4.405.12

```
(defun sum-m-p (w)
  (labels ((sum (v m p &aux (a (car v)))
            (cond ((null v) (list m p))
                  ((minusp a) (sum (cdr v) (+ m a) p))
                  (t (sum (cdr v) m (+ p a))))))
    (sum w 0 0)))

> (sum-m-p '(-3 -2 -1 0 1 2 3))
(-6 6)
```

Задача 4.406 pluralize.lisp

Определить функцию, возвращающую множественное число слова, представленного списком букв.

Решение 4.406.1

```
(defun pluralize (w)
  (cond ((and (member (car (last w 2))
                      '( b c d f g h j k l m n p q r s t v w x y z))
              (eql (car (last w)) 'y))
        (our-nconc (butlast w) '(i e s)))
        ((and (member (car (last w 2))
                      '( b c d f g h j k l m n p q r s t v w x y z))
              (eql (car (last w)) 'o))
        (our-nconc w '(e s)))
        (t (our-nconc w '(s)))))

(defun our-nconc (w v)
  (if w (progn (rplacd (last w) v) w) v))

> (pluralize '(h i s t o r y))
(H I S T O R I E S)
> (pluralize '(h e r o))
(H E R O E S)
> (pluralize '(w a r))
(W A R S)
```

Решение 4.406.2

```
(defun pluralize (w &aux
  (a (car (last w 2)))
  (b (car (last w)))
  (m (member
      a
      '( b c d f g h j k l m n p q r s t v w x y z))))
  (cond ((and m (eql b 'y)) (nconc (butlast w) '(i e s)))
        ((and m (eql b 'o)) (nconc w '(e s)))
        (t (nconc w '(s)))))
```

```
> (pluralize '(h i s t o r y))
(H I S T O R I E S)
> (pluralize '(h e r o))
(H E R O E S)
> (pluralize '(w a r))
(W A R S)
```

Решение 4.406.3

```
(defun pluralize (w &aux (z (last w 2)) (a (car z)) (b (cadr z))
  (m (member
    a
    '(b c d f g h j k l m n p q r s t v w x y z))))
  (cond ((and m (eql b 'y)) (nconc (butlast w) '(i e s)))
    ((and m (eql b 'o)) (nconc w '(e s)))
    (t (nconc w '(s)))))
```

```
> (pluralize '(h i s t o r y))
(H I S T O R I E S)
> (pluralize '(h e r o))
(H E R O E S)
> (pluralize '(w a r))
(W A R S)
```

Решение 4.406.4

```
(defun pluralize (w &aux (z (last w 2)) (a (car z)) (b (cadr z))
  (m (member
    a
    '(b c d f g h j k l m n p q r s t v w x y z))))
  (nconc (butlast w) (cond ((and m (eql b 'y)) '(i e s))
    ((and m (eql b 'o)) `(',b e s))
    (`(',b s)))))
```

```
> (pluralize '(h i s t o r y))
(H I S T O R I E S)
> (pluralize '(h e r o))
(H E R O E S)
> (pluralize '(w a r))
(W A R S)
```

Решение 4.406.5

```
(defun pluralize (w &aux (z (last w 2)) (a (car z)) (b (cadr z))
  (m (not (member a '(a e i o u y)))))
  (nconc (butlast w) (cond ((and m (eql b 'y)) '(i e s))
    ((and m (eql b 'o)) `(',b e s))
    (`(',b s)))))
```

```
> (pluralize '(h i s t o r y))
(H I S T O R I E S)
> (pluralize '(h e r o))
(H E R O E S)
```

```
> (pluralize '(w a r))
(W A R S)
```

Задача 4.407 *arrange.lisp*

Определить функцию, разбивающую список на пять списков: первый список содержит все целые числа входного списка, второй – все действительные числа, третий – все строчные константы, четвертый – все символы входного списка, пятый – все что не вошло в первые четыре.

Решение 4.407.1

```
(defun arrange ()
  (loop for a in (read)
    if (integerp a) collect a into integers
    else if (realp a) collect a into reals
    else if (stringp a) collect a into strings
    else if (symbolp a) collect a into symbols
    else collect a into other
    finally (return (list integers
                          reals
                          strings
                          symbols
                          other))))

> (arrange)
(1 "e" 2.3 3/4 10 20 dot '(a))
((1 10 20) (2.3 3/4) ("e") (DOT) ('(A)))
```

Решение 4.407.2 (автор – transformator.t, www.cyberforum.ru)

```
(define-modify-macro push-last (nw)
  (lambda (list nw) (nconc list (list nw))))

(defmacro arrange (list &rest type)
  `(loop
    with h = (make-hash-table :test #'equal)
    for e in ,list
    do (typecase e
        ,@(loop
            for z in type
            collect (list z `(push-last (gethash ',z h) e))))
    finally
      (terpri)
      (maphash #'(lambda (k v)
                    (format t "~A~20T:: ~A~%" k v))
                h)))

> (arrange '(1 "e" 2.3 3/4 10 20 dot) integer string double-float sym-
bol ratio)

SYMBOL                :: (DOT)
```



```

RATIO          :: (3/4)
STRING         :: (e)
INTEGER        :: (1 10 20)
NIL

```

Задача 4.408 char-codes.lisp

Определить функцию, которая в многоуровневом списке заменяет все элементы-литеры соответствующими им ASCII-кодами.

Решение 4.408.1

```

(defun char-codes (w)
  (loop for a in w
        if (listp a) collect (char-codes a)
        else collect (char-code a)))

> (char-codes '((#\a) #\b (#\Ш)))
((97) 98 (1064))

```

Решение 4.408.2

```

(defun char-codes (w)
  (loop for a in w
        if (listp a) collect (char-codes a)
        else collect (if (and (symbolp a)
                               (= (length (string a)) 1))
                          (chars a) a)))

(defun chars (s)
  (car (loop for a across (string s)
             collect (char-code a))))

> (char-codes '(a (aaa) 7 b bbb 8 c (ccc)))
(65 (AAA) 7 66 BBB 8 67 (CCC))

```

Задача 4.409 rems.lisp

Определить функцию, которая в многоуровневом списке заменяет все элементы-целые числа остатками от их деления на 2.

Решение 4.409.1

```

(defun rems (w)
  (cond ((null w) nil)
        ((listp (car w)) (cons (rems (car w)) (rems (cdr w)))))
        ((integerp (car w)) (cons (rem (car w) 2) (rems (cdr w)))))
        ((cons (car w) (rems (cdr w)))))

> (rems '(a 2 (3)))
(A 0 (1))

```

Решение 4.409.2

```
(defun remс (w)
  (when w (cons (cond ((listp (car w)) (remс (car w)))
                     ((integerp (car w)) (rem (car w) 2))
                     ((car w)))
                (remс (cdr w)))))

> (remс '(a 2 (3)))
(A 0 (1))
```

Решение 4.409.3

```
(defun remс (w)
  (loop for a in w
        if (listp a) collect (remс a)
        else if (integerp a) collect (rem a 2)
        else collect a))

> (remс '(a 2 (3) 8))
(A 0 (1) 0)
```

Задача 4.410 odd-sum.lisp

Дан список, содержащий целые числа, значения nil и другие списки. Определить функцию, которая заменяет все списки, содержащие только нечетные числа, суммой этих чисел.

Решение 4.410.1

```
(defun odd-sum (w &aux (a (car w)))
  (when w (cons (cond ((null a) nil)
                    ((and (listp a)
                          (every #'numberp a)
                          (every #'oddp a))
                  (reduce #'+ a))
                    ((listp a) (odd-sum a))
                    (t a))
                (odd-sum (cdr w)))))

> (odd-sum '(1 nil (2 3) (5 (9 11) 77)))
(1 NIL (2 3) (5 20 77))
```

Решение 4.410.2

```
(defun odd-sum (w &aux (a (car w)))
  (when w (cons (cond ((null a) nil)
                    ((listp a) (if (and (every #'numberp a)
                                          (every #'oddp a))
                                    (reduce #'+ a)
                                    (odd-sum a)))
                    (t a))
                (odd-sum (cdr w)))))
```

```

      (t a))
    (odd-sum (cdr w))))))

> (odd-sum '(1 nil (2 3) (5 (9 11) 77)))
(1 NIL (2 3) (5 20 77))

```

Решение 4.410.3

```

(defun odd-sum (w &aux (a (car w)))
  (when w (cons (when a (if (listp a) (if (and (every #'numberp a)
                                              (every #'oddp a))
                                          (reduce #'+ a)
                                          (odd-sum a))
                  a))
                (odd-sum (cdr w))))))

> (odd-sum '(1 nil (2 3) (5 (9 11) 77)))
(1 NIL (2 3) (5 20 77))

```

Решение 4.410.4

```

(defun odd-sum (w &aux (a (car w)))
  (when w (cons (when a (if (atom a) a
                            (if (and (every #'numberp a)
                                        (every #'oddp a))
                                (reduce #'+ a)
                                (odd-sum a))))
                (odd-sum (cdr w))))))

> (odd-sum '(1 nil (2 3) (5 (9 11) 77)))
(1 NIL (2 3) (5 20 77))

```

Решение 4.410.5

```

(defun odd-sum (w)
  (loop for a in w
        if (atom a) collect a
        else collect (if (and (every #'numberp a)
                              (every #'oddp a))
                          (reduce #'+ a)
                          (odd-sum a))))

> (odd-sum '(1 nil (2 3) (5 (9 11) 77)))
(1 NIL (2 3) (5 20 77))

```

Решение 4.410.6

```

(defun odd-sum (w)
  (mapcar #'(lambda (a)
              (if (atom a) a
                  (if (and (every #'numberp a)
                          (every #'oddp a))
                      (reduce #'+ a)
                      (odd-sum a)))) w))

```

```

                                (odd-sum a))))
      w))

> (odd-sum '(1 nil (2 3) (5 (9 11) 77)))
(1 NIL (2 3) (5 20 77))

```

Задача 4.411 ending.lisp

Множественное число большинства английских существительных получается путем добавления буквы s к форме единственного числа. Но если существительное заканчивается буквой y, следующей за согласной, множественное число образуется путем замены буквы y на сочетание ies: если же существительное заканчивается буквой o, следующей за согласной, множественное число образуется путем добавления сочетания es. Определить функцию, вычисляющую множественное число для английского слова, представленного списком букв.

Решение 4.411.1

```

(defun ending (w)
  (cond ((and (member (car (last w 2))
                      '(b c d f g h j k l m n p q r s t v w x y z))
              (eql (car (last w)) 'y))
         (our-nconc (butlast w) '(i e s)))
        ((and (member (car (last w 2))
                      '(b c d f g h j k l m n p q r s t v w x y z))
              (eql (car (last w)) 'o))
         (our-nconc w '(e s)))
        (t (our-nconc w '(s)))))

(defun our-nconc (w v)
  (if w (progn (rplacd (last w) v) w) v))

```

```

> (ending '(h i s t o r y))
(H I S T O R I E S)
> (ending '(w a r))
(W A R S)

```

Решение 4.411.2

```

(defun ending
  (w
   &aux
    (a (car (last w 2)))
    (b (car (last w)))
    (m (member a '(b c d f g h j k l m n p q r s t v w x y z))))
  (cond ((and m (eql b 'y)) (nconc (butlast w) '(i e s)))
        ((and m (eql b 'o)) (nconc w '(e s)))
        (t (nconc w '(s)))))

> (ending '(h i s t o r y))
(H I S T O R I E S)

```

```
> (ending '(w a r))
(W A R S)
```

Решение 4.411.3

```
(defun ending
  (w &aux (z (last w 2)) (a (car z)) (b (cadr z))
    (m (member a '(b c d f g h j k l m n p q r s t v w x y z))))
  (nconc (butlast w) (cond ((and m (eql b 'y)) '(i e s))
    ((and m (eql b 'o)) `(',b e s))
    (`(',b s)))))
```

```
> (ending '(h i s t o r y))
(H I S T O R I E S)
> (ending '(w a r))
(W A R S)
```

Решение 4.411.4

```
(defun ending (w &aux (z (last w 2)) (a (car z)) (b (cadr z))
  (m (not (member a '(a e i o u y)))))
  (nconc (butlast w) (cond ((and m (eql b 'y)) '(i e s))
    ((and m (eql b 'o)) `(',b e s))
    (`(',b s)))))
```

```
> (ending '(h i s t o r y))
(H I S T O R I E S)
> (ending '(w a r))
(W A R S)
```

Задача 4.412 num.lisp

Дан многоуровневый список, атомы которого могут быть как числа, так и символьные атомы. Определить функцию, возвращающую список, в котором все числа заменены на атом NUM.

Решение 4.412.1

```
(defun num (w &aux (a (car w)) (d (cdr w)))
  (cond ((null w) nil)
    ((listp a) (cons (num a) (num d)))
    ((numberp a) (cons 'num (num d)))
    ((cons a (num d)))))
```

```
> (num '(a s 4 (s 1 2) c))
(A S NUM (S NUM NUM) C)
```

Решение 4.412.2

```
(defun num (w &aux (a (car w)))
  (when w (cons (cond ((listp a) (num a))
    ((numberp a) 'num)
```

```

      (t a))
    (num (cdr w))))))

```

```

> (num '(a s 4 (s 1 2) c))
(A S NUM (S NUM NUM) C)

```

Решение 4.412.3

```

(defun num (w)
  (loop for a in w
        if (listp a) collect (num a)
        else collect (if (numberp a) 'num a)))

```

```

> (num '(a s 4 (s 1 2) c))
(A S NUM (S NUM NUM) C)

```

Решение 4.412.4

```

(defun num (w)
  (mapcar #'(lambda (a) (if (listp a)
                           (num a)
                           (if (numberp a)
                               'num
                               a)))
    w))

```

```

> (num '(a s 4 (s 1 2) c))
(A S NUM (S NUM NUM) C)

```

Решение 4.412.5

```

(defun num (w)
  (mapcar #'(lambda (a) (typecase a
                           (list (num a))
                           (number 'num)
                           (otherwise a)))
    w))

```

```

> (num '(a s 4 (s 1 2) c))
(A S NUM (S NUM NUM) C)

```

Решение 4.412.6

```

(defun num (w)
  (loop for a in w
        collect (typecase a
                     (list (num a))
                     (number 'num)
                     (otherwise a))))

```

```

> (num '(a s 4 (s 1 2) c))
(A S NUM (S NUM NUM) C)

```

Задача 4.413 odd-min.lisp

Дан дан список целых чисел. Определить функцию, возвращающую список, в котором все нечетные элементы заменены на значение минимального элемента исходного списка.

Решение 4.413.1

```
(defun odd-min (w) (substitute-if (reduce #'min w) #'oddp w))

> (odd-min '(1 2 3 4 -1 5 6 7 8))
(-1 2 -1 4 -1 -1 6 -1 8)
```

Задача 4.414 numbers.lisp

Определить функцию, которая извлекает из списка числа.

Решение 4.414.1

```
(defun numbers (w &optional ac)
  (cond ((null w) ac)
        ((atom w) (if (numberp w) (cons w ac) ac))
        ((numbers (car w) (numbers (cdr w) ac)))))

> (numbers '(a ((1 b)) ((2) c) 3))
(1 2 3)
```

Задача 4.415 <average<.lisp

Дан список чисел. Найти среднее значение элементов списка и расположить в виде подсписков: слева - меньшие среднего значения, справа - большие среднего значения.

Решение 4.415.1

```
(defun <average< (w &optional b (m (/ (reduce #'+ w) (length w))) c)
  (cond ((null w) (list (sort b #'<) m (sort c #'<)))
        ((< (car w) m) (<average< (cdr w) (cons (car w) b) m c))
        ((> (car w) m) (<average< (cdr w) b m (cons (car w) c)))
        (t (<average< (cdr w) b m c))))

> (<average< '(1 2 3 4 5))
((1 2) 3 (4 5))
```

Решение 4.415.2

```
(defun <average< (w &aux (m (/ (reduce #'+ w) (length w))))
  (loop for a in w
        if (< a m) collect a into b
        else if (> a m) collect a into c
        finally (return (list b m c))))
```

```
> (<average< '(1 2 3 4 5))
((1 2) 3 (4 5))
```

Задача 4.416 average-percent.lisp

Дан список с результатами экзаменов, содержащий подписки вида: (<Фамилия> (<Предмет 1>, <Оценка>), ..., (<Предмет 4>, <Оценка>)). Определить по каждому предмету средний балл и процент успеваемости.

Решение 4.416.1

```
(defun average-percent (w
                        &aux
                          (v (loop for a in w nconc (cdr a)))
                          (z (remove-duplicates
                              (loop for a in v collect (car a))
                              :test #'string=)))
  (loop for a in z collect (cons a (cnt a v))))

(defun cnt (s w
            &aux
              (v (loop for a in w
                        when (equal (car a) s) collect (cadr a)))
              (n (length v)))
  (list (float (/ (reduce '+ v) n))
        (float (/ (* (count-if #'(lambda (a) (> a 2)) v) 100) n))))

> (average-percent '((ivanov ("law" 5) ("computer science" 5)) (petrov
("computer science" 2))))
(("law" 5.0 100.0) ("computer science" 3.5 50.0))
```

Задача 4.417 minus-plus.lisp

Список целых чисел необходимо разбить на два списка, поместив в первый отрицательные, а во второй – положительные элементы.

Решение 4.417.1

```
(defun minus-plus (w &optional m p &aux (a (car w)))
  (cond ((null w) (list (nreverse m) (nreverse p)))
        ((minusp a) (minus-plus (cdr w) (cons a m) p))
        ((plusp a) (minus-plus (cdr w) m (cons a p)))
        ((minus-plus (cdr w) m p))))

> (minus-plus '(-2 -1 0 1 2))
((-2 -1) (1 2))
```

Решение 4.417.2 * запятая позволяет разквотировать s-выражение

```
(defun minus-plus (w &optional m p &aux (a (car w)))
```



```
(cond ((null w) `(,(nreverse m) ,(nreverse p)))
      ((minusp a) (minus-plus (cdr w) (cons a m) p))
      ((plusp a) (minus-plus (cdr w) m (cons a p)))
      ((minus-plus (cdr w) m p))))
```

```
> (minus-plus '(-2 -1 0 1 2))
((-2 -1) (1 2))
```

Решение 4.417.3

```
(defun minus-plus (w)
  (loop for a in w
        if (minusp a) collect a into m
        else if (plusp a) collect a into p
        finally (return (list m p))))
```

```
> (minus-plus '(-2 -1 0 1 2))
((-2 -1) (1 2))
```

Решение 4.417.4

```
(defun minus-plus (w)
  (if w (apply
        #'(lambda (m p)
            (list (if (minusp (car w)) (cons (car w) m) m)
                  (if (plusp (car w)) (cons (car w) p) p)))
        (minus-plus (cdr w)))
    (list nil nil)))
```

```
> (minus-plus '(-2 -1 0 1 2))
((-2 -1) (1 2))
```

Решение 4.417.5

```
(defun minus-plus (w)
  (if w (apply
        #'(lambda (m p)
            (list (if (minusp (car w)) (cons (car w) m) m)
                  (if (plusp (car w)) (cons (car w) p) p)))
        (minus-plus (cdr w)))
    (list nil nil)))
```

```
> (minus-plus '(-2 -1 0 1 2))
((-2 -1) (1 2))
```

Решение 4.417.6

```
(defun minus-plus (w)
  (if w (apply
        #'(lambda (m p &aux (a (car w)))
            (list (if (minusp a) (cons a m) m)
                  (if (plusp a) (cons a p) p)))
        (minus-plus (cdr w)))
```

```
(list nil nil)))
```

```
> (minus-plus '(-2 -1 0 1 2))
((-2 -1) (1 2))
```

Решение 4.417.7

```
(defun minus-plus (w)
  (if w (reduce
        #'(lambda (m p &aux (a (car w)))
            `((if (minusp a) (cons a m) m)
              , (if (plusp a) (cons a p) p)))
        (minus-plus (cdr w)))
    `(nil nil)))
```

```
> (minus-plus '(-2 -1 0 1 2))
((-2 -1) (1 2))
```

Решение 4.417.8

```
(defun minus-plus (w)
  `(,(mapcan #'(lambda (a) (when (< a 0) `(,a))) w)
    ,(mapcan #'(lambda (a) (when (> a 0) `(,a))) w)))
```

```
> (minus-plus '(-2 -1 0 1 2))
((-2 -1) (1 2))
```

Решение 4.417.9

```
(defun minus-plus (w) `(,(mp w #'<) ,(mp w #'>)))

(defun mp (w p)
  (mapcan #'(lambda (a) (when (funcall p a 0) `(,a))) w))
```

```
> (minus-plus '(-2 -1 0 1 2))
((-2 -1) (1 2))
```

Решение 4.417.10

```
(defun minus-plus (w)
  `(,(remove-if-not #'minusp w) ,(remove-if-not #'plusp w)))

> (minus-plus '(-2 -1 0 1 2))
((-2 -1) (1 2))
```

Задача 4.418 break-into-odds-evens.lisp

Произвольный список вида (a1,a2,...,ak) разбить на два подсписка (a1,a3,a5...) и (a2,a4,a6...).

Решение 4.418.1

```
(defun break-into-odds-evens (w) (list (odds w) (evens w)))

(defun odds (w)
  (when w (cons (car w) (odds (cddr w)))))

(defun evens (w)
  (when (cdr w) (cons (cadr w) (evens (cddr w)))))

> (break-into-odds-evens '(a 1 b 2 c 3))
((A B C) (1 2 3))
```

Решение 4.418.1

```
(defun break-into-odds-evens (w)
  (if w (reduce #'(lambda (odds evens)
                    `((, (cons (car w) evens) ,odds))
                    (break-into-odds-evens (cdr w)))
      `(nil nil)))

> (break-into-odds-evens '(a 1 b 2 c 3))
((A B C) (1 2 3))
```

Задача 4.419 student-average-percent.lisp

Дан список с результатами экзаменов, содержащий подписки вида: (<Фамилия> (<Предмет 1>, <Оценка>), ... (<Предмет 4>, <Оценка>)). Определить для каждого студента средний балл и его процент успеваемости.

Решение 4.419.1

```
(defun student-average-percent (w)
  (loop for a in w
        collect (list (car a)
                      (float (/ (loop for b in (cdr a)
                                       sum (cadr b))
                                (length (cdr a))))
                      (/ (* (loop for c in (cdr a)
                                   when (> (cadr c) 2) count c)
                          100)
                      (length (cdr a)))))

> (student-average-percent '((ivanov ("law" 5) ("computer science" 4))
                             (petrov ("computer science" 2))))
((IVANOV 4.5 100) (PETROV 2.0 0))
```

Задача 4.420 escort-plusp-zero-p-with-asterisk.lisp

Дан список. Определить функцию, которая возвращает список, где каждое неотрицательное число заменено на список, первым элементом которого является это число, а вторым - звездочка.

Решение 4.420.1

```
(defun escort-plusp-zero-with-asterisk (w)
  (loop for a in w
    if (and (numberp a)
            (or (zerop a) (plusp a)))
      collect ` (,a *)
      else collect a))

> (escort-plusp-zero-with-asterisk '(0 b a -2 4))
((0 *) B A -2 (4 *))
```

Решение 4.420.2

```
(defun escort-plusp-zero-with-asterisk (w)
  (loop for a in w
    if (and (numberp a) (>= a 0))
      collect ` (,a *)
      else collect a))

> (escort-plusp-zero-with-asterisk '(0 b a -2 4))
((0 *) B A -2 (4 *))
```

Решение 4.420.3

```
(defun escort-plusp-zero-with-asterisk (w)
  (loop for a in w collect
    (if (and (numberp a) (>= a 0)) ` (,a *) a)))

> (escort-plusp-zero-with-asterisk '(0 b a -2 4))
((0 *) B A -2 (4 *))
```

Решение 4.420.4

```
(defun escort-plusp-zero-with-asterisk (w &aux (a (car w)))
  (cond ((null w) nil)
        ((and (numberp a) (>= a 0))
         (cons ` (,a *) (escort-plusp-zero-with-asterisk (cdr w))))
        ((cons a (escort-plusp-zero-with-asterisk (cdr w))))))

> (escort-plusp-zero-with-asterisk '(0 b a -2 4))
((0 *) B A -2 (4 *))
```

Решение 4.420.5

```
(defun escort-plusp-zero-with-asterisk (w &aux (a (car w)))
  (when w (cons (if (and (numberp a) (>= a 0)) ` (,a *) a)
                (escort-plusp-zero-with-asterisk (cdr w)))))

> (escort-plusp-zero-with-asterisk '(0 b a -2 4))
((0 *) B A -2 (4 *))
```

Решение 4.420.6

```
(defun escort-plusp-zero-p-with-asterisk (w)
  (mapcar #'(lambda (a) (if (and (numberp a) (>= a 0)) `(,a *) a)) w))

> (escort-plusp-zero-p-with-asterisk '(0 b a -2 4))
((0 *) B A -2 (4 *))
```

```
(defun escort-plusp-zero-with-asterisk (w)
  (when w ((lambda (a r)
              (cons (if (and (numberp a) (>= a 0)) `(,a *) a) r))
            (car w) (escort-plusp-zero-with-asterisk (cdr w)))))

> (escort-plusp-zero-with-asterisk '(0 b a -2 4))
((0 *) B A -2 (4 *))
```

```
...nconc, nreconc, nreverse, mapcan, mapcon,
delete, delete-if, delete-if-not, nsubst,
nsubst-if, nsubst-if-not, union, nintersection,
nset-difference, nset-exclusive-or
разрушают вконец наши СПИСКИ...
```

На полках в магазине лежат фрукты: '((apple apple banana) (pineapple pinescone orange) (pear qiwi mango)). Определите функцию, которая перекладывает все фрукты на одну полку: (apple apple banana pineapple pinescone orange pear qiwi mango).

Решение 4.421.1

```
(defun shelf (w) (loop for a in w append a))

> (shelf '((apple apple banana) (pineapple pineapple orange) (pear qiwi mango)))
(APPLE APPLE BANANA PINEAPPLE PINECONE ORANGE PEAR QIWI MANGO)
> (setf w '((apple apple banana) (pineapple pineapple orange) (pear qiwi mango)))
((APPLE APPLE BANANA) (PINEAPPLE PINECONE ORANGE) (PEAR QIWI MANGO))
> (shelf w)
(APPLE APPLE BANANA PINEAPPLE PINECONE ORANGE PEAR QIWI MANGO)
> w
((APPLE APPLE BANANA) (PINEAPPLE PINECONE ORANGE) (PEAR QIWI MANGO))
```

```
(defun shelf (w) (reduce #'append w))

> (shelf '((apple apple banana) (pineapple pinecone orange) (pear qiwi
mango)))
```

```
(APPLE APPLE BANANA PINEAPPLE PINECONE ORANGE PEAR QIWI MANGO)
> (setf w '((apple apple banana) (pineapple pinecone orange) (pear
qiwi mango)))
((APPLE APPLE BANANA) (PINEAPPLE PINECONE ORANGE) (PEAR QIWI MANGO))
> (shelf w)
(APPLE APPLE BANANA PINEAPPLE PINECONE ORANGE PEAR QIWI MANGO)
> w
((APPLE APPLE BANANA) (PINEAPPLE PINECONE ORANGE) (PEAR QIWI MANGO))
```

Решение 4.421.3

```
(defun shelf (w) (loop for a in w nconc a))

> (shelf '((apple apple banana) (pineapple pinecone orange) (pear qiwi
mango)))
(APPLE APPLE BANANA PINEAPPLE PINECONE ORANGE PEAR QIWI MANGO)
> (setf w '((apple apple banana) (pineapple pinecone orange) (pear
qiwi mango)))
((APPLE APPLE BANANA) (PINEAPPLE PINECONE ORANGE) (PEAR QIWI MANGO))
> (shelf w)
(APPLE APPLE BANANA PINEAPPLE PINECONE ORANGE PEAR QIWI MANGO)
> w
((APPLE APPLE BANANA PINEAPPLE PINECONE ORANGE PEAR QIWI MANGO)
 (PINEAPPLE PINECONE ORANGE PEAR QIWI MANGO) (PEAR QIWI MANGO))
```

Решение 4.421.4

```
(defun shelf (w) (mapcan #'identity w))

> (shelf '((apple apple banana) (pineapple pinecone orange) (pear qiwi
mango)))
(APPLE APPLE BANANA PINEAPPLE PINECONE ORANGE PEAR QIWI MANGO)
> (setf w '((apple apple banana) (pineapple pinecone orange) (pear
qiwi mango)))
((APPLE APPLE BANANA) (PINEAPPLE PINECONE ORANGE) (PEAR QIWI MANGO))
> (shelf w)
(APPLE APPLE BANANA PINEAPPLE PINECONE ORANGE PEAR QIWI MANGO)
> w
((APPLE APPLE BANANA PINEAPPLE PINECONE ORANGE PEAR QIWI MANGO)
 (PINEAPPLE PINECONE ORANGE PEAR QIWI MANGO) (PEAR QIWI MANGO))
```

Задача 4.422 *capi.lisp*

Определить функцию, которая делает заглавной первую букву первого слова каждого предложения, если та не является таковой.

Решение 4.422.1

```
(defun capi (w)
  (mapcar #'
    (lambda (s)
      (concatenate 'string
```

```

                (string (char-upcase (char s 0)))
                (subseq s 1)))
    w))

> (capi '("a aa" "Bb b"))
("A aa" "Bb b")

```

Решение 4.422.2

```

(defun capi (w)
  (mapcar #'
    (lambda (s)
      (setf (char s 0) (char-upcase (char s 0)))
      s)
    w))

> (capi '("a aa" "Bb b"))
("A aa" "Bb b")

```

Решение 4.422.3

```

(defun capi (w)
  (loop for s in w do
    (setf (char s 0) (char-upcase (char s 0)))
    collect s))

> (capi '("a aa" "Bb b"))
("A aa" "Bb b")

```

Решение 4.422.4

```

(defun capi (w)
  (loop for s in w collect
    (concatenate 'string
      (string (char-upcase (char s 0)))
      (subseq s 1))))

> (capi '("a aa" "Bb b"))
("A aa" "Bb b")

```

Задача 4.423 cnt-elms.lisp

Определить функцию, которая подсчитывает количество каждого элемента в списке.

Решение 4.423.1

```

(defun cnt-elms (w
  &aux
    (v (flatten w))
    (z (remove-duplicates v)))
  (loop for a in z collect (list a (count a v))))

```

```
(defun flatten (w &optional acc)
  (cond ((null w) acc)
        ((atom w) (cons w acc))
        ((flatten (car w) (flat (cdr w) acc))))))

> (cnt-elms '(((2) -3 (4) 0 -2) (-4 0 2) ((-3) (3) -4 3)))
((4 1) (-2 1) (0 2) (2 2) (-3 2) (-4 2) (3 2))
```

Решение 4.423.2

```
(defun cnt-elms (w
  &aux
    (v (flatten w))
    (z (remove-duplicates v)))
  (loop for a in z collect (list a (count a v))))

(defun flatten (w)
  (loop for a in w
    if (listp a) append (flatten a)
    else collect a))

> (cnt-elms '(((2) -3 (4) 0 -2) (-4 0 2) ((-3) (3) -4 3)))
((4 1) (-2 1) (0 2) (2 2) (-3 2) (-4 2) (3 2))
```

Решение 4.423.3

```
(defun cnt-elms (w
  &aux
    (v (flatten w))
    (z (remove-duplicates v)))
  (loop for a in z collect (list a (count a v))))

(defun flatten (w)
  (mapcan #'(lambda (a) (if (atom a) `(,a) (flatten a)))
    w))

> (cnt-elms '(((2) -3 (4) 0 -2) (-4 0 2) ((-3) (3) -4 3)))
((4 1) (-2 1) (0 2) (2 2) (-3 2) (-4 2) (3 2))
```

Решение 4.423.4

```
(defun cnt-elms (w
  &aux
    (v (flatten w))
    (z (remove-duplicates v)))
  (loop for a in z collect (list a (count a v))))

(defun flatten (w)
  (typecase w
    (cons (mapcan #'flatten w))
    (t `(,w))))
```



```
> (cnt-elms '(((2) -3 (4) 0 -2) (-4 0 2) ((-3) (3) -4 3)))
((4 1) (-2 1) (0 2) (2 2) (-3 2) (-4 2) (3 2))
```

Решение 4.423.5

```
(defun cnt-elms (w
  &aux
    (v (flatten w))
    (z (remove-duplicates v)))
  (loop for a in z collect (list a (count a v))))

(defun flatten (w) (if (atom w) `(,w) (mapcan #'flatten w)))

> (cnt-elms '(((2) -3 (4) 0 -2) (-4 0 2) ((-3) (3) -4 3)))
((4 1) (-2 1) (0 2) (2 2) (-3 2) (-4 2) (3 2))
```

Задача 4.424 <<<<.lisp

Определить функцию, возвращающую список с элементами, перечисленными в обратном порядке.

Решение 4.424.1

```
(defun <<<< (w &optional ac)
  (if w (<<<< (cdr w) (cons (car w) ac)) ac))

> (<<<< '(1 2 3))
(3 2 1)
```

Задача 4.425 *symsb.lisp*

Определить функцию, которая из исходного списка формирует список, содержащий только символьные атомы.

Решение 4.425.1

```
(defun symsb (w &aux (a (car w)) (d (cdr w)))
  (when w (if (symbolp a) (cons a (symsb d)) (symsb d))))

> (symsb '(a 1 b 2 c 3))
(A B C)
```

Задача 4.426 *deep-reverse.lisp*

Определить функцию, которая возвращает список, содержащий элементы исходного списка (в том числе входящие в подспски), перечисленные в обратном порядке.

Решение 4.426.1

```
(defun deep-reverse (w &optional ac &aux (a (car w)) (d (cdr w)))
```

```
(cond ((null w) ac)
      ((atom a) (deep-reverse d (cons a ac)))
      ((deep-reverse d (cons (deep-reverse a) ac)))))
```

```
> (deep-reverse '(e ((2 d 1) c b) a))
(A (B C (1 D 2)) E)
```

Решение 4.426.2

```
(defun deep-reverse (w &optional ac &aux (a (car w)))
  (if w (deep-reverse (cdr w)
                      (cons (if (atom a) a
                                (deep-reverse a))
                            ac))
      ac))
```

```
> (deep-reverse '(e ((2 d 1) c b) a))
(A (B C (1 D 2)) E)
```

Решение 4.426.3

```
(defun deep-reverse (w)
  (loop for a in (reverse w)
        collect (if (atom a) a (deep-reverse a))))
```

```
> (deep-reverse '(e ((2 d 1) c b) a))
(A (B C (1 D 2)) E)
```

Решение 4.426.4

```
(defun deep-reverse (w)
  (mapcar #'(lambda (a) (if (atom a) a
                            (deep-reverse a)))
          (reverse w)))
```

```
> (deep-reverse '(e ((2 d 1) c b) a))
(A (B C (1 D 2)) E)
```

Задача 4.427 abstract-min.lisp

Дан список чисел. Вычесть из элементов списка его минимальный элемент.

Решение 4.427.1

```
(defun abstract-min (w &aux (m (reduce #'min w)))
  (mapcar #'(lambda (a) (- a m)) w))
```

```
> (abstract-min '(1 2 3 4 5))
(0 1 2 3 4)
```

Решение 4.427.2

```
(defun abstract-min (w)
  (loop for a in w collect (- a (reduce #'min w))))

> (abstract-min '(1 2 3 4 5))
(0 1 2 3 4)
```

Решение 4.427.3

```
(defun abstract-min (w &optional (m (reduce #'min w)))
  (when w (cons (- (car w) m) (abstract-min (cdr w) m))))

> (abstract-min '(1 2 3 4 5))
(0 1 2 3 4)
```

Задача 4.428 revv.lisp

Определить функцию, которая в каждом слове исходного текста меняет порядок букв на обратный и порядок слов на обратный.

Решение 4.428.1

```
(defun revv (w)
  (mapcar #'(lambda (a) (mapcar #'reverse (reverse a))) w))

> (revv '(("ab" "cd" ".") ("ef" "gh" ".")))
(("." "dc" "ba") (". " "hg" "fe"))
```

Решение 4.428.2

```
(defun revv (w)
  (loop for a in w
        collect (loop for e in (reverse a) collect (reverse e))))

> (revv '(("ab" "cd" ".") ("ef" "gh" ".")))
(("." "dc" "ba") (". " "hg" "fe"))
```

Решение 4.428.3

```
(defun revv (w &optional ac)
  (cond ((null w) ac)
        ((revv (cdr w) (cons (reverse (car w))
                              ac)))))

> (revv '(("ab" "cd" ".") ("ef" "gh" ".")))
(("." "dc" "ba") (". " "hg" "fe"))
```

Решение 4.428.4

```
(defun revv (w &optional ac)
  (if w (revv (cdr w) (cons (reverse (car w)) ac)) ac))

> (revv '(("ab" "cd" ".") ("ef" "gh" ".")))
(("." "dc" "ba") (". " "hg" "fe"))
```

```
(( "." "dc" "ba") ( "." "hg" "fe"))
```

Задача 4.429 glue-sort-count-deep.lisp

Определить функцию, возвращающую список вида: первый элемент — атом, появляющийся в исходном сложном списке один раз, второй элемент — атом, появляющийся два раза и т.д. Если есть несколько атомов, появляющихся одинаковое количество раз, то объединить их в список.

Решение 4.429.1

```
(defun flatten (w &optional acc)
  (cond ((null w) acc)
        ((atom w) (cons w acc))
        ((flatten (car w) (flat (cdr w) acc)))))

(defun count-deep (w &aux (v (flatten w)))
  (loop for a in (remove-duplicates v)
        collect (list a (count a v))))

(defun sort-count (w)
  (sort (count-deep w) #'< :key #'cadr))

(defun glue (w &optional (b (cadar w)) ac &aux (a (car w)))
  (cond ((null w) ac)
        ((equal b (cadadr w)) (glue (cdr w) b (cons (car a) ac)))
        ((cons (if ac (reverse (cons (car a) ac)) (car a))
                 (glue (cdr w) (cadadr a) nil)))))

(defun glue-sort-count-deep (w)
  (glue (sort-count w)))

> (glue-sort-count-deep '(a b c (a ((b c) c) c) c))
((A B) C)
> (glue-sort-count-deep '(z b c (a ((b c) c)
```

Решение 4.429.2

```
(defun flatten (w)
  (loop for a in w
        if (listp a) append (flatten a)
        else collect a))

(defun sort-count (w &aux (v (flatten w)))
  (sort (loop for a in (remove-duplicates v)
              collect `(,a ,(count a v)))
        #'< :key #'cadr))

(defun glue (w &optional (b (cadar w)) ac)
  (cond ((null w) ac)
        ((equal b (cadadr w)) (glue (cdr w) b (cons (caar w) ac)))
        ((cons (if ac (reverse (cons (caar w) ac)) (caar w))
```

```

      (glue (cdr w) (caddar w) nil))))))

(defun glue-sort-count-deep (w) (glue (sort-count w)))

> (glue-sort-count-deep '(a b c (a ((b c) c) c) c))
((A B) C)
> (glue-sort-count-deep '(z b c (a ((b c) c) c) c))
((Z A) B C)

```

Решение 4.429.3

```

(defun flatten (w)
  (mapcan #'(lambda (a) (if (atom a) `(,a) (flatten a))) w))

(defun sort-count (w &aux (v (flatten w)))
  (sort (loop for a in (remove-duplicates v)
             collect `(,a ,(count a v)))
        #'< :key #'cadr))

(defun glue (w &optional (b (cadar w)) ac &aux (a (car w)))
  (cond ((null w) ac)
        ((equal b (cadadr w)) (glue (cdr w) b (cons (car a) ac)))
        ((cons (if ac (reverse (cons (car a) ac)) (car a))
                (glue (cdr w) (caddr a) nil)))))

(defun glue-sort-count-deep (w)
  (glue (sort-count w)))

> (glue-sort-count-deep '(a b c (a ((b c) c) c) c))
((A B) C)
> (glue-sort-count-deep '(z b c (a ((b c) c) c) c))
((Z A) B C)

```

Решение 4.429.4

```

(defun flatten (w)
  (typecase w
    (cons (mapcan #'flatten w))
    (t `(,w))))

(defun sort-count (w &aux (v (flatten w)))
  (sort (loop for a in (remove-duplicates v)
             collect `(,a ,(count a v)))
        #'< :key #'cadr))

(defun glue (w &optional (b (cadadr w)) ac &aux (a (car w)))
  (cond ((null w) ac)
        ((equal b (cadadr w)) (glue (cdr w) b (cons (car a) ac)))
        ((cons (if ac (reverse (cons (car a) ac)) (car a))
                (glue (cdr w) (caddr a) nil)))))

(defun glue-sort-count-deep (w)
  (glue (sort-count w)))

```

```
> (glue-sort-count-deep '(a b c (a ((b c) c) c) c))
((A B) C)
> (glue-sort-count-deep '(z b c (a ((b c) c) c) c))
((Z A) B C)
```

Решение 4.429.5

```
(defun flatten (w)
  (if (atom w) `(),w) (mapcan #'flatten w)))

(defun sort-count (w &aux (v (flatten w)))
  (sort (loop for a in (remove-duplicates v)
             collect `(),a , (count a v))
        #'< :key #'cadr))

(defun glue (w &optional (b (cadar w)) ac &aux (a (car w)))
  (cond ((null w) ac)
        ((equal b (cadadr w)) (glue (cdr w) b (cons (car a) ac)))
        ((cons (if ac (reverse (cons (car a) ac)) (car a))
                (glue (cdr w) (caddr a) nil)))))

(defun glue-sort-count-deep (w)
  (glue (sort-count w)))

> (glue-sort-count-deep '(a b c (a ((b c) c) c) c))
((A B) C)
> (glue-sort-count-deep '(z b c (a ((b c) c) c) c))
((Z A) B C)
```

Задача 4.430 plus-minus-zero.lisp

Определить функцию, которая по списку с подписками возвращает список из положительных числовых элементов, список отрицательных числовых элементов и количество нулевых элементов.

Решение 4.430.1

```
(defun plus-minus-zero (w)
  (loop for a in (flatten w)
        if (plusp a) collect a into p
        else if (minusp a) collect a into m
        else count a into z
        finally return (values p m z)))

(defun flatten (w &optional acc)
  (cond ((null w) acc)
        ((atom w) (cons w acc))
        ((flatten (car w) (flat (cdr w) acc)))))

> (plus-minus-zero '(1 (2 3) 0 0 (((-4))) ((-6))))
```

```
(1 2 3)
(-4 -6)
2
```

Решение 4.430.2

```
(defun plus-minus-zero (w)
  (loop for a in (flatten w)
        if (plussp a) collect a into p
        else if (minusp a) collect a into m
        else count a into z
        finally return (values p m z)))

(defun flatten (w)
  (loop for a in w
        if (listp a) append (flatten a)
        else collect a))

> (plus-minus-zero '(1 (2 3) 0 0 (((-4))) ((-6))))
(1 2 3)
(-4 -6)
2
```

Решение 4.430.3

```
(defun plus-minus-zero (w)
  (loop for a in (flatten w)
        if (plussp a) collect a into p
        else if (minusp a) collect a into m
        else count a into z
        finally return (values p m z)))

(defun flatten (w)
  (if (atom w) `(),w) (mapcan #'flatten w)))

> (plus-minus-zero '(1 (2 3) 0 0 (((-4))) ((-6))))
(1 2 3)
(-4 -6)
2
```

Задача 4.431 *xenna.lisp*

Определить функцию, которая список `(a1 a2 a3 ...aN)` преобразует в сложный список `((((aN)...a3) a2) a1)`.

Решение 4.431.1

```
(defun xenna (w &aux (d (cdr w))) (if d `((, (xenna d) , (car w)) w))

> (xenna '(a1 a2 a3 a4))
(((A4) A3) A2) A1)
```

Задача 4.432 pair.lisp

Определить функцию, которая разбивает список (a b c d e) на пары ((a b) (c d) (e)) с использованием циклов.

Решение 4.432.1

```
(defun pair (w &aux (b (car w)))
  (loop for a in (cdr w)
        when b collect (list b a) into z
        do (if b (setf b nil) (setf b a))
        finally (return (if b (append z `((,b))) z))))
```

```
> (pair '(a b c d e f))
((A B) (C D) (E F))
> (pair '(a b c d e))
((A B) (C D) (E))
```

Решение 4.432.2

```
(defun pair (w)
  (loop for (a b) on w by #'cddr
        if b collect (list a b)
        else collect (list a)))
```

```
> (pair '(a b c d e f))
((A B) (C D) (E F))
> (pair '(a b c d e))
((A B) (C D) (E))
```

Решение 4.432.3

```
(defun pair (w)
  (loop for (a b) on w by #'cddr
        collect (if b (list a b) (list a))))
```

```
> (pair '(a b c d e f))
((A B) (C D) (E F))
> (pair '(a b c d e))
((A B) (C D) (E))
```

Решение 4.432.4

```
(defun pair (w)
  (loop for (a b) on w by #'cddr
        collect (if b `((,a ,b)) `((,a)))))
```

```
> (pair '(a b c d e f))
((A B) (C D) (E F))
> (pair '(a b c d e))
((A B) (C D) (E))
```


Задача 4.433 pack.lisp

Определить функцию, которая «упаковывает» повторяющиеся элементы списка в подписки следующего формата: ("packed" <number_of_elements> <packed_element>). Например, список (7 7 7 "a" "b" "b" "b" "b" "b") должен быть преобразован в список (("packed" 3 7) "a" ("packed" 5 "b")).

Решение 4.433.1

```
(defun pack (w &optional z (n 1))
  (cond ((null w) (when z (if (= n 1) `(:z) `(("packed" ,n ,z)))))
        ((equalp (car w) z) (pack (cdr w) z (1+ n)))
        (z (cons (if (= n 1) z `("packed" ,n ,z))
                  (pack (cdr w) (car w) 1)))
        ((pack (cdr w) (car w) 1))))

> (pack '(7 7 7 "a" "b" "b" "b" "b" "b"))
(("packed" 3 7) "a" ("packed" 5 "b"))
> (pack '("a"))
("a")
> (pack '())
NIL
```

Задача 4.434 abba.lisp

Дан список слов. Определить функцию, возвращающую слова, которые начинаются и заканчиваются на одну и ту же букву.

Решение 4.434.1

```
(defun abba (w)
  (loop for a in w when
        (char-equal (char a 0)
                    (char a (1- (length a))))
    collect a))

> (abba '("asdf" "asda"))
("asda")
```

Решение 4.434.2

```
(defun abba (w)
  (loop for a in w when
        (char-equal (char a 0)
                    (char (reverse a) 0))
    collect a))

> (abba '("asdf" "asda"))
("asda")
```

Решение 4.434.3

```
(defun abba (w &aux (a (car w)))
  (cond ((null w) nil)
        ((char-equal (char a 0)
                      (char (reverse a) 0))
         (cons a (abba (cdr w))))
        ((abba (cdr w)))))

> (abba '("asdf" "asda"))
("asda")
```

Решение 4.434.4

```
(defun abba (w &aux (a (car w)))
  (when w (if (char-equal (char a 0)
                          (char (reverse a) 0))
              (cons a (abba (cdr w)))
              (abba (cdr w)))))

> (abba '("asdf" "asda"))
("asda")
```

Решение 4.434.5

```
(defun abba (w)
  (mapcan #'(lambda (a)
              (when (char-equal (char a 0)
                                (char (reverse a) 0))
                a))
          w))

> (abba '("asdf" "asda"))
("asda")
```

Задача 4.435 maxx.lisp

Дан список, состоящий из подсписков. Определить функцию, которая возвращает подсписок, сумма числовых элементов которого максимальна.

Решение 4.435.1

```
(defun maxx (w
              &optional
              (v (car w))
              (b (reduce #'(lambda (x y) (+ x y)) (remove-if-not #'numberp w))))
  &aux
  (m (reduce #'(lambda (x y) (+ x y)) (remove-if-not #'numberp (car w)))))
  (cond ((null w) v)
        ((> m b) (maxx (cdr w) (car w) m))
        ((maxx (cdr w) v b)))))

> (maxx '((a 1 2) (b 2 3) (c 3 4)))
```

```
(C 3 4)
> (maxx '((a 1 2) () ()))
(A 1 2)
> (maxx '(() () ()))
NIL
```

Решение 4.435.2

```
(defun maxx (w
  &optional
  (v (car w))
  (b (reduce #' + (remove-if-not #'numberp v)))
  &aux
  (m (reduce #' + (remove-if-not #'numberp (car w)))))
  (if w (if (> m b) (maxx (cdr w) (car w) m) (maxx (cdr w) v b)) v))

> (maxx '((a 1 2) (b 2 3) (c 3 4)))
(C 3 4)
> (maxx '((a 1 2) () ()))
(A 1 2)
> (maxx '(() () ()))
NIL
```

Задача 4.436 maxx.lisp

Дан список, содержащий подсписки. Определить функцию, которая возвращает подсписок, сумма числовых элементов которого максимальна.

Решение 4.436.1

```
(defun maxxxx (w
  &aux
  (v (remove-if #'atom w))
  (m (loop for a in v maximize (reduce #' + a))))
  (loop for a in v
    when (= (reduce #' + a) m) return a))

> (maxxxx '((0 0 1) 3 4 (4 2) (1 2)))
(4 2)
```

Задача 4.437 maxxxx.lisp

Дан список, содержащий подсписки. Определить функцию, которая возвращает подсписки, сумма числовых элементов которого максимальна.

Решение 4.437.1

```
(defun maxxxx (w
  &aux
  (v (remove-if #'atom w))
  (m (loop for a in v maximize (reduce #' + a))))
```

```

(loop for a in v
      when (= (reduce #' + a) m) collect a))

> (maxxxx '((0 0 1) (3 3) 3 4 (4 2) (1 2)))
((3 3) (4 2))

```

Задача 4.438 sum-columns.lisp

Almost all programming languages provide now what made Lisp so "special" like multi paradigms, garbage collection, foreign function interfaces and all the usual features a modern programming language needs. But there is still one field where Lisp shines above all the others: Meta programming. Lisp is a programmable programming language.

Sofiane Akermoun

Дана квадратная матрица целых чисел. Определить функцию, возвращающую список, элементами которого являются суммы столбцов данной матрицы.

Решение 4.438.1

```

(defun sum-columns (w &optional (h 0) (j 0) (k 0))
  (cond ((null w) (list h j k))
        (t (sum-columns (cdr w)
                          (+ h (caar w))
                          (+ j (cadar w))
                          (+ k (caddar w))))))

> (sum-columns '((1 4 6) (2 8 1)))
(3 12 7)

```

Решение 4.438.2

```

(defun sum-columns (w &optional (h 0) (j 0) (k 0))
  (if w (sum-columns (cdr w)
                      (+ h (caar w))
                      (+ j (cadar w))
                      (+ k (caddar w)))
        (list h j k)))

> (sum-columns '((1 4 6) (2 8 1)))
(3 12 7)

```

Решение 4.438.3

```

(defun sum-columns (w &optional (h 0) (j 0) (k 0) &aux (a (car w)))
  (if w (sum (cdr w)
              (+ h (first a)))
        (list h j k)))

```

```

      (+ j (second a))
      (+ k (third a)))
(list h j k)))

```

```

> (sum-columns '((1 4 6) (2 8 1)))
(3 12 7)

```

Решение 4.438.4

```

(defun sum-columns (w)
  (loop for (h j k) in w
        sum h into hs
        sum j into js
        sum k into ks
        finally (return (list hs js ks))))

```

```

> (sum-columns '((1 4 6) (2 8 1)))
(3 12 7)

```

Решение 4.438.5

```

(defun sum-columns (w)
  (loop for (h j k) in w
        sum h into hs
        sum j into js
        sum k into ks
        finally (return `(,hs ,js ,ks))))

```

```

> (sum-columns '((1 4 6) (2 8 1)))
(3 12 7)

```

Решение 4.438.6

```

(defun sum-coulumns (w &optional (ac '(0 0 0)))
  (cond ((null w) ac)
        (t (sum-coulumns (cdr w)
                           (list (+ (first ac) (caar w))
                                   (+ (second ac) (cadar w))
                                   (+ (third ac) (caddar w)))))))

```

```

> (sum-columns '((1 4 6) (2 8 1)))
(3 12 7)

```

Решение 4.438.7

```

(defun sum-coulumns (w)
  (loop for a in (transpose w)
        collect (reduce #'+ a)))

```

```

(defun transpose (m)
  (apply #'mapcar #'list m))

```

```

> (sum-coulumns '((1 4 6) (2 8 1) (10 100 1000)))

```

```
(13 112 1007)
```

Решение 4.438.8

```
(defun sum-coulumns (w)
  (loop for a in (apply #'mapcar #'list w)
        collect (reduce #'+ a)))

> (sum-coulumns '((1 4 6) (2 8 1) (10 100 1000)))
(13 112 1007)
```

Решение 4.438.8

```
(defun sum-coulumns (w)
  (loop for a in (transpose w) collect (reduce #'+ a)))

(defun transpose (v)
  (when (car v) (cons (cr v #'caar)
                      (transpose (cr v #'cdar))))))

(defun cr (v c)
  (when v (cons (funcall c v) (cr (cdr v) c))))

> (sum-coulumns '((1 4 6) (2 8 1) (10 100 1000)))
(13 112 1007)
```

Решение 4.438.9

```
(defun sum-coulumns (w) (-sum (transpose w)))

(defun -sum (w)
  (cond ((null w) nil)
        (t (cons (apply #'+ (car w)) (-sum (cdr w))))))

(defun transpose (v)
  (when (car v) (cons (cr v #'caar)
                      (transpose (cr v #'cdar))))))

(defun cr (v c)
  (when v (cons (funcall c v) (cr (cdr v) c))))

> (sum-coulumns '((1 4 6) (2 8 1) (10 100 1000)))
(13 112 1007)
```

Решение 4.438.10

```
(defun sum-coulumns (w) (-sum (transpose w)))

(defun -sum (w)
  (cond ((null w) nil)
        (t (cons (apply #'+ (car w)) (-sum (cdr w))))))

(defun transpose (v)
```

```

    (when (car v) (cons (cr v #'caar)
                        (transpose (cr v #'cdar)))))

(defun cr (v c)
  (when v (cons (funcall c v) (cr (cdr v) c))))

> (sum-coulumns '((1 4 6) (2 8 1) (10 100 1000)))
(13 112 1007)

```

Решение 4.438.11

```

(defun sum-coulumns (w) (-sum (transpose w)))

(defun -sum (w)
  (cond ((null w) nil)
        (t (cons (reduce #'+ (car w)) (-sum (cdr w))))))

(defun transpose (m) (apply #'mapcar #'list m))

> (sum-coulumns '((1 4 6) (2 8 1) (10 100 1000)))
(13 112 1007)

```

Решение 4.438.12

```

(defun sum-coulumns (w) (-sum (transpose w)))

(defun -sum (w)
  (when w (cons (reduce #'+ (car w)) (-sum (cdr w)))))

(defun transpose (m)
  (apply #'mapcar #'list m))

> (sum-coulumns '((1 4 6) (2 8 1) (10 100 1000)))
(13 112 1007)

```

Задача 4.439 maxx.lisp

Дана прямоугольная матрица. Найти в ней строку, имеющую наибольшую сумму. Вывести значение суммы и эту строку.

Решение 4.439.1

```

(defun maxx (w &aux (m (loop for a in w maximize (reduce #'+ a))))
  (loop for a in w when (= (reduce #'+ a) m) return (values m a)))

> (maxx '((0 0 1) (0 0 2) (0 0 3)))
3
(0 0 3)

```

Решение 4.439.2

```

(defun maxx (w

```

```

      &aux
      (m (reduce #'max w
                  :key #'(lambda (a)
                          (reduce #'+ a))))
    (loop for a in w
          when (= (reduce #'+ a) m) return (values m a)))
> (maxx '((0 0 1) (0 1 2) (0 0 3)))
3
(0 1 2)

```

Решение 4.439.3

```

(defun maxx (w
              &aux
              (m (reduce #'max w
                        :key #'(lambda (a)
                                (eval (cons '+ a))))))
  (loop for a in w
        when (= (eval (cons '+ a)) m) return (values m a)))
> (maxx '((0 0 1) (0 1 2) (0 0 3)))
3
(0 1 2)

```

Задача 4.440 maxx.lisp

С середины XVI века по начало XIX века между Россией и Швецией было 10 войн.

~

Между Россией и Османской империей было 12 войн за 241 год. В среднем, одну Русско-турецкую войну от другой отделяло 19 лет.

~

Соловьев насчитывает с 1240 года по 1462 год (за 222 года) — 200 войн и нашествий. С XIV века по XX-ый (за 525 лет) Сухотин насчитывает 329 лет войны. Россия провела две трети своей жизни.

Иван Ильин

Дана прямоугольная матрица. Найти в ней строки, имеющие наибольшую сумму. Вывести значение суммы и эти строки.

Решение 4.440.1

```

(defun maxx (w &aux (m (loop for a in w maximize (reduce #'+ a))))
  (loop for a in w
        when (= (reduce #'+ a) m) collect a into v

```



```
finally (return (values m v))))
```

```
> (maxxx '(0 0 1) (0 1 2) (0 0 3))
3
((0 1 2) (0 0 3))
```

Решение 4.440.2

```
(defun maxxx (w
              &aux
              (m (reduce #'max w
                        :key #'(lambda (a)
                               (reduce #'+ a)))))
  (loop for a in w
        when (= (reduce #'+ a) m) collect a into v
        finally (return (values m v))))
```

```
> (maxxx '(0 0 1) (0 1 2) (0 0 3))
3
((0 1 2) (0 0 3))
```

Решение 4.440.3

```
(defun maxxx (w
              &aux
              (m (reduce #'max w
                        :key #'(lambda (a)
                               (eval (cons '+ a))))))
  (loop for a in w
        when (= (eval (cons '+ a)) m) collect a into v
        finally (return (values m v))))
```

```
> (maxxx '(0 0 1) (0 1 2) (0 0 3))
3
((0 1 2) (0 0 3))
```

Решение 4.440.4

```
(defun maxxx (w &aux (m (mx w))) (list m (maxx w m)))

(defun maxx (w m)
  (cond ((null w) nil)
        ((= (sum (car w)) m) (cons (car w) (maxx (cdr w) m)))
        (t (maxx (cdr w) m))))

(defun mx (w
           &optional (m (sum (car w)))
           &aux (n (sum (car w))))
  (cond ((null w) m)
        ((> n m) (mx (cdr w) n))
        (t (mx (cdr w) m))))
```

```
(defun sum (w) (if w (+ (car w) (sum (cdr w))) 0))
```

```
> (maxxx '((0 0 1) (0 1 2) (0 0 3)))
(3 ((0 1 2) (0 0 3)))
```

Решение 4.440.5

```
(defun maxxx (w &aux (m (mx w))) (progn (princ m) (maxx w m)))
```

```
(defun maxx (w m)
  (cond ((null w) nil)
        ((= (sum (car w)) m) (cons (car w) (maxx (cdr w) m)))
        (t (maxx (cdr w) m))))
```

```
(defun mx (w
           &optional (m (sum (car w)))
           &aux (n (sum (car w))))
  (cond ((null w) m)
        ((> n m) (mx (cdr w) n))
        (t (mx (cdr w) m))))
```

```
(defun sum (w) (if w (+ (car w) (sum (cdr w))) 0))
```

```
> (maxxx '((0 0 1) (0 1 2) (0 0 3)))
3
((0 1 2) (0 0 3))
```

Задача 4.441 level-sum.lisp

Определить функцию, которая возвращает сумму чисел на каждом уровне списка.

Решение 4.441.1

```
(defun dive-elms (n w &aux (a (car w)) (d (cdr w)))
  (cond ((null w) nil)
        ((= n 0) w)
        ((atom a) (dive-elms n d))
        ((nconc (dive-elms (1- n) a) (dive-elms n d)))))
```

```
(defun max-depth (w)
  (if (atom w) 0
      (max (1+ (max-depth (car w)))
            (max-depth (cdr w)))))
```

```
(defun level-sum (w &aux (n (1- (max-depth w))))
  (loop for a from 0 to n
        collect (list a (loop for a in (dive-elms a w)
                               when (numberp a) sum a))))
```

```
> (level-sum '((12 (1 1) 2) (11 4)))
((0 0) (1 29) (2 2))
```

Решение 4.441.2

```

(defun dive-elms (n w &aux (a (car w)) (d (cdr w)))
  (cond ((null w) nil)
        ((= n 0) w)
        ((atom a) (dive-elms n d))
        ((nconc (dive-elms (1- n) a) (dive-elms n d)))))

(defun max-depth (w)
  (if (atom w) 0
      (max (1+ (max-depth (car w)))
            (max-depth (cdr w)))))

(defun level-sum (w &optional (n (1- (max-depth w))))
  (cond ((zerop n) (list n (loop for a in (dive-elms n w)
                                   when (numberp a) sum a)))
        ((cons (list n (loop for a in (dive-elms n w)
                                   when (numberp a) sum a))
                (level-sum w (1- n))))))

> (level-sum '((12 (1 1) 2) (11 4)))
((2 2) (1 29) 0 0)

```

Решение 4.441.3

```

(defun dive-elms (n w &aux (a (car w)) (d (cdr w)))
  (cond ((null w) nil)
        ((= n 0) w)
        ((atom a) (dive-elms n d))
        ((nconc (dive-elms (1- n) a) (dive-elms n d)))))

(defun max-depth (w)
  (if (atom w) 0
      (max (1+ (max-depth (car w)))
            (max-depth (cdr w)))))

(defun level-sum (w &optional (n (1- (max-depth w))))
  (cond ((zerop n) (list n (sum (dive-elms n w))))
        ((cons (list n (sum (dive-elms n w)))
                (level-sum w (1- n))))))

(defun sum (w)
  (cond ((null w) 0)
        ((numberp (car w)) (+ (car w) (sum (cdr w))))
        ((sum (cdr w)))))

> (level-sum '((12 (1 1) 2) (11 4)))
((2 2) (1 29) 0 0)

```

Решение 4.441.4

```

(defun dive-elms (n w &aux (a (car w)) (d (cdr w)))

```

```

(cond ((null w) nil)
      ((= n 0) w)
      ((atom a) (dive-elms n d))
      ((nconc (dive-elms (1- n) a) (dive-elms n d)))))

(defun max-depth (w)
  (if (atom w) 0
      (max (1+ (max-depth (car w)))
            (max-depth (cdr w)))))

(defun level-sum (w &optional (n (1- (max-depth w))))
  (cond ((zerop n) (list n (sum (dive-elms n w))))
        ((cons (list n (sum (dive-elms n w)))
                 (level-sum w (1- n))))))

(defun sum (w)
  (if w (+ (if (numberp (car w)) (car w) 0) (sum (cdr w))) 0))

> (level-sum '((12 (1 1) 2) (11 4)))
((2 2) (1 29) 0 0)

```

Решение 4.441.5

```

(defun dive-elms (n w &aux (a (car w)) (d (cdr w)))
  (cond ((null w) nil)
        ((= n 0) w)
        ((atom a) (dive-elms n d))
        ((append (dive-elms (1- n) a) (dive-elms n d)))))

(defun max-depth (w)
  (if (atom w) 0
      (max (1+ (max-depth (car w)))
            (max-depth (cdr w)))))

(defun level-sum (w &optional (n (1- (max-depth w))))
  (if (zerop n) (list n (sum (dive-elms n w)))
      (cons (list n (sum (dive-elms n w)))
              (level-sum w (1- n)))))

(defun sum (w)
  (if w (+ (if (numberp (car w)) (car w) 0) (sum (cdr w))) 0))

> (level-sum '((12 (1 1) 2) (11 4)))
((2 2) (1 29) 0 0)

```

Решение 4.441.6

```

(defun dive-elms (n w &aux (a (car w)) (d (cdr w)))
  (cond ((null w) nil)
        ((= n 0) w)
        ((atom a) (dive-elms n d))
        ((append (dive-elms (1- n) a) (dive-elms n d)))))

```

```

(defun max-depth (w)
  (if (atom w) 0
      (max (1+ (max-depth (car w)))
            (max-depth (cdr w))))))

(defun level-sum (w &optional (n (1- (max-depth w))))
  (if (zerop n) (list n (sum (dive-elms n w)))
      (cons (list n (sum (dive-elms n w)))
            (level-sum w (1- n)))))

(defun sum (w &aux (a (car w)))
  (if w (+ (if (numberp a) a 0) (sum (cdr w))) 0))

> (level-sum '((12 (1 1) 2) (11 4)))
((2 2) (1 29) 0 0)

```

Решение 4.441.7

```

(defun level-sum (w &optional (n (1- (max-depth w))))
  (if (zerop n) `(,n ,(sum (dive-elms n w)))
      (cons `(,n ,(sum (dive-elms n w)))
            (level-sum w (1- n)))))

(defun max-depth (w)
  (if (atom w) 0 (max (1+ (max-depth (car w)))
                      (max-depth (cdr w)))))

(defun sum (w &aux (a (car w)))
  (if w (+ (if (numberp a) a 0) (sum (cdr w))) 0))

(defun dive-elms (n w &aux (a (car w)) (d (cdr w)))
  (when w (cond ((= n 0) w)
                ((atom a) (dive-elms n d))
                ((append (dive-elms (1- n) a)
                          (dive-elms n d)))))

> (level-sum '((12 (1 1) 2) (11 4)))
((2 2) (1 29) 0 0)

```

Решение 4.441.8

```

(defun level-sum (w &optional (n (1- (max-depth w))))
  (cond ((zerop n)
        (format t "lvl: ~D summ:~D~%" n (sum (dive-elms n w)))
        ((progn (format t "lvl: ~D summ:~D~%" n (sum (dive-elms n w)))
                (level-sum w (1- n)))))

(defun max-depth (w)
  (if (atom w) 0 (max (1+ (max-depth (car w)))
                      (max-depth (cdr w)))))

(defun sum (w &aux (a (car w)))
  (if w (+ (if (numberp a) a 0) (sum (cdr w))) 0))

```

```
(defun dive-elms (n w &aux (a (car w)) (d (cdr w)))
  (when w (cond ((= n 0) w)
                ((atom a) (dive-elms n d))
                ((append (dive-elms (1- n) a)
                          (dive-elms n d))))))
```

```
> (level-sum '((12 (1 1) 2) (11 4)))
lvl: 2 summ:2
lvl: 1 summ:29
lvl: 0 summ:0
NIL
```

Решение 4.441.9

```
(defun level-sum (w &optional (n (1- (max-depth w))))
  (unless (minusp n)
    (progn (format t "lvl: ~D sum: ~D~%" n (sum (dive-elms n w)))
           (level-sum w (1- n)))))
```

```
(defun max-depth (w)
  (if (atom w) 0 (max (1+ (max-depth (car w)))
                      (max-depth (cdr w)))))
```

```
(defun sum (w &aux (a (car w)))
  (if w (+ (if (numberp a) a 0) (sum (cdr w))) 0))
```

```
(defun dive-elms (n w &aux (a (car w)) (d (cdr w)))
  (when w (cond ((zerop n) w)
                ((atom a) (dive-elms n d))
                ((append (dive-elms (1- n) a)
                          (dive-elms n d))))))
```

```
> (level-sum '((12 (1 1) 2) (11 4)))
lvl: 2 sum: 2
lvl: 1 sum: 29
lvl: 0 sum: 0
NIL
```

Решение 4.441.10

```
(defun level-sum (w) (lev-sum (flat-mark w)))
```

```
(defun lev-sum (w &optional (m (reduce #'max w :key #'car)))
  (loop for a from 0 to m collect
    (list a (loop for e in (remove-if-not
                          #'(lambda (b) (= (car b) a))
                          w)
          sum (cadr e)))))
```

```
(defun flat-mark (w &optional (n -1) ac)
  (cond ((null w) ac)
        ((atom w) (cons (list n w) ac)))
```

```

      ((flat-mark (car w) (1+ n) (flat-mark (cdr w) n ac))))
> (level-sum '((12 (1 1) 2) (11 4)))
((0 0) (1 29) (2 2))

```

Решение 4.441.11

```

(defun level-sum (w) (lev-sum (flat-mark w)))

(defun lev-sum (w &optional (m (reduce #'max w :key #'car)))
  (loop for a from 0 to m collect
    (list a (loop for e in w
      when (= (car e) a) sum (cadr e)))))

(defun flat-mark (w &optional (n -1) ac)
  (cond ((null w) ac)
        ((atom w) (cons (list n w) ac))
        ((flat-mark (car w) (1+ n) (flat-mark (cdr w) n ac)))))

> (level-sum '((12 (1 1) 2) (11 4)))
((0 0) (1 29) (2 2))

```

Решение 4.441.12

```

(defun level-sum (w) (lev-sum (flat-mark w)))

(defun lev-sum (w &optional (m (reduce #'max w :key #'car)))
  (loop for a from 0 to m do
    (format t "lvl: ~D sum: ~D~%"
      a (loop for e in w
        when (= (car e) a) sum (cadr e)))))

(defun flat-mark (w &optional (n -1) ac)
  (cond ((null w) ac)
        ((atom w) (cons (list n w) ac))
        ((flat-mark (car w) (1+ n) (flat-mark (cdr w) n ac)))))

> (level-sum '((12 (1 1) 2) (11 4)))
lvl: 0 sum: 0
lvl: 1 sum: 29
lvl: 2 sum: 2
NIL

```

Решение 4.441.13

```

(defun level-sum (w) (lev-sum (flat-mark w)))

(defun lev-sum (w &optional (m (reduce #'max w :key #'car)))
  (unless (minusp m)
    (progn (format t "lvl: ~D sum: ~D~%"
      m (sum w m))
      (lev-sum w (1- m)))))

```

```

(defun sum (w n &aux (a (car w)))
  (cond ((null w) 0)
        ((= (car a) n) (+ (cadr a) (sum (cdr w) n)))
        ((sum (cdr w) n))))

(defun flat-mark (w &optional (n -1) ac)
  (cond ((null w) ac)
        ((atom w) (cons (list n w) ac))
        ((flat-mark (car w) (1+ n) (flat-mark (cdr w) n ac)))))

> (level-sum '((12 (1 1) 2) (11 4)))
lvl: 2 sum: 2
lvl: 1 sum: 29
lvl: 0 sum: 0
NIL

```

Задача 4.442 drop-mid.lisp

(Где-то звенит трамвай, глаза слипаются, солдаты возвращаются с песнями из борделя, дождь — единственное, что напоминает Гегеля.) Истина заключается в том, что истины не существует. Это не освобождает от ответственности, но ровно наоборот: этика — тот же вакуум, заполняемый человеческим поведением, практически постоянно; тот же, если угодно, космос. И боги любят добро не за его глаза, но потому что, не будь добра, они бы не существовали. И они, в свою очередь, заполняют вакуум. И может быть, даже более систематически, нежели мы: ибо на нас нельзя рассчитывать. Хотя нас гораздо больше, чем когда бы то ни было, мы — не в Греции: нас губит низкая облачность и, как сказано выше, дождь.

Иосиф Бродский

Задан список, длина которого кратна трем. Получить список в котором удалена вторая треть.

Решение 4.442.1

```

(defun drop-mid (w &optional (m 1) (n (/ (length w) 3)))
  (cond ((null w) nil)
        ((or (<= m n) (> m (* n 2)))
         (cons (car w) (drop-mid (cdr w) (1+ m) n)))
        ((drop-mid (cdr w) (1+ m) n))))

> (drop-mid '(1 2 3 4 5 6))
(1 2 5 6)

```


Решение 4.442.2

```
(defun drop-mid (w &aux (n (/ (length w) 3)))
  (loop for a in w for b upfrom 1
    when (or (<= b n) (> b (* n 2)))
    collect a))

> (drop-mid '(1 2 3 4 5 6))
(1 2 5 6)
```

Решение 4.442.3

```
(defun drop-mid (w &aux (n (/ (length w) 3)))
  (append (subseq w 0 n) (subseq w (* n 2))))

> (drop-mid '(1 2 3 4 5 6))
(1 2 5 6)
```

Задача 4.443 *drop-empty.lisp*

C++ is a badly designed and ugly language.
It would be a shame to use it in Emacs.

Richard Stallman

Определить функцию, которая удаляет пустые строки из заданного списка.

Решение 4.443.1

```
(defun drop-empty (w &aux (a (car w)))
  (cond ((null w) nil)
        ((or (zerop (length a)) (empty a)) (drop-empty (cdr w)))
        ((cons a (drop-empty (cdr w))))))

(defun empty (s) (loop for a across s always (char= a #\Space)))

> (drop-empty '(" " "Hello" ", " "" "World!"))
("Hello" ", " "World!")
```

Решение 4.443.2

```
(defun drop-empty (w)
  (loop for a in w
    unless (or (zerop (length a)) (empty a)) collect a))

(defun empty (s) (loop for a across s always (char= a #\Space)))

> (drop-empty '(" " "Hello" ", " "" "World!"))
("Hello" ", " "World!")
```

Решение 4.443.3

```
(defun drop-empty (w)
  (remove-if #'(lambda (a) (or (zerop (length a)) (empty a))) w))

(defun empty (s) (loop for a across s always (char= a #\Space)))
(defun sum-nrow (w)
  (reverse (loop for d on (reverse w) collect (reduce #'+ d))))
> (drop-empty '(" " "Hello" "," "" "World!"))
("Hello" "," "World!")
```

Задача 4.444 b.lisp

Read other people's code and 'steal' the good parts. Even if you try to do things exactly like someone else, you'll fail, but in doing so create something that's truly your own.

Peter Cooper (invoking Roman Mars)

Give someone a program, you frustrate them for a day; teach them how to program, you frustrate them for a lifetime.

David Leinweber

Определите функцию *a*, единственный аргумент которой является списком. Функция *a* возвращает *t* только в том случае, если первые два элемента этого списка являются списками. В остальных случаях функция должна выдавать *nil*. Затем, определите функцию *b* с одним аргументом – списком. Если функция *a* возвращает *t*, создать список из четырех элементов: 1 и 2 элемент 1-го списка + 1 и 2 элементы 2-го списка. Если функция возвращает *nil*, то убрать из списка первые два элемента.

Решение 4.444.1

```
(defun a (w)
  (when (cdr w) (and (listp (car w))
                     (listp (cadr w))))))

(defun b (w)
  (if (a w)
      (list (caar w) (cadr w) (caadr w) (cadadr w))
      (nthcdr 2 w)))

> (b '((a b d) (e f g) h i))
(A B E F)
> (b '((a) (e) h i))
(A NIL E NIL)
> (b '((a) h i))
(I)
```

Задача 4.445 *ssort.lisp*

As a programmer, never underestimate your ability to come up with ridiculously complex solutions for simple problems.

Thomas Fuchs

If you can get today's work done today, but you do it in such a way that you can't possibly get tomorrow's work done tomorrow, then you lose.

Martin Fowler

You're building your own maze, in a way, and you might just get lost in it.

Marijn Haverbeke

Определить функцию, сортирующую список по третьему элементу (числу) подписков по возрастанию.

Решение 4.445.1

```
(defun ssort (w)
  (sort w #'< :key #'third))

> (ssort '((d e 2) (e c 3) (b c 1) (c d 4) (a b 5) (a z 6) (c a 2) (b
d 3 )))
((B C 1) (D E 2) (C A 2) (E C 3) (B D 3) (C D 4) (A B 5) (A Z 6))
```

Задача 4.446 *deep-double.lisp*

Learning to code is the single best thing anyone can do to get the most out of the amazing future in front of us.

Marc Andreessen

And programming computers was so fascinating. You create your own little universe, and then it does what you tell it to do.

Vint Cerf

Определить функцию, которая увеличивает все элементы многоуровневого числового списка в два раза.

Решение 4.446.1

```
(defun deep-double (w)
```

```

(loop for a in w
      if (atom a) collect (* a 2)
      else collect (deep-double a)))

> (deep-double '(1 2 3 (4 (5 6)) 7))
(2 4 6 (8 (10 12)) 14)

```

Решение 4.446.2

```

(defun deep-double (w &aux (a (car w)))
  (cond ((null w) nil)
        ((atom a) (cons (* a 2)
                          (deep-double (cdr w))))
        (t (cons (deep-double a)
                    (deep-double (cdr w))))))

> (deep-double '(1 2 3 (4 (5 6)) 7))
(2 4 6 (8 (10 12)) 14)

```

Решение 4.446.3

```

(defun deep-double (w &aux (a (car w)))
  (when w (cons (if (atom a)
                    (* a 2)
                    (deep-double a))
                 (deep-double (cdr w)))))

> (deep-double '(1 2 3 (4 (5 6)) 7))
(2 4 6 (8 (10 12)) 14)

```

Решение 4.446.4

```

(defun deep-double (w &optional ac &aux (a (car w)))
  (if w (deep-double (cdr w)
                     (cons (if (atom a)
                               (* a 2)
                               (deep-double a))
                           ac))
      (reverse ac)))

> (deep-double '(1 2 3 (4 (5 6)) 7))
(2 4 6 (8 (10 12)) 14)

```

Решение 4.446.5

```

(defun deep-double (w)
  (loop for a in w collect
        (if (atom a)
            (* a 2)
            (deep-double a))))

> (deep-double '(1 2 3 (4 (5 6)) 7))
(2 4 6 (8 (10 12)) 14)

```

Задача 4.447 sum-nrow.lisp

Everyday life is like programming, I guess.
If you love something you can put beauty into it.

Donald Knuth

Пройдёте в сад? Я покажу Вас розам.

Ричард Шеридан

Определить функцию, возвращающую список сумм: 1-го элемента, 2-х первых элементов, 3-х первых элементов и т.д.

Решение 4.447.1

```
(defun sum-nrow (w)
  (reverse (maplist #'(lambda (d) (reduce #'+ d))
                (reverse w))))

> (sum-nrow '(1 2 3 4 5 6 7))
(1 3 6 10 15 21 28)
```

Решение 4.447.2

```
(defun sum-nrow (w)
  (reverse (loop for d on (reverse w) collect (reduce #'+ d))))

> (sum-nrow '(1 2 3 4 5 6 7))
(1 3 6 10 15 21 28)
```

Задача 4.448 modify.lisp

That's what's cool about working with computers. They don't argue, they remember everything and they don't drink all your beer.

Paul Leary

In carpentry, you measure twice and cut once. In software development, you never measure and make cuts until you run out of time.

Adam Morse

Дан список. Если первый и последний элементы списка чётные – то поменять их местами. Иначе поменять местами голову и хвост списка.

Пример:

```
(0 8 3 4 1 6 2) -> (2 8 3 4 1 6 0)
(1 5 3 2 7 8) -> ((5 3 2 7 8) 1)
```

Решение 4.448.1

```
(defun modify (w
              &aux
                (a (car w))
                (z (car (last w))))
  (if (and (evenp a) (evenp z))
      (append (list z) (butlast (cdr w)) (list a))
      (cons (cdr w) (list a))))

> (modify '(0 8 3 4 1 6 2))
(2 8 3 4 1 6 0)
> (modify '(1 5 3 2 7 8))
((5 3 2 7 8) 1)
```

Решение 4.448.2

```
(defun modify (w &aux (a (car w)) (z (last w)))
  (if (and (evenp a) (evenp (car z)))
      (append z (butlast (cdr w)) (list a))
      (cons (cdr w) (list a))))

> (modify '(0 8 3 4 1 6 2))
(2 8 3 4 1 6 0)
> (modify '(1 5 3 2 7 8))
((5 3 2 7 8) 1)
```

Решение 4.448.3

```
(defun modify (w
              &aux
                (a (car w))
                (d (cdr w))
                (z (last w)))
  (if (and (evenp a) (evenp (car z)))
      (append z (butlast d) (list a))
      (cons d (list a))))

> (modify '(0 8 3 4 1 6 2))
(2 8 3 4 1 6 0)
> (modify '(1 5 3 2 7 8))
((5 3 2 7 8) 1)
```

Решение 4.448.4

```
(defun modify (w
              &aux
                (a (car w))
                (d (cdr w))
                (z (car (last w))))
  (nconc (if (every #'evenp (list a z))
             (cons z (butlast d))
```

```

      ` (,d))
    ` (,a)))

```

```

> (modify '(0 8 3 4 1 6 2))
(2 8 3 4 1 6 0)
> (modify '(1 5 3 2 7 8))
((5 3 2 7 8) 1)

```

Решение 4.448.5

```

(defun modify (w
  &aux
    (a (car w))
    (d (cdr w))
    (z (car (last w))))
  (nconc (if (some #'oddp ` (,a ,z))
    ` (,d)
    (cons z (butlast d)))
    ` (,a)))

```

```

> (modify '(0 8 3 4 1 6 2))
(2 8 3 4 1 6 0)
> (modify '(1 5 3 2 7 8))
((5 3 2 7 8) 1)

```

Решение 4.448.6

```

(defun modify (w
  &aux
    (a (car w))
    (d (cdr w))
    (z (last w)))
  (nconc (if (some #'oddp (cons a z))
    ` (,d)
    (cons (car z) (butlast d)))
    ` (,a)))

```

```

> (modify '(0 8 3 4 1 6 2))
(2 8 3 4 1 6 0)
> (modify '(1 5 3 2 7 8))
((5 3 2 7 8) 1)

```

Задача 4.449 drop-atoms.lisp

Определить функцию, которая для заданного списка формирует новый список по правилу: в новый список включены только те элементы, которые сами являются списками. Пример: (function '(a (b a) () c) -> ((b a) ())).

Решение 4.449.1

```

(defun drop-atoms (w)
  (loop for a in w when (listp a) collect a))

```

```
> (drop-atoms '(a (b a) () c))
((B A) NIL)
```

Решение 4.449.2

```
(defun drop-atoms (w &optional ac &aux (a (car w)))
  (if w (drop-atoms (cdr w)
                    (if (listp a) (cons a ac)
                        ac))
      (reverse ac)))
```

```
> (drop-atoms '(a (b a) () c))
((B A) NIL)
```

Задача 4.450 list-pair.lisp

You might not think that programmers are artists, but programming is an extremely creative profession. Its logic-based creativity.

John Romero

Be curious. Read widely. Try new things. I think a lot of what people call intelligence boils down to curiosity.

Aaron Swartz

Documentation is a love letter that you write to your future self.

Damian Conway

Дан список слов. Определить функцию, которая возвращает список слов с указанием их частоты.

Решение 4.450.1

```
(defun list-pair (words &aux (hash (make-hash-table)) ac)
  (dolist (word words) (incf (gethash word hash 0)))
  (maphash #'(lambda (e n) (push `(,e ,n) ac)) hash) ac)
```

```
> (list-pair '(lisp is the second-oldest high-level programming language in widespread use today lisp has changed since its early days and many dialects have existed over its history today the best known general-purpose Lisp dialects are common lisp and scheme))
((LISP 4) (IS 1) (THE 2) (SECOND-OLDEST 1) (HIGH-LEVEL 1) (PROGRAMMING 1) (LANGUAGE 1) (IN 1) (WIDESPREAD 1) (USE 1) (TODAY 2) (HAS 1) (CHANGED 1) (SINCE 1) (ITS 2) (EARLY 1) (DAYS 1) (AND 2) (MANY 1) (DI- ALECTS 2) (HAVE 1) (EXISTED 1) (OVER 1) (HISTORY 1) (BEST 1) (KNOWN 1)
```


(GENERAL-PURPOSE 1) (ARE 1) (COMMON 1) (SCHEME 1))

Задача 4.451 adjacency-list.lisp

The purpose of software engineering is to control complexity, not to create it.

Pamela Zave

Simplicity is about subtracting the obvious and adding the meaningful.

John Maeda

Определить функцию, которая строит список смежностей по представлению графа, заданному в виде матрицы смежностей.

Решение 4.451.1

```
(defun adjacency-list (w &aux (n (length w)) ac)
  (dotimes (i n (reverse ac))
    (dotimes (j (- n i) t)
      (unless (zerop (nth (+ i j) (nth i w)))
        (setf ac (cons (list (1+ i) (+ j i 1)) ac))))))

> (adjacency-list '((0 1 1 0) (1 0 0 1) (1 0 0 0) (0 1 0 0)))
((1 2) (1 3) (2 4))
```

Задача 4.452 not-arithmetic-progression-permutations.lisp

Дан список различных натуральных чисел. Определить функцию возвращающую список таких перестановок, чтобы никакие три элемента не представляли собой арифметическую прогрессию.

Решение 4.452.1

```
(defun not-arithmetic-progression-permutations (v)
  (not-arithmetic (permute v)))

(defun permute (w)
  (cond ((null w) nil)
        ((null (cdr w)) (list w))
        ((loop for a in w
                 nconc (mapcar #'(lambda (e) (cons a e))
                               (permute (remove a w))))))

(defun not-arithmetic (w)
  (cond ((null (cddr w)) nil)
        ((arithmetic-progressionp (caar w) (cadar w) (caddar w))
         (not-arithmetic (cdr w)))
        (t (cons (car w) (not-arithmetic (cdr w))))))
```

```
(defun arithmetic-progressionp (a b c)
  (or (and (< a b c) (= (- b a) (- c b)))
      (and (> a b c) (= (- a b) (- b c)))))

> (not-arithmetic-progression-permutations '(1 2 3 4))
((1 2 4 3) (1 3 2 4) (1 3 4 2) (1 4 2 3) (1 4 3 2) (2 1 3 4) (2 1 4 3)
 (2 3 1 4) (2 4 1 3) (2 4 3 1) (3 1 2 4) (3 1 4 2) (3 2 4 1) (3 4 1 2)
 (3 4 2 1) (4 1 2 3) (4 1 3 2) (4 2 1 3) (4 2 3 1))
```

Решение 4.452.2

```
(defun not-arithmetic-progression-permutations (v)
  (not-arithmetic (permutate v)))

(defun permutate (w)
  (when w (if (cdr w)
              (loop for a in w
                    nconc (mapcar #'(lambda (e) (cons a e))
                                   (permutate (remove a w))))
              (list w))))

(defun not-arithmetic (w)
  (cond ((null (cddr w)) nil)
        ((arithmetic-progressionp (caar w) (cadar w) (caddr w))
         (not-arithmetic (cdr w)))
        (t (cons (car w) (not-arithmetic (cdr w))))))

(defun arithmetic-progressionp (a b c)
  (or (and (< a b c) (= (- b a) (- c b)))
      (and (> a b c) (= (- a b) (- b c)))))

> (not-arithmetic-progression-permutations '(1 2 3 4))
((1 2 4 3) (1 3 2 4) (1 3 4 2) (1 4 2 3) (1 4 3 2) (2 1 3 4) (2 1 4 3)
 (2 3 1 4) (2 4 1 3) (2 4 3 1) (3 1 2 4) (3 1 4 2) (3 2 4 1) (3 4 1 2)
 (3 4 2 1) (4 1 2 3) (4 1 3 2) (4 2 1 3) (4 2 3 1))
```

Задача 4.453 frequency.lisp

Определить функцию, преобразующий список *w* в новый список, элементы которого имеют вид: (<элемент списка *w*> <кол-во вхождений этого элемента в список *w*>).

Решение 4.453.1

```
(defun frequency (w)
  (loop for a in (remove-duplicates w)
        collect (list a (count a w))))

> (frequency '(a b a a c b))
((A 3) (C 1) (B 2))
```

Решение 4.453.2

```
(defun frequency (w)
  (loop for a in (remove-duplicates w)
        collect `(,a ,(count a w))))

> (frequency ' (a b a a c b))
((A 3) (C 1) (B 2))
```

Решение 4.453.3 (автор - VH, www.cyberforum.ru)

```
(defun frequency (w)
  (when w ((lambda (e r)
              ((lambda (found)
                 (if found
                     (subst (cons e (1+ (cdr found)))
                             found r :test #'equal)
                     (cons (cons e 1) r)))
                (assoc e r)))
           (car w) (frequency (cdr w)))))

> (frequency ' (a b a a c b))
((A . 3) (C . 1) (B . 2))
```

Решение 4.453.4 (автор - vlisp, www.cyberforum.ru)

```
(defun frequency (w &aux (a (car w)))
  (when w (cons (list a (count a w))
                 (frequency (remove a w)))))
```

Задача 4.454 *blast.lisp*

Lisp is the simplest, most elegant notation we've discovered to represent both programmer code and arbitrary data at the same time. All the other data notations are worse (more verbose) for code. And all the other code notations are worse (lose structure) for data.

Phil Jones

Определить функцию, которая отсекает последний элемент списка.

Решение 4.454.1

```
(defun blast (w)
  (cond ((null (cdr w)) nil)
        (t (cons (car w) (blast (cdr w))))))

> (blast '(1 2 3))
(1 2)
```

Решение 4.454.2

```
(defun blast (w)
  (when (cdr w) (cons (car w) (blast (cdr w)))))

> (blast '(1 2 3))
(1 2)
```

Решение 4.454.3

```
(defun blast (w &aux (d (cdr w)))
  (when d (cons (car w) (blast d))))

> (blast '(1 2 3))
(1 2)
```

Задача 4.455 *flatten.lisp*

The thing that blows my mind about Lisp is that Lisp was first developed in 1958. Yes, it's gone through a lot of changes since that time, but the fact that it still seems like a modern meaningful language after all that time is a testament to its fundamental strengths. When you look at the kind of amazing things that were going on in programming language design in those decades it just really makes you think.

Ryan Brown

Определить функцию, раскрывающую вложенные списки.

Решение 4.455.1

```
(defun flatten (w &optional ac)
  (cond ((null w) ac)
        ((atom w) (cons w ac))
        ((flatten (car w) (flatten (cdr w) ac)))))

> (flatten '(a (b (c))))
(A B C)
```

Решение 4.455.2

```
(defun flatten (w)
  (loop for a in w
        if (listp a) append (flatten a)
        else collect a))

> (flatten '(a (b (c))))
(A B C)
```

Решение 4.455.3

```
(defun flatten (w)
  (mapcan #'(lambda (a) (if (listp a) (flatten a) `(:,a))) w))

> (flatten '(a (b (c))))
(A B C)
```

Решение 4.455.4

```
(defun flatten (w) (if (atom w) `(:,w) (mapcan #'flatten w)))

> (flatten '(a (b (c))))
(A B C)
```

Решение 4.455.5

```
(defun flatten (w &optional ac)
  (if w (if (atom w) (cons w ac)
            (flatten (car w) (flatten (cdr w) ac)))
    ac))

> (flatten '(a (b (c))))
(A B C)
```

Решение 4.455.6

```
(defun flatten (w)
  (loop for a in w
        if (listp a) nconc (flatten a)
        else collect a))

> (flatten '(a (b (c))))
(A B C)
```

Решение 4.455.7

```
(defun flatten (w)
  (loop for a in w nconc (if (listp a) (flatten a) `(:,a))))

> (flatten '(a (b (c))))
(A B C)
```

Структура 5 (функция АТОМ АТОМ) > АТОМ

Incidentally, a real power of Emacs is that all these things everyone mentioned can be combined and multiplied off each other. No single feature can be considered in isolation, because they feed off each other. And instead of being a monster, this same power allows one to hide and control complexity.

Tayssir John Gabbour

Задача 5.1 runner.lisp

Дано расстояние в метрах и время в секундах. Определить функцию, возвращающую скорость.

Решение 5.1.1

```
(defun runner (d s) (float (/ d s)))  
  
> (runner 2420 720)  
3.3611112
```

Задача 5.2 average.lisp

Определить функцию, которая возвращает среднее арифметическое двух положительных чисел.

Решение 5.2.1

```
(defun average (a b)  
  "Returns the average of the numbers a and b."  
  ;; Doesn't round or truncate integers  
  (float (/ (+ a b) 2)))  
  
> (average 2 3)  
2.5
```

Задача 5.3 string-right-n.lisp

Дана строка. Определить функцию, которая возвращает строку из n последних знаков исходной строки.

Решение 5.3.1

```
(defun string-right (s n) (subseq s (- (length s) n)))
```

```
> (string-right "abcde" 3)
"cde"
```

Задача 5.4 *add-range.lisp*

Определить функцию вычисления суммы целых чисел от a до b .

Решение 5.4.1

```
(defun add-range (a b)
  (if (= a b) b (+ a (add-range (1+ a) b))))

> (add-range 4 7)
22
```

Решение 5.4.2

```
(defun add-range (a b)
  (loop for e from a to b sum e))

> (add-range 4 7)
22
```

Решение 5.4.3

```
(defun add-range (a b)
  (/ (* (+ a b) (1+ (- b a))) 2))

> (add-range 4 7)
22
```

Задача 5.5 *disk-area.lisp*

Написать функцию, которая считает площадь круга, если известны внутренний и внешний радиусы.

Решение 5.5.1

```
(defun disk-area (r2 r1)
  (- (* pi r2 r2) (* pi r1 r1)))

> (disk-area 3 2)
15.707963267948966193L0
```

Задача 5.6 *solid-torus.lisp*

Написать функцию, которая считает объем, ограниченный внутри тора, если известны внутренний и внешний радиусы.

Решение 5.6.1

```
(defun solid-torus (r2 r1) (* pi pi r1 (expt (- r2 r1) 2) 1/2))

> (solid-torus 3 2)
9.869604401089358619L0
```

Задача 5.7 $a+a*[a+1]+a*[a+1]*[a+2]+\dots+a*[a+1]*\dots*[a+n]$.lisp

Определите функцию $(a+a*[a+1]+a*[a+1]*[a+2]+\dots+a*[a+1]*\dots*[a+1] \ a \ n)$, которая от двух числовых аргументов вычисляет величину:
 $a + a * (a + 1) + a * (a + 1) * (a + 2) + \dots + a * (a + 1) * \dots * (a + n)$.

Решение 5.7.1

```
(defun az (a n)
  (cond ((zerop n) a)
        ((* (+ a n) (az a (1- n))))))

(defun a+a*[a+1]+a*[a+1]*[a+2]+\dots+a*[a+1]*\dots*[a+n] (a n)
  (cond ((zerop n) a)
        ((+ (a+a*[a+1]+a*[a+1]*[a+2]+\dots+a*[a+1]*\dots*[a+n]
              a (1- n)) (az a n)))))

> (a+a*[a+1]+a*[a+1]*[a+2]+\dots+a*[a+1]*\dots*[a+n] 1 2)
9
> (a+a*[a+1]+a*[a+1]*[a+2]+\dots+a*[a+1]*\dots*[a+n] 1 3)
33
```

Решение 5.7.2

```
(defun az (a n)
  (cond ((zerop n) a)
        ((* (+ a n) (az a (1- n))))))

(defun bx (a n)
  (cond ((zerop n) a)
        ((+ (bx a (1- n)) (az a n)))))

> (bx 1 3)
33
```

Решение 5.7.3

```
(defun az (a n)
  (if (zerop n) a (* (+ a n) (az a (1- n)))))

(defun bx (a n)
  (if (zerop n) a (+ (bx a (1- n)) (az a n))))

> (bx 1 3)
33
```


Решение 5.7.4 (автор - VH, www.hardforum.ru)

```
(defun bx (a n)
  (if (zerop n) a (* a (1+ (bx (1+ a) (1- n))))))

> (bx 1 3)
33
```

Задача 5.8 insert-lets.lisp

Написать программу обработки текста естественного языка. Дан текст. В каждом слове вставить после заданного 3-буквенного сочетания заданное 2-буквенное.

Решение 5.8.1*

```
(defun insert-lets (s p r &aux (e (search p s)))
  (cond ((null e) s)
        ((concatenate 'string
                        (subseq s 0 e)
                        p
                        r
                        (insert-lets (subseq s (+ e 3)) p r)))))

> (insert-lets "abcde abcde abcde" "bcd" "zz")
"abcdzze abcdzze abcdzze"
```

По функциям:

```
> (concatenate 'string "a" "b" "c")
"abc"
> (subseq '(a b c d) 1 3)
(B C)
> (subseq "abcd" 1 3)
"bc"
> (subseq "abcd" 1)
"bcd"
```

Решение 5.8.2

```
(defun insert-lets (s p r &aux (e (search p s)))
  (if e
      (concatenate 'string
                    (subseq s 0 e)
                    p
                    r
                    (insert-lets (subseq s (+ e 3)) p r))
      s))

> (insert-lets "abcde abcde abcde" "bcd" "zz")
"abcdzze abcdzze abcdzze"
```

Задача 5.9 calculate.lisp

Написать функцию, которая принимает ввод двух чисел и операции между ними и возвращает результат.

Решение 5.9.1

```
(defun calculate (&aux cmd)
  (loop (print "Enter: Number Operator Number")
        (setq cmd (read-from-string
                     (concatenate 'string "(" (read-line) ")"))))
        (when (eql (car cmd) 'quit) (return 'ok))
        (print (eval `(,(cadr cmd) ,(car cmd) ,(caddr cmd))))))

> (calculate)
"Enter: Number Operator Number" 2 + 3
5
"Enter: Number Operator Number" quit
OK
```

Решение 5.9.2

```
(defun calculate (&aux cmd)
  (loop (print '>)
        (setq cmd (read-from-string
                     (concatenate 'string "(" (read-line) ")"))))
        (when (eql (car cmd) 'quit) (return 'ok))
        (print (eval `(,(cadr cmd) ,(car cmd) ,(caddr cmd))))))

> (calculate)
> 2 + 3
5
> quit
OK
```

Задача 5.10 greatest-common-divisor.lisp

Определить функцию, вычисляющую наибольший общий делитель двух чисел - алгоритм Евклида (Eukléides, около 365-300 г. до н. э.).

Решение 5.10.1

```
(defun greatest-common-divisor (a b)
  (if (zerop (rem a b)) b (greatest-common-divisor b (rem a b))))

> (greatest-common-divisor 32453 345)
23
```

Решение 5.10.2

```
(defun greatest-common-divisor (a b)
  (if (zerop (mod a b)) b (greatest-common-divisor b (mod a b))))
```

```
> (greatest-common-divisor 32453 345)
23
```

Решение 5.10.3

```
(defun greatest-common-divisor (a b)
  (if (zerop b) a (greatest-common-divisor b (mod a b))))

> (greatest-common-divisor 32453 345)
23
```

Решение 5.10.4

```
(defun greatest-common-divisor (a b)
  (gcd a b))

> (greatest-common-divisor 32453 345)
23
```

Задача 5.11 *replace-vowels.lisp*

Дана строка текста. Определить функцию, которая заменяет все гласные буквы на букву «z»

Решение 5.11.1

```
(defun replace-vowels (s r &optional (w '("a" "e" "i" "o" "u")))
  (if w (replace-vowels (replace-lets s (car w) r) r (cdr w)) s))

(defun replace-lets (s p r &aux (e (search p s)))
  (if e
      (concatenate
        'string
        (subseq s 0 e)
        r
        (replace-lets (subseq s (1+ e)) p r))
      s))

> (replace-vowels "aberikodubra" "z")
"zbzrzkdzdzbrz"
```

Задача 5.12 *co-prime.lisp*

Написать функцию (предикат), определяющую, являются ли ее аргументы взаимно простыми числами.*

* Обычно число зубьев на звёздочках и число звеньев цепи в цепной передаче стремятся делать взаимно простыми, чтобы обеспечить равномерность износа: каждый зуб звёздочки будет поочередно работать со всеми звеньями цепи.

Решение 5.12.1

```
(defun co-prime (a b) (= (gcd a b) 1))

> (co-prime 14 25)
T
```

Решение 5.12.2

```
(defun co-prime (&rest w) (= (apply #'gcd w) 1))

> (co-prime 10 15 27)
T
```

Задача 5.13 *sum-cubes.lisp*

Написать функцию, вычисляющую сумму кубов чисел от n до m включительно.

Решение 5.13.1

```
(defun sum-cubes (n m)
  (if (> n m) 0
      (+ (expt n 3) (sum-cubes (1+ n) m))))

> (sum-cubes 2 5)
224
```

Задача 5.14 *sum-expt-n-m.lisp*

Определить функцию, вычисляющую сумму чисел от 1 до n включительно в степени m .

Решение 5.14.1

```
(defun sum-expt-n-m (n m) (loop for a from 1 to n sum (expt a m)))

> (sum-expt-n-m 10 2)
385
```

Задача 5.15 *quotient.lisp*

Определить функцию деления нацело через вычитание.

Решение 5.15.1

```
(defun quotient (a b)
  (cond ((zerop b) nil)
        ((< a b) 0)
        ((> a b) (1+ (quotient (- a b) b)))))
```

```
> (quotient 10 3)
3
```

Решение 5.15.2

```
(defun quotient (a b)
  (unless (zerop b) (if (< a b) 0 (1+ (quotient (- a b) b)))))

> (quotient 10 3)
3
```

Задача 5.16 quotient-remainder.lisp

Определить через вычитание функцию, возвращающую частное и остаток от деления.

Решение 5.16.1

```
(defun quotient-remainder (a b &optional (n 0))
  (if (zerop b)
      'division-by-zero
      (if (< a b)
          (values n a)
          (quotient-remainder (- a b) b (1+ n)))))

> (quotient-remainder 10 3)
3
1
```

Решение 5.16.2

```
(defun quotient-remainder (a b &optional (n 0))
  (cond ((zerop b) 'division-by-zero)
        ((< a b) (list n a))
        ((quotient-remainder (- a b) b (1+ n)))))

> (quotient-remainder 10 3)
(3 1)
```

Задача 5.17 -expt.lisp

Определить функцию возведения в степень.

Решение 5.17.1

```
(defun -expt (m n)
  (cond ((zerop n) 1)
        ((= n 1) m)
        ((* m (-expt m (1- n))))))
```

```
> (-expt 2 0)
1
> (-expt 2 3)
8
```

Функция `-expt` является рекурсивной, то есть вызывает себя до тех пор, пока ее аргумент `n` (счетчик) не уменьшится до 1 (то есть `n - 1` раз), после чего функция вернет значение `m`. При каждой передаче значения `m` на предыдущий уровень, результат умножается на `m`. Так `m` окажется перемноженным на себя `n` раз.

Решение 5.17.2

```
(defun -expt (n m) (reduce #'* (loop repeat m collect n)))

> (-expt 2 0)
1
> (-expt 2 3)
8
```

Решение 5.17.3

```
(defun -expt (m n)
  (cond ((= n 1) m)
        ((zerop n) 1)
        ((minusp n) (/ 1 (-expt m (- n)))))
        ((* m (-expt m (1- n))))))

> (-expt 2 -3)
1/8
```

Задача 5.18 *add.lisp*

Определить функцию, которая складывает два числа.

Решение 5.18.1

```
(defun dec (a) (- a 1))

(defun inc (a) (+ a 1))

(defun add (a b)
  (if (= a 0) b (+ (dec a) (inc b))))

> (add 2 3)
5
```

Решение 5.18.2

```
(defun dec (a) (- a 1))

(defun inc (a) (+ a 1))
```

```
(defun add (a b)
  (if (= a 0) b (inc (+ (dec a) b))))

> (add 2 3)
5
```

Задача 5.19 money-lost.lisp

Эта задача для второго класса церковно приходской школы была придумана Львом Толстым. Продавец продает шапку, которая стоит 10 р. Подходит покупатель, меряет и согласен взять, но у него есть только 25 р. Продавец отсылает мальчика с этими 25 р. к соседке, разменять. Мальчик прибегает и отдает 10+10+5. Продавец отдает шапку и сдачу в 15 руб. Через какое то время приходит соседка и говорит что 25 р. фальшивые, требует отдать ей деньги. Что делать? Мужик лезет в кассу и возвращает ей деньги. Вопрос: На сколько обманули продавца?

Решение 5.19.1

```
(defun money-lost (hat fake)
  (- fake hat (- fake hat) fake))

> (money-lost 10 25)
-25
```

Решение 5.19.2

Стоимость шапки можно не брать в расчет:

```
(defun money-lost (hat fake)
  (- (- fake fake) (- hat hat) fake))

> (money-lost 20 25)
-25
```

или

Решение 5.19.1

```
(defun money-lost (fake) (- fake))

> (money-lost 25)
-25
```

Можно по разному комбинировать числа в скобках, но иногда решающий учитывает лишь отдачу -10 (шапка покупателю), -15 (сдача покупателю), -25 (возврат соседке) и не учитывает настоящие деньги, полученные от соседки +25. Продавец получит +25 и вернет соселке полученные от нее настоящие деньги -25 и выйдет в ноль, по этому направлению не формируются потери, а все его потери в направлении покупателя -25 (стоимость шапки -10 и сдача покупателю -15 или фальшивые деньги -25).

Или в столбик:

```
- 10 ушла шапка к покупателю
- 15 ушла сдача к покупателю
+ 25 пришли настоящие деньги от соседки
- 25 ушли настоящие деньги к соседке
Итого: - 25
```

Задача 5.20 *brick.lisp*

Задачка на сообразительность для выявления типа мышления: кирпич весит килограмм плюс полкирпича. Вопрос: сколько весит кирпич?

Решение 5.20.1

```
(defun brick (a b) (/ a (- 1 b)))

> (brick 1 0.5)
2.0
> (brick 3 0.25)
4.0
```

или:

```
x = a + b * x
x - b * x = a
x * (1 - b) = a
x = a : (1 - b)
```

Задача 5.21 *vowels>a.lisp*

Дана строка текста. Определить функцию, которая возвращает копию этой строки, в которой все гласные буквы заменены на букву «а».

Решение 5.21.1

```
(defun vowels>a (s z)
  (coerce
    (loop for c across s
      if (find c '(#\a #\e #\i #\o #\u))
        collect z
        else collect c)
    'string))

> (vowels>a "congratulations" #\a)
"cangratalataans"
```

Задача 5.22 *adjacent-digits-greatest-product.lisp*

The four adjacent digits in the 1000-digit number that have the greatest product are $9 \times 9 \times 8 \times 9 = 5832$. Find the thirteen adjacent dig-

its in the 1000-digit number that have the greatest product. What is the value of this product.

Наибольшее произведение четырех последовательных цифр в нижеприведенном 1000-значном числе равно $9 \times 9 \times 8 \times 9 = 5832$. Найдите наибольшее произведение тринадцати последовательных цифр в данном числе)?

```
73167176531330624919225119674426574742355349194934
96983520312774506326239578318016984801869478851843
85861560789112949495459501737958331952853208805511
12540698747158523863050715693290963295227443043557
66896648950445244523161731856403098711121722383113
62229893423380308135336276614282806444486645238749
30358907296290491560440772390713810515859307960866
70172427121883998797908792274921901699720888093776
65727333001053367881220235421809751254540594752243
52584907711670556013604839586446706324415722155397
53697817977846174064955149290862569321978468622482
8397224137565705605749026140797296865241453100474
82166370484403199890008895243450658541227588666881
16427171479924442928230863465674813919123162824586
17866458359124566529476545682848912883142607690042
24219022671055626321111109370544217506941658960408
07198403850962455444362981230987879927244284909188
84580156166097919133875499200524063689912560717606
05886116467109405077541002256983155200055935729725
71636269561882670428252483600823257530420752963450
```

Решение 5.22.1

```
(defun digitize (n &optional m)
  (cond ((zerop n) m)
        (t (multiple-value-bind (q r) (truncate n 10)
              (digitize q (cons r m))))))

(defun product (n s &optional (z 0))
  (cond ((< (length s) n) z)
        ((> (reduce #'* (digitize
                          (parse-integer
                           (subseq s 0 n))))
            z)
         (product n
                   (subseq s 1)
                   (reduce #'* (digitize
                               (parse-integer
                                (subseq s 0 n)))))))
        ((product n (subseq s 1) z))))

(defun adjacent-digits-greatest-product (n m)
  (product n (write-to-string m)))
```

```
> (adjacent-digits-greatest-product 13
7316717653133062491922511967442657474235534919493496983520312774506326
```

```

2395783180169848018694788518438586156078911294949545950173795833195285
3208805511125406987471585238630507156932909632952274430435576689664895
0445244523161731856403098711121722383113622298934233803081353362766142
8280644448664523874930358907296290491560440772390713810515859307960866
7017242712188399879790879227492190169972088809377665727333001053367881
2202354218097512545405947522435258490771167055601360483958644670632441
5722155397536978179778461740649551492908625693219784686224828397224137
5657056057490261407972968652414535100474821663704844031998900088952434
5065854122758866688116427171479924442928230863465674813919123162824586
1786645835912456652947654568284891288314260769004224219022671055626321
1111093705442175069416589604080719840385096245544436298123098787992724
4284909188845801561660979191338754992005240636899125607176060588611646
7109405077541002256983155200055935729725716362695618826704282524836008
23257530420752963450)
23514624000

```

Решение 5.22.2

```

(defun digitize (n &optional ac)
  (if (zerop n) ac (digitize (truncate n 10)
                              (cons (rem n 10) ac))))

(defun product (n s
                &optional (z 0)
                &aux
                (k (length s))
                (m (when (>= k n)
                        (reduce #'* (digitize
                                   (parse-integer
                                    (subseq s 0 n)))))))
  (cond ((< k n) z)
        ((> m z) (product n (subseq s 1) m))
        ((product n (subseq s 1) z))))

(defun adjacent-digits-greatest-product (n m)
  (product n (write-to-string m)))

> (adjacent-digits-greatest-product 13
7316717653133062491922511967442657474235534919493496983520312774506326
2395783180169848018694788518438586156078911294949545950173795833195285
3208805511125406987471585238630507156932909632952274430435576689664895
0445244523161731856403098711121722383113622298934233803081353362766142
8280644448664523874930358907296290491560440772390713810515859307960866
7017242712188399879790879227492190169972088809377665727333001053367881
2202354218097512545405947522435258490771167055601360483958644670632441
5722155397536978179778461740649551492908625693219784686224828397224137
5657056057490261407972968652414535100474821663704844031998900088952434
5065854122758866688116427171479924442928230863465674813919123162824586
1786645835912456652947654568284891288314260769004224219022671055626321
1111093705442175069416589604080719840385096245544436298123098787992724
4284909188845801561660979191338754992005240636899125607176060588611646
7109405077541002256983155200055935729725716362695618826704282524836008
23257530420752963450)

```

23514624000

Решение 5.22.3

```
(defun digitize (n &optional ac)
  (if (zerop n) ac (digitize (truncate n 10)
                              (cons (rem n 10) ac))))

(defun product (n w
                &optional (z 0)
                &aux
                (k (< (length w) n))
                (m (unless k (reduce #'* (subseq w 0 n)))))
  (if k
      z
      (product n (cdr w) (if (> m z) m z))))

(defun adjacent-digits-greatest-product (n m)
  (product n (digitize m)))

> (adjacent-digits-greatest-product 13
2316717653133062491922511967442657474235534919493496983520312774506326
2395783180169848018694788518438586156078911294949545950173795833195285
3208805511125406987471585238630507156932909632952274430435576689664895
0445244523161731856403098711121722383113622298934233803081353362766142
8280644448664523874930358907296290491560440772390713810515859307960866
7017242712188399879790879227492190169972088809377665727333001053367881
2202354218097512545405947522435258490771167055601360483958644670632441
5722155397536978179778461740649551492908625693219784686224828397224137
5657056057490261407972968652414535100474821663704844031998900088952434
5065854122758866688116427171479924442928230863465674813919123162824586
1786645835912456652947654568284891288314260769004224219022671055626321
1111093705442175069416589604080719840385096245544436298123098787992724
4284909188845801561660979191338754992005240636899125607176060588611646
7109405077541002256983155200055935729725716362695618826704282524836008
23257530420752963450)
23514624000
```

Задача 5.23 *sum-primes.lisp*

The sum of the primes below 10 is $2 + 3 + 5 + 7 = 17$. Find the sum of all the primes below two million.

Сумма простых чисел меньше 10 – это $2 + 3 + 5 + 7 = 17$. Найдите сумму всех простых чисел меньше двух миллионов.

Решение 5.23.1

```
(defun check (n)
  (loop for a from 2 to (isqrt n)
        never (zerop (mod n a))))
```

```
(defun sum-primes (n m)
  (loop for a from n to m
        when (check a) sum a))

> (time (sum-primes 2 2000000))
Real time: 79.05548 sec.
Run time: 78.88971 sec.
Space: 2278420 Bytes
GC: 5, GC time: 0.0468003 sec.
142913828922
```

Задача 5.24 power-digit-sum.lisp

(expt 2 15) = 32768 and the sum of its digits is $3 + 2 + 7 + 6 + 8 = 26$. What is the sum of the digits of the number (expt 2 1000)?

(expt 2 15) = 32768, сумма цифр $3 + 2 + 7 + 6 + 8 = 26$. Какова сумма цифр числа (expt 2 1000)?

Решение 5.24.1

```
(defun sum-digit (n)
  (if (zerop n) 0
      (+ (rem n 10) (sum-digit (truncate n 10)))))

(defun power-digit-sum (a b)
  (sum-digit (expt a b)))

> (power-digit-sum 2 1000)
1366
```

Задача 5.25 number-letter-counts.lisp

If the numbers 1 to 5 are written out in words: one, two, three, four, five, then there are $3 + 3 + 5 + 4 + 4 = 19$ letters used in total. If all the numbers from 1 to 1000 (one thousand) inclusive were written out in words, how many letters would be used?

NOTE: Do not count spaces or hyphens. For example, 342 (three hundred and forty-two) contains 23 letters and 115 (one hundred and fifteen) contains 20 letters. The use of "and" when writing out numbers is in compliance with British usage.

Если записать числа от 1 до 5 английскими словами (one, two, three, four, five), то используется $3 + 3 + 5 + 4 + 4 = 19$ букв. Сколько букв понадобится для записи всех чисел от 1 до 1000 (one thousand) включительно?

Примечание: Не считайте пробелы и дефисы. Например, число 342 (three hundred and forty-two) состоит из 23 букв, число 115 (one hundred and fifteen) – из 20 букв. Использование "and" при записи чисел соответствует правилам британского английского.

Решение 5.25.1

```
(defun remove-extras (s)
  (read-from-string
   (concatenate
    'string "("
    (delete-if #'(lambda (x)
                    (or (equal x #\,)
                        (equal x #\-)
                        (equal x #\)))
                s) ")"))))

(defun number-letter-counts (n m)
  (loop for a from n to m
        sum
        (reduce
         #' +
         (mapcar
          #'length
          (mapcar
           #'string
           (remove-extras
            (format nil "~r" a))))))))

> (number-letter-counts 1 5)
19
> (number-letter-counts 1 1000)
21124
```

Задача 5.26 sum-cube.lisp

Составить программу вычисления суммы кубов чисел от n до m .

Решение 5.26.1

```
(defun sum-cube (n m) (loop for a from n to m sum (expt a 3)))

> (sum-cube 25 55)
2281600
```

Решение 5.26.2

```
(defun sum-cube (n m)
  (cond ((= n m) (expt n 3))
        ((+ (expt n 3) (sum-cube (1+ n) m)))))

> (sum-cube 25 55)
2281600
```

Решение 5.26.3

```
(defun sum-cube (n m)
  (if (= n m)
```

```

(expt n 3)
(+ (expt n 3) (sum-cube (1+ n) m)))

> (sum-cube 25 55)
2281600

```

Решение 5.26.4

```

(defun sum-cube (n m)
  (+ (expt n 3)
     (if (= n m) 0 (sum-cube (1+ n) m))))

> (sum-cube 25 55)
2281600

```

Решение 5.26.5

```

(defun sum-cube (n m)
  (+ (* n n n) (if (= n m) 0 (sum-cube (+ n 1) m))))

> (sum-cube 25 55)
2281600

```

Задача 5.27 *product.lisp*

With respect to the lives and the fortunes ANSI Standard documents save, they're dirt cheap. with respect to what they make possible, they're also dirt cheap. note that ANSI standards also cost way too much compared to toilet paper, and they're pretty bad quality as toilet paper goes, too. I recently bought a fountain pen. it cost the equivalent of about 1500 throw-away ball pens. 1500 ball pens would have made me very frustrated, but this sleek, elegant pen made me happy. I also choose Common Lisp.

Erik Naggum

Определить функцию, возвращающую произведение двух целых положительных чисел (использовать суммирование).

Решение 5.27.1

```

(defun product (n m)
  (if (zerop n) 0 (+ m (product (1- n) m))))

> (product 8 17)
136

```

Решение 5.27.1

```
(defun product (n m &optional (ac 0))
  (if (zerop n) ac (product (1- n) m (+ ac m))))

> (product 8 17)
136
```

Решение 5.27.3

```
(defun product (n m)
  (loop for a from 1 to n sum m))

> (product 8 17)
136
```

Задача 5.28 *sum-input.lisp*

Определить функцию, возвращающую сумму двух введенных чисел.

Решение 5.28.1

```
(defun sum-input ()
  (write (+ (read) (read))))

> (sum-input)
2 3
5
5
```

Задача 5.29 *newton.lisp*

Определить функцию, использующую метод Ньютона для вычисления квадратного корня. Метод Ньютона вычисления квадратного корня из числа x начинается с выбора начального приближения z . Это приближение считается достаточно точным, если $(x - z^2) \leq \text{error}$, где error – некоторая заранее определенная погрешность. В противном случае более точным приближением будет $1/2 * (z + x/z)$, которое можно вычислить и точно так же подвергнуть проверке на погрешность. Пример: вычисления квадратного корня из числа 5, начальное приближение 1, уровень допустимой погрешности 0.05.

Решение 5.29.1

```
(defun newton (x e &optional (z 1))
  (if (> (abs (- x (expt z 2))) e)
      (newton x e (* (+ z (/ x z)) .5))
      z))

> (newton 5 0.05)
2.2380953
> (newton 7 0.05)
```

2.6548913

Решение 5.29.2

```
(defun newton (x e &optional (z 1))
  (if (> (abs (- x (* z z))) e)
      (newton x e (* (+ z (/ x z)) .5))
      z))

> (newton 5 0.05)
2.2380953
> (newton 7 0.05)
2.6548913
```

Задача 5.30 div-abs.lisp

Решить задачу через хвостовую рекурсию: деление нацело через вычитание.

Решение 5.30.1

```
(defun div-abs (m n) (div m n 0))

(defun div (m n c)
  (cond ((< m n) c)
        ((div (- m n) n (1+ c)))))

> (div-abs 27 6)
4
```

Задача 5.31 digit.lisp

Дано целое положительное число. Определить функцию, возвращающую k-ую цифру.

Решение 5.31.1

```
(defun digit (n m)
  (nth (1- n) (mapcar #'parse-integer
                      (map 'list #'string (write-to-string m)))))

> (digit 3 12345)
3
```

Решение 5.31.2

```
(defun digit (n m) (nth (- n 1) (int-digit* m nil)))

(defun int-digit* (n ac)
  (if (zerop n) ac
      (int-digit* (truncate n 10) (cons (rem n 10) ac))))
```



```
> (digit 3 12345)
3
```

Задача 5.32 *caesar-encipher.lisp*

Дана строка из букв латинского алфавита, цифр, знаков препинания и ключ для шифрования. Написать программу, реализующую шифрование – сдвиг по ключу букв данной строки – шифр Цезаря, в котором каждый символ в открытом тексте заменяется символом, находящимся на некотором постоянном числе позиций левее или правее него в алфавите. Например, в шифре со сдвигом вправо на 3, А была бы заменена на D, В станет Е, и так далее. Назван в честь римского императора Гая Юлия Цезаря, использовавшего его для секретной переписки со своими генералами.

Решение 5.32.1

```
(defun z (h k
  &aux
  (c (char-code h))
  (b (or (when (<= 97 c 122) 97)
        (when (<= 65 c 90) 65))))
  (if b (code-char (+ (mod (+ (- c b) k) 26) b))
    h))

(defun caesar-encipher (s k)
  (map 'string #'(lambda (c) (z c k)) s))

> (caesar-encipher "The quick brown fox jumps over the lazy dog" 23)
"Qeb nrfzh yoltk clu grjmp lsbo qeb ixwv ald"
```

Задача 5.33 *caesar-decipher.lisp*

Дана строка из букв латинского алфавита, цифр и знаков препинания, зашифрованная кодом Цезаря, и ключ для расшифровки. Написать программу, реализующую дешифрование.

Решение 5.33.1

```
(defun z (h k
  &aux
  (c (char-code h))
  (b (or (when (<= 97 c 122) 97)
        (when (<= 65 c 90) 65))))
  (if b (code-char (+ (mod (+ (- c b) k) 26) b))
    h))

(defun caesar-encipher (s k)
  (map 'string #'(lambda (c) (z c k)) s))

(defun caesar-decipher (s k)
```

```
(caesar-encipher s (- k))
```

```
> (caesar-decipher "Qeb nrfzh yoltk clu grjnp lsbo qeb ixwv ald" 23)
"The quick brown fox jumps over the lazy dog"
```

Задача 5.34 *ionescu.lisp*

Проверить решение из пьесы Эжена Ионеско «Урок» (La leçon, 1951):

Учитель: Послушайте, мадмуазель, если вы не способны понять азы, первоосновы арифметики, вы никогда не станете грамотным инженером. И уж тем более — преподавателем в высшей политехнической школе или в высшем дошкольном учреждении. Все это, бесспорно, сложно, очень и очень отвлеченно... разумеется... но как же, без глубоких знаний основ, вы сможете сосчитать в уме — а это самое малое, что требуется от рядового инженера, — сколько будет, ну, скажем, если три миллиарда семьсот пятьдесят пять миллионов девятьсот девяносто восемь тысяч сто пятьдесят один умножить на пять миллиардов сто шестьдесят два миллиона триста три тысячи пятьсот восемь?

Ученица: Это будет девятнадцать квинтиллионов триста девяносто квадриллионов два триллиона восемьсот сорок четыре миллиарда двести девятнадцать миллионов сто шестьдесят четыре тысячи пятьсот восемь...

Учитель (удивленно) : Нет. Кажется, не так. Должно получиться девятнадцать квинтиллионов триста девяносто квадриллионов два триллиона восемьсот сорок четыре миллиарда двести девятнадцать миллионов сто шестьдесят четыре тысячи пятьсот девять...

Ученица: Нет... пятьсот восемь...

Учитель (сосчитав в уме, с нарастающим изумлением) : Да... Вы правы... ответ, действительно... (Невнятно бормочет.) квадриллионов... триллионов... миллиардов... миллионов... (Разборчиво.) Сто шестьдесят четыре тысячи пятьсот восемь... (Ошеломленно.) Но каким образом вы это вычислили, если вам недоступны простейшие приемы арифметического мышления?

Ученица: Очень просто. Поскольку я не могу положиться на свое арифметическое мышление, я взяла и выучила наизусть все результаты умножения, какие только возможны.

Решение 5.34.1

```
> (* 3755998151 5162303508)
19389602430948813708
> (format nil "~r" (* 3755998151 5162303508))
"nineteen quintillion, three hundred and eighty-nine quadrillion, six hundred and two trillion, four hundred and thirty billion, nine hundred and forty-eight million, eight hundred and thirteen thousand, seven hundred and eight"
```

Задача 5.35 concord-mask-string.lisp

Определить функцию для сопоставления введенного шаблона (маски) и строки (слова). Под маской, в данном случае, подразумевается комбинация известных букв слова и знаков вопроса вместо неизвестных букв. Для замены неизвестного количества неизвестных букв используется знак звездочка *. Функция с двумя переменными: первая – собственно, шаблон, вторая – строка для проверки соответствия. Шаблон строится по следующим правилам: символ * заменяет любые символы, символ ? заменяет только один любой символ, количество * и ? в шаблоне не ограничено. Например, при введенном шаблоне L?*S**P строка LISP будет ему соответствовать.

Решение 5.35.1

```
(defun mems (s p)
  (member (car s) (cdr p) :test #'equal))

(defun check (p s)
  (cond ((null s) t)
        ((or (equal (car p) (car s))
              (equal (car p) "?")
              (check (cdr p) (cdr s)))
         ((when (equal (car p) "**")
              (if (mems s p)
                  (if (equal (car s) (cadr s))
                      (check (cdr (mems s p)) (cddr s))
                      (check (cdr (mems s p)) (cdr s)))
                  (check p (cdr s)))))))

(defun squeeze (p &optional z)
  (when p (if (equal (car p) "**")
              (if z
                  (squeeze (cdr p) z)
                  (cons "" (squeeze (cdr p) t)))
              (cons (car p) (squeeze (cdr p) nil)))))

(defun cleave (s) (map 'list #'string s))

(defun concord-mask-string (p s)
  (when (> (length s) 0)
    (check (squeeze (cleave p)) (cleave s))))

> (concord-mask-string "L?*S**P" "LISP")
T
> (concord-mask-string "L?*S**P" "LCISP")
T
> (concord-mask-string "L?*S**P" "LISRP")
T
> (concord-mask-string "L?*S**P" "LLISP")
T
```

Задача 5.36 alphabet-to-words.lisp

Common Lisp (a unification of various commercial and open flavors of Lisp) is a general purpose language that offers the ability to write in a mix of paradigms – functional, object-oriented, procedural, rule-based, and so on. The original purpose of Lisp was to tackle symbolic representation of natural language. It works well for that because one can natively, and thus without a bunch of obscuring overhead, represent sentences as a list of sentence fragments, and fragments as lists of symbols. People often think it stops there, because “LISP” = “List Processing”. However, most flavors of Lisp have a full complement of data structures – trees (lists of lists), multi-dimensional arrays, string objects (with no buffer overflow issues), hash tables whose keys can be distinguished by object identity or by a more generic sense of being ‘equal’, small integers, integers with unlimited size, complex numbers, and so on. I’ve used Lisp at various jobs to analyze health insurance claims, handle resource management, and manage queues of RFPs. It was 10x more efficient in terms of programmer time and code-base size than the other languages used at those same companies. I mention that so you don’t accept the trope that Lisp is for AI and nothing else. You’ll often hear that claim phrased as “right tool for the job”, but that’s probably short for, “Um, it’s easier to hire another Java programmer.”

Eric Moss

Алфавит языка содержит n букв. Создать функцию, которая определяет количество слов, которые можно составить из k букв этого алфавита при условии, что буквы в словах могут повторяться.

Решение 5.36.1

```
(defun alphabet-to-words (n k) (expt n k))
```

```
> (alphabet-to-words 26 10)
141167095653376
```

Решение 5.36.2

```
(defun alphabet-to-words (n k) (format nil "~r" (expt n k)))
```

```
> (alphabet-to-words 26 10)
"one hundred and forty-one trillion, one hundred and sixty-seven bil-
lion, ninety-five million, six hundred and fifty-three thousand, three
hundred and seventy-six"
```

Задача 5.37 *binomial.lisp*

I would recommend printing out a cheat sheet and having it on your desk when you're editing. I print out cheat sheets for many development related topics (SVN, Mercurial, CSS, Emacs, etc.) and I put them into a three-ring binder that is always within arm's reach... For example, if you want to copy a rectangle of text and you don't know how. Instead of taking a shortcut using some other technique, take step back. First, look up the sequence of operations and key combinations, then try until it works. Once it does, practice a few times. The next time you have to do the same thing, if you don't remember, repeat the process. Once you've done this a few times, it'll stick. Also, ...recording macros are key. Learn how to use them.

Todd Lucas

Имеется слово, состоящее из n различных символов. Написать программу, которая определяет, сколько различных слов может быть образовано из k символов данного слова при условии, что символы в словах не повторяются.

Решение 5.37.1

```
(defun binomial (n k)
  (if (or (zerop k) (= n k)) 1
      (+ (binomial (1- n) (1- k))
         (binomial (1- n) k))))
```

```
> (binomial 26 10)
5311735
```

Решение 5.37.2

```
(defun binomial (n k) (format nil "~r" (binomial- n k)))

(defun binomial- (n k)
  (if (or (zerop k) (= n k)) 1
      (+ (binomial- (1- n) (1- k))
         (binomial- (1- n) k))))
```

```
> (binomial 26 10)
"five million, three hundred and eleven thousand, seven hundred and
thirty-five"
```

Задача 5.38 *sum-cube.lisp*

And you're right: we were not out to win over the Lisp programmers; we were after the C++ programmers. We managed to drag a lot of them about halfway to Lisp.

Guy Steele

Напишите программу вычисления суммы кубов чисел от 25 до 125.

Решение 5.38.1

```
(defun sum-cube (n m)
  (loop for a from n to m sum (* a a a)))

> (sum-cube 25 125)
61925625
> (sum-cube 2 4)
99
```

Решение 5.38.2

```
(defun sum-cube (n m)
  (loop for a from n to m sum (expt a 3)))

> (sum-cube 25 125)
61925625
> (sum-cube 2 4)
99
```

Решение 5.38.3

```
(defun sum-cube (n m)
  (if (> n m) 0
      (+ (expt n 3) (sum-cube (1+ n) m))))

> (sum-cube 25 125)
61925625
> (sum-cube 2 4)
99
```

Решение 5.38.4

```
(defun sum-cube (n m &optional (ac 0))
  (if (> n m) ac
      (sum-cube (1+ n) m (+ (expt n 3) ac))))
```

```
> (sum-cube 25 125)
61925625
> (sum-cube 2 4)
99
```

Задача 5.39 *cnt.lisp*

Даны буква и строка. Определить функцию, возвращающую количество вхождений буквы в строку.

Решение 5.39.1

```
(defun cnt (a s)
  (loop for e across s
        when (eql e a) count e))

> (cnt #\a "aaaaaabbbbb")
6
> (cnt #\b "aaaaaabbbbb")
5
```

Решение 5.39.2

```
(defun cnt (a s) (count a s))

> (cnt #\a "aaaaaabbbbb")
6
> (cnt #\b "aaaaaabbbbb")
5
```

Задача 5.40 *max-palindrome.lisp*

A well-defined problem is half solved.

Michael Osborne

Определить функцию нахождения наибольшего палиндрома из произведений двух 3-значных чисел.

Решение 5.40.1

```
(defun max-palindrome (n m &aux (p 1) ac)
  (loop for a from n to m nconc
        (loop for b from n to m do
              (setf p (* a b))
              (when (palindrome p) (push p ac))))
  (reduce #'max ac))

(defun palindrome (n &aux (s (write-to-string n))
  (string= s (reverse s)))
```

```
> (max-palindrome 900 999)
906609
```

Задача 5.41 *product-primes.lisp*

Lisp is still #1 for key algorithmic techniques such as recursion and condensation.

Verity Stob

Определить функцию, возвращающую произведение всех простых чисел, не превышающих n .

Решение 5.41.1

```
(defun primep (n)
  (loop for a from 2 to (isqrt n)
        never (zerop (mod n a))))

(defun product-primes (n m &aux (p 1))
  (loop for a from n to m do
    (when (primep a) (setf p (* a p)))))

p)

> (time (product-primes 2 2000))
Real time: 0.0 sec.
Run time: 0.0 sec.
Space: 51140 Bytes
2893746419629456018526156696863524385571983891016355059219786875335346
0318409910335671322024504568938219397902846655662330092838642686864126
5517885761774500699854169800840685305650571905326393891147443503522793
8605840715914528796557757343239934472802299505162081684562428597045395
4504302947072301688306212986518898330744575994413178756377775223653404
2838736571825768147309875879568648761848492474692269075737993820510968
8470597977501377385796805835070414876850171495246279160636443750860689
9949529898134977654032079673848327197271244957392886939014687867827855
1706732964746955815021418849063826580384371871911662113251288424597019
0515058116574554626291842232351682026357515627185226065166986223960251
1888493439932617900609311758000459181619937785387630516653871526948649
6913965338299018641563718412655206993166415109405802877992948356769154
670
> (time (product-primes 2 20000))
Real time: 0.1720104 sec.
Run time: 0.15625 sec.
Space: 3796940 Bytes
GC: 7, GC time: 0.0 sec.
2171111972612485690304537347277148754525044224281933984018705661760140
3452555450169262307092011415739232700388204991185338221890627409626901
9313276586369845497781454624070983158737452258281929261302225954258375
6735523397997641680702221104950878783909974527786207287130926293373498
9451364595898745108703078017499333239988202181259107324042898169522225
```


0849289398208212008553740111827878958142793373572730472306679637646634
 3380543543261371062619175779528188644921842311860787876809647897385083
 3624205500718841030726155882087829774510409060423402343937434146774378
 2722570328543574807830071596428711081066735282099845933537597254312726
 6645165842409629866907980288447681045452635827372867977547180434870587
 5666405247936776698493354572567212480405721393035135216815004572397425
 14670176314919177369454650797593710395632272729086685782367135669016074
 9170720917137946363397879070554192432299139182038669264275143055550204
 0219340385850396288532218819888671283784688743408716306665039425090571
 8970529077513134725645570646139609985093540871860313089837214180056289
 282308199035811142072833761772208795911094953026626401958053987507991
 9848292609182871396468454266098657524363459130294312850712404792994192
 7297028613486440103340171399964672700752273532371127557368190619783767
 5342274957588009165704400481200984869134636284123358776339239921348024
 7963812388535520353161608727632779471785550175348066387156014218046
 1578431387485545954729607890614267767376833595472767330015505106018073
 8481615992068364577049360653655428582896096464168874828292155182708143
 2718228093554958706208584370514636119419021149633271916998859251971767
 5222081779254355980769180727508992728065045914764463541758231795509608
 6231453030610748483730901576680209366425557285266644184259318249840282
 6010582272581320015275038133341733868553167250574772882445037989378207
 7737347493505112511588492349198093102066123526589529119290554520830250
 7884493441311269493939765053860798008846110614201209484035895816435088
 5762185556016874563550280921508279642495218779817523048807744640998797
 1597690768773403337673849599566322739062528243997822569720956978733271
 5074816067465343047854128636441750707128097075261202908720360877689034
 3626886324471768233972438410122982776772905941880279391099590131675412
 5507476268819968113085935527889561792711275179209179401641950726245036
 7302882900381871776561425168467835412486183976960770060858410348389847
 1644192368826777860460915348746604053106052802997235323481122825255356
 5334739523157625632012674495458782873983394478157640357627384699815667
 8331023581542992198519398382485805857750450637462232993135806619080951
 8640051968859440410863462271697607239161796059899365339344524904506961
 1264144953792874345076546903077160043379343212775247639929348323582596
 4502571018087519338654882915930717557426213332435326992958766569535451
 4399813837319187266419374005389822515014855744838845032367118350228482
 5000418662041292725640493315817342011229401290300078250040375319082327
 2396815045665956996391580770623371387823014310940312277815604126164618
 5428196942395007362158202946387783245850166476830931544922353144095112
 579951302939175944362666840716626010781090491279304940736557425204606
 8130127021342047258274591409384429261134921816754175858549581131774831
 4896065878523840287644509911942960876513262983386153120990764756591500
 45182162624896617075333004442450393877638760469247346451233352895357
 306858328050828159106272650809885964939571297364922119002625330388
 0353054409591909018042601211004751844083777250404143838614062813855663
 8893558319648086608338782870755268782632849537870522561185247225256686
 6087597373486852359527772857993046521046796044980502093089546857353411
 46246549772455376184593175786217721387152098922651560341128644288182
 21519044122347370389685980069854632120987210061846077660910887279168295
 1460625779946550885780203287516695155181725448486082657611654343859950
 7827913357230031753787470754503264808376181938916952290065882875472855
 6240111723099775424139050439024387303350519974701580872966916532435833
 4181824833005695371274595824998309344669509326614435604136160087146362

7292398813580925043621265876391440222744623905799048600136413563615428
8262932601765089086012106679290838863976197278428741326667040043623681
1940789799541331033467018673488374387209317210565879194526183400146544
1419309559779274459771576538897641519547417970838220353083664250739675
7365399504154088221792323624351045274179429345098689751478039908059674
9230331540221378901681694100966529848691816412899398030914220454013491
7172738577088027830841070674090541329811061007311515946592855607535516
5823105152779131321487177726691301906242066782524520294343795224575725
0604749687571408838500654100361860700006743080259677718848440464884125
4961536987058987865004222279584278911834762170677860823792539212119241
25000224734395859551919161880404296485726644641713257500302322724146596192
3299871304639484346426554475889857312223209514035381642941691919401444
1625778186900458751981186713033324826137713494509445186908727869651197
1270213227238409891526956173199503276108734159724028837708737878154752
918538057496038252029026539458794975333776457570030302322724146596192
5455596243486320918943212987491367999531011513353554008129403165865091
1412591277372798486174122618831490964124246013999053520110024538012997
4019465786792382885423229115659383421254331939018251081761287985904190
3662775242932456362635646785454165479355229012491863530292823254634938
6855692862495768449826700643255797917147257733252445538651154683880932
3419415917254077620972102698878168318889494201298036355663833798815022
3342568750104401242986419428833974147804654489840374211521925628718987
5102762894369842699311903259645713324504548213994237313649146480197663
9181654007900409776102435040792761287563133886567896853977117539843847
3784969797944472490862073591090750882086465471697697300129818456360719
6018174260201475015183284393424983691251515593353195828562795931717402
6051613070050527553612268595565938990874231120765384481316926015833649
5836238857634478694579918630351057565667239095977629393377791301135946
7922598218722410994885031177912313547459742072418486877632782254681301
6849656651032644199895134231186636082763961871063876331163312811412582
3015321534697780998974507608227490585096155416154642860104468580279855
0775577621436770780782704213610445797334505746690421433384145936063960
8443605635420932264284660708204528026086803022817029433766325069799989
9363115411826840075196727432244378024114048778688685999576528440977570
6357258491621044586686892673724384330495777073102232240985198719469094
5040402091748919125338209768202183237611655412092944446157193035773087
9853462967475009972685750205757465259776660617982786582321634932761234
9642593626025259038264312046208935213475851116339790137859692577319102
5282041083025154752009382152458825857148230335041988996314918164271031
1665113403066697966206466383392155766734241277191149002679103599456015
6021233407549642355412550058170165168815556053005648434771573546863505
7362300586107670770275151552709626606779787691299301306578535354886885
3095074901188221683616280996387266525392567875335212891338790418620000
217458340362312927836729861792177402023745562152941302437310110096101
3482890157443871947332431671949748564364853588724396284950998709701520
4861287005963935320788036461350715238601046361952389861710230730705019
5029076743276433236413327148753201705393247387226654797425762524828367
484526562969912998464502762139828024680758404239795485210152257775002
4901129386803634236763268426448991993959508816789129872705186168308409
5448619691994592036781763039745891225924426967072070260080000558977140
7223018035767386806771648259527519610009735844700533802987137924917929
7622566980341318333901806815944562644302493360558917308243950295656098
0896118059890356371941093484296551828801213510762035599617239021618466

```

7484044246882305149451301631038854580470446423946457481538096275449963
2768828948801622026465200727565996090158079516687537195427482506453987
6805934184651720207816858530185592573681597762177463302227475716862181
6687014183413923284363675613124807488615209182879734086309164352145902
8816181107257808495330622769897081799441060938685623302801715688020567
8846162406160834975748780551737943669784562997243666605258056493321138
2932263078239845695805730037930404438396827618424630632005872635141916
3084080948068625362104521630580513294192921737344854660893550599018609
6133696361893925298989844566857459289896680568872319184265051714806750
0529456295905716186661938992966290506336197001388370738653279584668442
8812531654384160675060347565598647371024218210547418934149079877027266
77701846426003576304909348635505518391245956960690309930389490

```

Задача 5.42 *compare-triplets.lisp*

Lisp was the language of old MIT and the early days of AI. It is the sort of language that inspires religious devotion among people who don't care about writing commercial software or starting up start-ups but do care a lot about programming. It is an old and weird and beautiful language that takes purity and simplicity of language design to almost a mathematical extreme.

Joe Wezorek

Определить функцию, которая принимает результаты двух спортсменов - введенные с клавиатуры построчно две тройки чисел, разделенные пробелами: a1, a2, a3 и b1, b2, b3, а выводит их суммы в баллах через пробел: a b (если a1 > b1 - то первому начисляется один балл, если a1 < b1 - то балл присуждается второму).

Пример ввода:

```

7 23 44
5 23 45

```

Образец вывода:

```
1 1
```

Решение 5.42.1

```

(defun compare-triplets (&aux (w (w-read)) (v (w-read)))
  (loop for a in w for b in v
    if (> a b) count a into x
    else if (> b a) count b into y
    finally (princ x) (princ " ") (princ y)))

(defun w-read () (read-from-string
  (concatenate 'string

```

```
"(" (read-line) ")")))
```

```
> (compare-triplets)
7 23 44
5 23 45
1 1
NIL
```

Задача 5.43 birthday-cake-candles.lisp

С клавиатуры последовательно вводятся: в первой строке n (количество свече на торте) и во второй строке высоты свечек – числа, разделенные пробелами. Именинник может задуть лишь самые высокие свечки. Определить функцию, выводящую на экран количество задутых свечек.

Пример ввода:

```
10
1 5 2 5 3 5 4 5 3 5
```

Образец вывода:

```
5
```

Решение 5.43.1

```
(defun birthday-cake-candles (&aux
                               (w (loop repeat (read)
                                     collect (read))))
  (format t "~A" (count (reduce #'max w) w)))
```

```
> (birthday-cake-candles)
10
1 5 2 5 3 5 4 5 3 5
5
NIL
```

Задача 5.44 product-row.lisp

Определить функцию, возвращающую значение произведения ряда целых чисел от n до m .

Решение 5.44.1

```
(defun product-row (n m)
  (reduce #'* (loop for a from n to m collect a)))
```

```
> (product-row 5 7)
210
```

Решение 5.44.2

```
(defun product-row (n m)
  (cond ((> n m) 1)
        (t (* n (product-row (1+ n) m)))))
```

```
> (product-row 5 7)
210
```

Решение 5.44.3

```
(defun product-row (n m)
  (if (> n m) 1 (* n (product-row (1+ n) m))))
```

```
> (product-row 5 7)
210
```

Задача 5.45 *count-hv.lisp*

hobz: у моего папы дома кластер и 8 мониторов...
eSyr: привет, лейн.

bash.im/quote/106922

Определить функцию, вычисляющую количество вхождений слова по вертикали и горизонтали. Например, слово: "bc", файл test.txt:

```
abcbdace
acaac
aabca
aacbc
```

Решение 5.45.1

```
(defun count-hv (s path &aux (w (read-lines-chars path)))
  (+ (count-h s (car w))
     (count-h s (loop for a in (mapcar
                                #'reverse
                                (transpose (cadr w)))
                       collect (coerce a 'string)))))
```

```
(defun count-h (z w) (loop for a in w sum (cnt z a)))
```

```
(defun cnt (a s &optional (p 0)
  &aux (z (search a s :start2 p)))
  (if z (1+ (cnt a s (1+ z))) 0))
```

```
(defun read-lines-chars (path &aux u v)
  (with-open-file (s path :direction :input)
    (do ((line (read-line s nil :eof)
                (read-line s nil :eof)))
        ((eql line :eof))
      (setf u (cons line u))
      (setf v (cons (loop for c across line
```

```

                                collect c)
                                v))))
(list u v))

(defun transpose (m) (apply #'mapcar #'list m))

> (count-hv "bc" "c:/temp/test.txt")
7

```

Задача 5.46 common-words.lisp

Some people WITHIN the Lisp community think that Lisp is not as popular as it deserves to be because there are flaws in the Common Lisp language. The perceived solution to the problem is thus to create a better Lisp dialect. Frankly, if this were true, then no single other language would have a single follower, given the number of flaws in other languages compared to the flaws of Common Lisp. One typical attempt to create a better Lisp was Dylan (more like Scheme than like Lisp, really) which defines a parenthesis-free syntax for a Lisp-like language, thus essentially depriving Lisp of one of its perhaps greatest advantages, namely the nearly one-to-one mapping between surface syntax and internal representation of the code, an essential feature for making macros possible. And Dylan is no more popular than Common Lisp.

Robert Strandh

Два файла. Определить функцию, записывающую в третий файл общие слова без повторов. Например, файлы:

```

test1.in:
abcd, abcd.
bcde, bcde.
cdef, cdef.

```

```

test2.in:
bcde, bcde.
cdef, cdef.
defg, defg.

```

```

test.out:
CDEF BCDE

```

Решение 5.46.1

```

(defun common-words (p1 p2 p &aux v1 v2)
  (with-open-file (s1 p1 :direction :input)
    (do ((line (read-line s1 nil :eof) (read-line s1 nil :eof)))
        ((eql line :eof))
      (setf v1 (nconc (list-string line) v1))))
  (with-open-file (s2 p2 :direction :input)
    (do ((line (read-line s2 nil :eof) (read-line s2 nil :eof)))
        ((eql line :eof))
      (setf v2 (nconc (list-string line) v2))))
  (with-open-file (s p
                    :direction :output
                    :if-exists :supersede
                    :if-does-not-exist :create)
    (format s "~{~a~^ ~}"
      (remove-duplicates (intersection v1 v2))))

(defun list-string (s)
  (read-from-string
   (concatenate
    'string "(" (delete-if-not #'(lambda (x)
                                   (or (alpha-char-p x)
                                       (equal x #\space)
                                       (equal x #\-)))
                                s) ")"))))

> (common-words "d:/test1.in" "d:/test2.in" "d:/test.out")
NIL

```

Задача 5.47 number-of-psychiatric-hospital-patients.lisp

Всероссийская олимпиада школьников. Типовые задания I (школьного) этапа по математике. 8 класс: В психиатрической больнице есть главный врач и много сумасшедших. В течение недели каждый сумасшедший один раз в день кусал кого-нибудь (возможно и себя). В конце недели оказалось, что у каждого из больных по два укуса, а у главного врача – сто укусов. Сколько сумасшедших в больнице?

Решение 5.47.1

; $7x = 2x + 100 \rightarrow x = 20$

```

(defun number-of-psychiatric-hospital-patients (n m)
  (/ m (- 7 n)))

> (number-of-psychiatric-hospital-patients 2 100)
20
> (number-of-psychiatric-hospital-patients 3 100)
25
> (number-of-psychiatric-hospital-patients 4 100)
100/3
> (number-of-psychiatric-hospital-patients 5 100)
50

```

```
> (number-of-psychiatric-hospital-patients 6 100)
100
> (number-of-psychiatric-hospital-patients 6 200)
200
```

Задача 5.48 *pw.lisp*

What programming languages should every software engineer know? I hate to say it... LISP. It would lead to a higher standard in programming across the board. Im... Not saying everyone should use LISP, just know it.

Simon Millward

Определить функцию возведения в степень при помощи циклов.

Решение 5.48.1

```
(defun pw (x k)
  (let ((y 1))
    (loop repeat k do (setf y (* x y)))
    y))
```

```
> (pw 2 4)
16
```

Решение 5.48.2

```
(defun pw (x k &aux (y 1))
  (loop repeat k do (setf y (* x y))
    finally (return y)))
```

```
> (pw 2 4)
16
```

Решение 5.48.3

```
(defun pw (x k)
  (loop with y = 1
    repeat k do (setf y (* x y))
    finally (return y)))
```

```
> (pw 2 4)
16
```


Структура 6 (функция АТОМ АТОМ) > (СПИСОК)

Everytime some VS user comes by and watches me or watches someone else that knows-what-they-are-doing-with-emacs, they instantly want to convert. It's something that words can't convey. You have to really see it in action to truly appreciate the leaps in efficiency that you can gain.

—

Computer science education cannot make anybody an expert programmer any more than studying brushes and pigment can make somebody an expert painter.

Eric Raymond

Languages shape the way we think, or don't.

Erik Naggum

Задача 6.1 *same-words.lisp*

Даны два предложения. Вывести слова, которые входят в каждое из них, т.е. совпадающие.

Решение 6.1.1

```
(defun sw (s)
  (read-from-string (concatenate 'string "(" s " ")))

(defun same-words (a b)
  (intersection (sw a) (sw b)))

> (same-words "a b c" "b c d")
(B C)
```

Задача 6.2 *quotient&remainder.lisp*

Определите функцию, которая возвращает целое и остаток от деления положительных чисел. Не использовать функции деления и остатка.

Решение 6.2.1 (автор - VH, www.cyberforum.ru)

```
(defun quotient&remainder (x y)
  (cond ((zerop y) nil)
        ((< x y) (cons 0 x))))
```

```

(t ((lambda (res)
      (cons (1+ (car res)) (cdr res)))
    (quotient&remainder (- x y) y))))

> (quotient&remainder 5 3)
(1 . 2)

```

Задача 6.3 *random-row.lisp*

Создайте функцию, порождающую по заданным числам m , k список, состоящий из случайного количества случайных чисел из промежутка от m до k .

Решение 6.3.1

```

(defun random-row (m k &optional (n (random 25)) &aux (a (random k)))
  (cond ((zerop n) nil)
        ((> a m) (cons a (random-row m k (1- n))))
        ((random-row m k n))))

> (random-row 40 50)
(49 48 48 48 46 49)
> (random-row 40 50)
(44 48 43 43 41 48 48 46 47 43 49 43 46 48 42 47 47 44 48)
> (random-row 40 50)
(48 45 46 48 46 43 45 47 42 42 49 48 42)

```

Задача 6.4 *onion.lisp*

Определите функцию, которая строит список "луковица" с уровнем вложенности n для параметра a . Например, при $n=4$, $a=0$ функция должна возвращать список `((((0))))`.

Решение 6.4.1

```

(defun onion (a n)
  (if (zerop n) a (list (onion a (1- n)))))

> (onion 0 4)
((((0))))

```

Решение 6.4.2

```

(defun onion (a n)
  (loop repeat n
        for e = (list a) then (list e)
        finally (return e)))

> (onion 0 4)
((((0))))

```

Задача 6.5 primes.lisp

You become a serious programmer by going through a stage where you are fully aware of the degree to which you know the specification, meaning both the explicit and the tacit specification of your language and of your problem. "hey, it works most of the time" is the very antithesis of a serious programmer, and certain languages can only support code like that. over time, you get a good gut feeling for the failure mode of your assumptions, and when you can trust that you are unlikely to make grossly invalid assumptions, the dangers that people run into in the absence of that trust vanish in a puff of standardization: it's the kind of trust you buy from somebody who claims to be conformant. non-conformance is about violating this sense of trust. certain languages support serious programmers, and others don't. e.g., I don't think it is at all possible to become a serious programmer using Visual Basic or Perl. if you think hard about what Perl code will do on the borders of the known input space, your head will explode. if you write Perl code to handle input problems gracefully, your programs will become gargantuan: the normal failure mode is to terminate with no idea how far into the process you got or how much of an incomplete task was actually performed. in my view, serious programmers don't deal with tools that force them to hope everything works.

Erik Naggum

Определить функцию, которая принимает числа n и m и составляет список простых чисел в диапазоне от n до m .

Решение 6.5.1

```
(defun check (x)
  (when (> x 1) (loop for i from 2 to (isqrt x)
                     never (zerop (mod x i)))))

(defun primes (n m &optional w)
  (cond ((= n m) (nreverse w))
        ((or (= n 2) (check n)) (primes (1+ n) m (cons n w)))
        ((primes (1+ n) m w))))
```

```
> (primes 1 30)
(2 3 5 7 11 13 17 19 23 29)
> (primes 6 30)
(7 11 13 17 19 23 29)
```

Решение 6.5.2

```
(defun primes(x n &optional w)
  (cond ((= x n) (nreverse w))
        ((or (= x 2) (when (> x 1)
                          (loop for i from 2 to (isqrt x)
                                never (zerop (mod x i))))))
        (primes (1+ x) n (cons x w)))
  ((primes (1+ x) n w))))

> (primes 1 30)
(2 3 5 7 11 13 17 19 23 29)
> (primes 6 30)
(7 11 13 17 19 23 29)
```

Решение 6.5.3

```
(defun check (n)
  (when (> n 1) (loop for a from 2 to (isqrt n)
                      never (zerop (mod n a)))))

(defun primes(n m)
  (loop for a from n to m
        when (or (check a) (= a 2)) collect a))

> (primes 1 30)
(2 3 5 7 11 13 17 19 23 29)
> (primes 6 30)
(7 11 13 17 19 23 29)
```

Решение 6.5.4

```
(defun check (n)
  (when (> n 1) (loop for a from 2 to (isqrt n)
                      never (zerop (mod n a)))))

(defun primes (n m)
  (if (> n 2)
      (loop for a from n to m
            when (check a) collect a)
      (cons 2 (loop for a from n to m
                    when (check a) collect a))))

> (primes 1 30)
(2 3 5 7 11 13 17 19 23 29)
> (primes 6 30)
(7 11 13 17 19 23 29)
```

Задача 6.6 numbers.lisp

Определить функцию, которая порождает все целые числа, отвечающие условию $a \leq x \leq b$.

Решение 6.6.1

```
(defun numbers (a b)
  (when (<= a b) (cons a (numbers (1+ a) b))))

> (numbers 4 8)
(4 5 6 7 8)
```

Решение 6.6.2

```
(defun numbers (a b)
  (loop for x from a to b collect x))

> (numbers 4 8)
(4 5 6 7 8)
```

Задача 6.7 count-dictionary.lisp

Написать программу вывода словарных слов в порядке их встречаемости в заданном тексте. Вывести частоту встречаемости словарных слов в тексте в виде десятичной дроби. Текст и словарные слова вводить из отдельных файлов.

Решение 6.7.1

```
d:/text.txt:
aaaa bbbb cccc dddd
eeee ffff gggg aaaa
bbbb hhhh iiii aaaa
aaaa bbbb cccc

d:/dict.txt:
aaaa bbbb
cccc

(defun sw (s)
  (read-from-string (concatenate 'string "(" s ")")))

(defun count-dictionary (pt pd &aux text dict)
  (with-open-file (s pd :direction :input)
    (do ((line (read-line s nil :eof) (read-line s nil :eof)))
        ((eql line :eof))
        (setf dict (nconc dict (sw line)))))
  (with-open-file (s pt :direction :input)
    (do ((line (read-line s nil :eof) (read-line s nil :eof)))
        ((eql line :eof))
        (setf text (nconc text (sw line)))))
```

```

(let ((m (length text)))
  (loop for a in dict
        collect (list a (float (/ (count a text) m))))))

> (count-dictionary "d:/text.txt" "d:/dict.txt")
((AAAA 0.26666668) (BBBB 0.2) (CCCC 0.13333334))

```

Решение 6.7.2

```

d:/text.txt:
aaaa bbbb cccc dddd
eeee ffff gggg aaaa
bbbb hhhh iiii aaaa
aaaa bbbb cccc aaal
bbb2 ccc1 aaa2
d:/dict.txt:
aaaa aaal aaa2
bbbb bbb1 bbb2
cccc ccc1 ccc2

(defun sw (s)
  (read-from-string (concatenate 'string "(" s ")")))

(defun count-dictionary (pt pd &aux text dict)
  (with-open-file (s pd :direction :input)
    (do ((line (read-line s nil :eof) (read-line s nil :eof)))
        ((eql line :eof))
        (setf dict (cons (sw line) dict))))
  (with-open-file (s pt :direction :input)
    (do ((line (read-line s nil :eof) (read-line s nil :eof)))
        ((eql line :eof))
        (setf text (nconc text (sw line)))))
  (let ((m (length text)))
    (loop for a in (reverse dict)
          collect (list (car a)
                        (float (/ (loop for e in a
                                         sum (count e text))
                                m))))))

> (count-dictionary "d:/text.txt" "d:/dict.txt")
((AAAA 0.31578946) (BBBB 0.21052632) (CCCC 0.15789473))

```

Решение 6.7.3 * с выводом содержания словаря и текста

```

d:/text.txt:
aaaa bbbb cccc dddd
eeee ffff gggg aaaa
bbbb hhhh iiii aaaa
aaaa bbbb cccc aaal
bbb2 ccc1 aaa2

d:/dict.txt:
aaaa aaal aaa2

```

```
bbbb bbb1 bbb2
cccc cccl ccc2
```

```
(defun sw (s)
  (read-from-string (concatenate 'string "(" s ")")))

(defun count-dictionary (pt pd &aux text dict)
  (with-open-file (s pd :direction :input)
    (do ((line (read-line s nil :eof) (read-line s nil :eof)))
        ((eql line :eof))
        (setf dict (cons (sw line) dict))))
  (with-open-file (s pt :direction :input)
    (do ((line (read-line s nil :eof) (read-line s nil :eof)))
        ((eql line :eof))
        (setf text (nconc text (sw line)))))
  (print dict)
  (print text)
  (let ((m (length text)))
    (loop for a in (reverse dict)
          collect (list (car a)
                        (float (/ (loop for e in a
                                         sum (count e text))
                                m))))))

> (count-dictionary "d:/text.txt" "d:/dict.txt")
((CCCC CCC1 CCC2) (BBBB BBB1 BBB2) (AAAA AAA1 AAA2))
(AAAA BBBB CCCC DDDD EEEE FFFF GGGG AAAA BBBB HHHH IIII AAAA AAAA BBBB
CCCC AAA1 BBB2 CCC1 AAA2)
((CCCC 0.15789473) (BBBB 0.21052632) (AAAA 0.31578946))
```

Задача 6.8 max-divided.lisp

Определить функцию, возвращающую числа из диапазона n m , имеющие наибольшее количество делителей.

Решение 6.8.1

```
(defun num-divisors (n)
  (loop for a from 2 to (/ n 2) when (zerop (rem n a)) count a))

(defun divided (n m ac &aux (z (num-divisors n)))
  (cond ((= n m) ac)
        ((> z (cadar ac)) (divided (1+ n) m `((,n ,z))))
        ((= z (cadar ac)) (divided (1+ n) m (cons `((,n ,z) ac)))
        ((divided (1+ n) m ac))))

(defun max-divided (n m)
  (divided (1+ n) m `((,n ,(num-divisors n)))))

> (max-divided 2 100)
((96 10) (90 10) (84 10) (72 10) (60 10))
> (max-divided 2 1000)
```

```
((840 30))
> (max-divided 2 10000)
((9240 62) (7560 62))
```

Решение 6.8.2

```
(defun num-divisors (n)
  (loop for a from 2 to (/ n 2) when (zerop (rem n a)) count a))

(defun divided (n m ac &aux (z (num-divisors n)))
  (if (= n m)
      ac
      (divided (1+ n)
                 m
                 (cond ((> z (cadar ac)) `((,n ,z)))
                       ((= z (cadar ac)) (cons `((,n ,z) ac))
                       (t ac))))))

(defun max-divided (n m)
  (divided (1+ n) m `((,n ,(num-divisors n)))))

> (max-divided 2 100)
((96 10) (90 10) (84 10) (72 10) (60 10))
> (max-divided 2 1000)
((840 30))
> (max-divided 2 10000)
((9240 62) (7560 62))
```

Решение 6.8.3

```
(defun num-divisors (n)
  (loop for a from 2 to (/ n 2) when (zerop (rem n a)) count a))

(defun divided (n m ac &aux (z (num-divisors n)) (d (cadar ac)))
  (if (= n m)
      (values (mapcar #'car ac) d)
      (divided (1+ n) m (cond ((> z d) `((,n ,z)))
                        ((= z d) (cons `((,n ,z) ac))
                        (t ac))))))

(defun max-divided (n m)
  (divided (1+ n) m `((,n ,(num-divisors n)))))

> (max-divided 2 100)
(96 90 84 72 60)
10
> (max-divided 2 1000)
(840)
30
> (max-divided 2 10000)
(9240 7560)
62
```


Задача 6.9 generate.lisp

Определить функцию, которая возвращает список-«маску» заданной длины.

Решение 6.9.1

```
(defun generate (n a)
  (cond ((zerop n) nil)
        (t (cons a (generate (1- n) a)))))
```

```
> (loaf 'x 4)
(X X X X)
```

Решение 6.9.2

```
(defun generate (n a)
  (if (zerop n) nil (cons a (generate (1- n) a))))
```

```
> (loaf 'x 4)
(X X X X)
```

Решение 6.9.3

```
(defun generate (n a)
  (when (plussp n) (cons a (generate (1- n) a)))))
```

```
> (generate 5 'x)
(X X X X X)
```

Решение 6.9.4

```
(defun generate (n a)
  (make-list n :initial-element a))
```

```
R> (generate 5 'x)
(X X X X X)
```

Решение 6.9.5

```
(defun generate (n a)
  (loop for b from 1 to n collect a))
```

```
> (generate 5 'x)
(X X X X X)
```

Решение 6.9.6

```
(defun generate (n a)
  (loop repeat n collect a))
```

```
> (generate 5 'x)
(X X X X X)
```

Задача 6.10 onion-right.lisp

Определить функцию, которая по заданному уровню вложенности *n* генерирует список вида (a (a (a (a (a (a)))))).

Решение 6.10.1

```
(defun onion-right (a n)
  (if (= n 1) (list a) (list a (onion-right a (1- n)))))

> (onion-right 'a 7)
(A (A (A (A (A (A (A (A)))))))
```

Решение 6.10.2

```
(defun onion-right (a n)
  (if (= n 1) `(,a) `(,a , (onion-right a (1- n)))))

> (onion-right 'a 7)
(A (A (A (A (A (A (A (A)))))))
```

Задача 6.11 rand-add-pair.lisp

Построить два списка из пяти случайных чисел и получить список, попарно суммируя элементы полученных списков.

Решение 6.11.1

```
(defun rand (r n)
  (loop for i from 1 to n
        collect (random r)))

(defun rand-add-pair (n m)
  (mapcar #'(+ (rand n m) (rand n m)))

> (rand-add-pair 100 5)
(74 59 123 85 29)
> (rand-add-pair 100 5)
(132 35 97 46 118)
```

Задача 6.12 rand-even.lisp

Разработать программу, которая выводит список чисел, а затем, используя рекурсию, выводит четные числа.

Решение 6.12.1

```
(defun rand (r n)
  (loop for i from 1 to n
        collect (random r)))
```

```

(defun even (w &aux (a (car w)))
  (cond ((null w) nil)
        ((evenp a) (cons a (even (cdr w))))
        (t (even (cdr w)))))

(defun rand-even (r n &aux (v (rand r n)))
  (values v (even v)))

> (rand-even 10 20)
(8 3 5 2 8 6 5 3 4 8 0 2 7 6 7 5 4 7 5 9)
(8 2 8 6 4 8 0 2 6 4)

```

Решение 6.12.2

```

(defun rand (r n)
  (loop repeat n collect (random r)))

(defun even (w &aux (a (car w)))
  (cond ((null w) nil)
        ((evenp a) (cons a (even (cdr w))))
        (t (even (cdr w)))))

(defun rand-even (r n &aux (v (rand r n)))
  (values v (even v)))

> (rand-even 10 20)
(1 4 7 7 5 9 6 6 4 5 8 0 5 2 0 5 0 2 3 4)
(4 6 6 4 8 0 2 0 0 2 4)

```

Решение 6.12.3

```

(defun rand (r n) (loop repeat n collect (random r)))

(defun even (w &aux (a (car w)))
  (when w (if (evenp a)
              (cons a (even (cdr w)))
              (even (cdr w)))))

(defun rand-even (r n &aux (v (rand r n)))
  (values v (even v)))

> (rand-even 10 20)
(5 1 2 3 4 1 2 4 8 1 1 0 1 9 0 9 7 0 9 5)
(2 4 2 4 8 0 0 0)

```

Решение 6.12.4

```

(defun rand-even (r n &aux (v (loop repeat n collect (random r))))
  (values v (labels ((even (v)
                      (when v (if (evenp (car v))
                                  (cons (car v) (even (cdr v)))
                                  (even (cdr v))))))
            (even v)))))

```

```
> (rand-even 10 20)
(1 7 8 8 4 8 4 9 7 8 1 6 4 3 0 6 1 4 1 4)
(8 8 4 8 4 8 6 4 0 6 4 4)
```

Решение 6.12.5

```
(defun rand-even (r
                  n
                  &aux
                  (v (loop repeat n
                           collect
                           (random r))))
  (values v
          (labels ((even (v
                           &aux
                           (a (car v))
                           (d (cdr v)))
                     (when v
                       (if (evenp a)
                           (cons a (even d))
                           (even d))))))
          (even v))))

> (rand-even 10 20)
(0 9 5 8 1 8 8 4 8 5 8 2 1 3 3 9 5 4 8 4)
(0 8 8 8 4 8 8 2 4 8 4)
```

Задача 6.13 *list-a-n.lisp*

Определить функцию, которая строит список из n одинаковых элементов.

Решение 6.13.1

```
(defun list-a-n (a n) (when (plusp n) (cons a (list-a-n a (1- n)))))

> (list-a-n 10 5)
(10 10 10 10 10)
```

Решение 6.13.2

```
(defun list-a-n (a n) (loop repeat n collect a))

> (list-a-n 10 5)
(10 10 10 10 10)
```

Задача 6.14 *add.lisp*

Определить функцию, добавляющую заданное параметром число a к каждому элементу числового списка w .

Решение 6.14.1

```
(defun add (w a)
  (cond ((null w) nil)
        (t (cons (+ (car w) a) (add (cdr w) a)))))

> (add '(1 2 3) 10)
(11 12 13)
```

Решение 6.14.2

```
(defun add (w a)
  (if w
      (cons (+ (car w) a) (add (cdr w) a))
      nil))

> (add '(1 2 3) 10)
(11 12 13)
```

Решение 6.14.3

```
(defun add (w a)
  (when w (cons (+ (car w) a) (add (cdr w) a))))

> (add '(1 2 3) 10)
(11 12 13)
```

Решение 6.14.4

```
(defun add (w a)
  (and w (cons (+ (car w) a) (add (cdr w) a))))

> (add '(1 2 3) 10)
(11 12 13)
```

Решение 6.14.5

```
(defun add (w a)
  (mapcar #'(lambda (e) (+ e a)) w))

> (add '(1 2 3) 10)
(11 12 13)
```

Решение 6.14.6

```
(defun add (w a)
  (loop for e in w collect (+ e a)))

> (add '(1 2 3) 10)
(11 12 13)
```

Задача 6.15 *positions.lisp*

The advantage I've seen is it teaches you to build up a set of primitive functions, which you can use over and over to code higher level functions, though macros make up a significant part of that, since they allow you to create new control structures.

Mark Miller

Для текстов *w* и *v* определить функцию, подсчитывающую сколько раз *v* встречается в *w* и на каких позициях.

Решение 6.15.1

```
(defun positions (w v &optional ac (n 0) (m (length v))
  &aux (z (search v w)))
  (cond ((null z) (values n (reverse ac)))
        (t (positions (subseq w (+ z m))
                       v
                       (cons (+ z
                               (if ac (car ac) 0)
                               (if ac m 0))
                             ac)
                       (1+ n)
                       m))))

> (positions "abrakadabramaram" "ra")
3
(2 9 13)
```

Решение 6.15.2

```
(defun positions (w v &optional ac (n 0) (m (length v))
  &aux (z (search v w)))
  (if z
      (positions (subseq w (+ z m))
                  v
                  (cons (+ z
                          (if ac (car ac) 0)
                          (if ac m 0))
                        ac)
                  (1+ n)
                  m)
      (values n (reverse ac))))

> (positions "abrakadabramaram" "ra")
3
(2 9 13)
```

Решение 6.15.3

```

(defun positions (w v &optional ac (m (length v))
                &aux (z (search v w)))
  (if z
      (positions (subseq w (+ z m))
                  v
                  (cons (+ z
                          (if ac (car ac) 0)
                          (if ac m 0))
                        ac)
                  m)
      (values (loop for a in ac count a)
              (reverse ac))))

> (positions "abrakadabramaram" "ra")
3
(2 9 13)

```

Решение 6.15.4

```

(defun positions (w v &optional ac (m (length v))
                &aux (z (search v w)))
  (if z
      (positions (subseq w (+ z m))
                  v
                  (cons (+ z
                          (if ac (car ac) 0)
                          (if ac m 0))
                        ac)
                  m)
      (values (count-if #'identity ac)
              (reverse ac))))

> (positions "abrakadabramaram" "ra")
3
(2 9 13)

```

Задача 6.16 largest-palindrome-product.lisp

A palindromic number reads the same both ways. The largest palindrome made from the product of two 2-digit numbers is $9009 = 91 \times 99$. Find the largest palindrome made from the product of two 3-digit numbers.

Число-палиндром с обеих сторон (справа налево и слева направо) читается одинаково. Самое большое число-палиндром, полученное умножением двух двузначных чисел – $9009 = 91 \times 99$. Найдите самый большой палиндром, полученный умножением двух трёхзначных чисел.

Решение 6.16.1

```

(defun make-palindrome (n &aux (m (write-to-string n)))
  (parse-integer (concatenate 'string m (reverse m))))

```

```

(defun largest-divisor (p m &aux (z (/ p m)))
  (cond ((= m 99) nil)
        ((integerp z) (list p m z))
        (t (largest-divisor p (1- m)))))

(defun largest-palindrome-product (n m)
  (loop for a downfrom n to m
    when (largest-divisor (make-palindrome a) 999)
    collect (largest-divisor (make-palindrome a) 999)))

> (largest-palindrome-product 998 900)
((998899 781 1279) (995599 319 3121) (994499 121 8219) (992299 539
1841) (991199 359 2761) (990099 803 1233) (989989 989 1001) (987789
809 1221) (986689 209 4721) (981189 901 1089) (980089 641 1529)
(979979 979 1001) (977779 863 1133) (975579 799 1221) (973379 827
1177) (972279 759 1281) (970079 319 3041) (969969 969 1001) (967769
907 1067) (966669 363 2663) (965569 671 1439) (963369 789 1221)
(961169 649 1481) (960069 619 1551) (959959 959 1001) (958859 671
1429) (954459 933 1023) (953359 121 7879) (952259 913 1043) (951159
779 1221) (949949 949 1001) (947749 319 2971) (946649 451 2099)
(944449 253 3733) (943349 449 2101) (942249 231 4079) (941149 737
1277) (940049 457 2057) (939939 939 1001) (938839 121 7759) (937739
523 1793) (934439 323 2893) (929929 929 1001) (928829 187 4967)
(925529 121 7649) (924429 771 1199) (922229 649 1421) (921129 813
1133) (919919 919 1001) (917719 209 4391) (916619 253 3623) (914419
857 1067) (913319 869 1051) (912219 847 1077) (911119 733 1243)
(909909 909 1001) (907709 461 1969) (906609 993 913) (905509 313 2893)
(903309 883 1023))

```

Решение 6.16.2

```

(defun make-palindrome (n &aux (m (write-to-string n)))
  (parse-integer (concatenate 'string m (reverse m))))

(defun largest-divisor (p m &aux (z (/ p m)))
  (cond ((= m 99) nil)
        ((integerp z) (list p m z))
        (t (largest-divisor p (1- m)))))

(defun largest-palindrome-product (n m)
  (remove-if
    #'(lambda (a) (> (/ (caddr a) 1000) 1))
    (loop for a downfrom n to m
      when (largest-divisor (make-palindrome a) 999)
      collect (largest-divisor (make-palindrome a) 999))))

> (largest-palindrome-product 998 900)
((906609 993 913))
> (largest-palindrome-product 998 800)
((906609 993 913) (888888 962 924) (886688 968 916) (861168 932 924)
(855558 957 894) (853358 982 869) (840048 946 888) (828828 966 858)
(824428 914 902) (821128 946 868) (819918 909 902) (809908 932 869))

```



```
(807708 957 844) (804408 968 831) (802208 946 848) (801108 924 867))
```

Решение 6.16.3

```
(defun make-palindrome (n &aux (m (write-to-string n)))
  (parse-integer (concatenate 'string m (reverse m))))

(defun largest-divisor (p m &aux (z (/ p m)))
  (cond ((= m 99) nil)
        ((integerp z) (list p m z))
        (t (largest-divisor p (1- m)))))

(defun largest-palindrome-product (n m)
  (car (remove-if
        #'(lambda (a) (> (/ (caddr a) 1000) 1))
        (loop for a downfrom n to m
              when (largest-divisor (make-palindrome a) 999)
                collect (largest-divisor (make-palindrome a) 999)))))

> (largest-palindrome-product 998 800)
(906609 993 913)
```

Решение 6.16.4

```
(defun make-palindrome (n &aux (m (write-to-string n)))
  (parse-integer (concatenate 'string m (reverse m))))

(defun largest-divisor (p m &aux (z (/ p m)))
  (cond ((= m 99) nil)
        ((integerp z) (list p m z))
        (t (largest-divisor p (1- m)))))

(defun largest-palindrome-product (n m)
  (car (remove-if
        #'(lambda (a) (> (/ (caddr a) 999) 1))
        (loop for a from n downto m
              when (largest-divisor (make-palindrome a) 999)
                collect (largest-divisor (make-palindrome a) 999)))))

> (largest-palindrome-product 998 800)
(906609 993 913)
```

Решение 6.16.5

```
(defun make-palindrome (n &aux (m (write-to-string n)))
  (parse-integer (concatenate 'string m (reverse m))))

(defun largest-divisor (p m &aux (z (/ p m)))
  (when (> m 99) (if (and (integerp z) (< z 1000))
                    (list (list p m z))
                    (largest-divisor p (1- m)))))

(defun largest-palindrome-product (n m)
```

```

(car (loop for a from n downto m
          nconc (largest-divisor (make-palindrome a) 999))))
> (largest-palindrome-product 998 800)
(906609 993 913)

```

Решение 6.16.6

```

(defun make-palindrome (n &aux (m (write-to-string n)))
  (parse-integer (concatenate 'string m (reverse m))))

(defun largest-divisor (p m &aux (z (/ p m)))
  (when (> m 99) (if (and (integerp z) (< z 1000))
    (list p m z)
    (largest-divisor p (1- m)))))

(defun largest-palindrome-product
  (n m &aux (z (largest-divisor (make-palindrome n) 999)))
  (when (> n m) (if z z (largest-palindrome-product (1- n) m))))

> (largest-palindrome-product 998 800)
(906609 993 913)

```

Задача 6.17 word-set-difference.lisp

Найти все слова, которые есть в первом предложении и отсутствуют во втором.

Решение 6.17.1

```

(defun word-set-difference (a b)
  (symbols-to-strings (set-difference (string-list a)
    (string-list b))))

(defun symbols-to-strings (w)
  (when w (cons (string (car w)) (symbols-to-strings (cdr w)))))

(defun string-list (s)
  (read-from-string
    (concatenate
      'string "(" (delete-if-not #'(lambda (x)
        (or (alpha-char-p x)
          (equal x #\space)
          (equal x #\-)))
        s) ")")))

> (word-set-difference "aa b-b cc" "cc dd")
("AA" "B-B")

```

Решение 6.17.2

```

(defun word-set-difference (a b)

```

```

(loop for a in (set-difference (string-list a)
                               (string-list b))
      collect (string a)))

(defun string-list (s)
  (read-from-string
   (concatenate
    'string "(" (delete-if-not #'(lambda (x)
                                   (or (alpha-char-p x)
                                       (equal x #\space)
                                       (equal x #\-)))
                                s) ")"))))

> (word-set-difference "aa b-b cc" "cc dd")
("AA" "B-B")

```

Решение 6.17.3

```

(defun word-set-difference (a b)
  (mapcar #'string (set-difference (string-list a)
                                   (string-list b))))

(defun string-list (s)
  (read-from-string
   (concatenate
    'string "(" (delete-if-not #'(lambda (x)
                                   (or (alpha-char-p x)
                                       (equal x #\space)
                                       (equal x #\-)))
                                s) ")"))))

> (word-set-difference "aa b-b cc" "cc dd")
("AA" "B-B")

```

Задача 6.18 eeq.lisp

Определить функцию, которая возвращает список цифр натурального числа, совпадающих n раз.

Решение 6.18.1

```

(defun int-digits (n)
  (loop for c across (write-to-string n)
        collect (digit-char-p c)))

(defun eeq (n m
            &aux
              (s (int-digits m))
              (u (remove-duplicates s)))
  (loop for c in u
        when (= (count c s) n)
        collect c))

```

```
> (eeq 3 1112)
(1)
> (eeq 3 111222)
(1 2)
```

Решение 6.18.2

```
(defun eeq (n m
  &aux
    (s (loop for c across (write-to-string m)
      collect (digit-char-p c)))
    (u (remove-duplicates s)))
  (loop for c in u
    when (= (count c s) n)
    collect c))
```

```
> (eeq 3 1112)
(1)
> (eeq 3 111222)
(1 2)
```

Решение 6.18.3

```
(defun eeq (n m
  &aux
    (s (write-to-string m))
    (u (remove-duplicates s)))
  (loop for c across u
    when (= (count (string c) s :test #'string-equal) n)
    collect (digit-char-p c)))
```

```
> (eeq 3 1112)
(1)
> (eeq 3 111222)
(1 2)
```

Решение 6.18.4

```
(defun int-digits (n &optional ac)
  (if (zerop n)
    ac
    (int-digits (truncate n 10)
      (cons (rem n 10) ac))))

(defun e (w n
  &optional (v (remove-duplicates w))
  &aux (a (car v)))
  (cond ((null v) nil)
        ((= (count a w) n) (cons a (e w n (cdr v))))
        ((e w n (cdr v)))))

(defun eeq (n m)
```

```

(e (int-digits m) n))

> (eeq 3 1112)
(1)
> (eeq 3 111222)
(1 2)

```

Решение 6.18.5

```

(defun int-digits (n &optional ac)
  (if (zerop n)
      ac
      (int-digits (truncate n 10)
                   (cons (rem n 10) ac))))

(defun e (w n
          &optional (v (remove-duplicates w))
          &aux (a (car v)))
  (when v (if (= (count a w) n)
               (cons a (e w n (cdr v)))
               (e w n (cdr v)))))

(defun eeq (n m)
  (e (int-digits m) n))

> (eeq 3 1112)
(1)
> (eeq 3 111222)
(1 2)

```

Решение 6.18.6

```

(defun int-digits (n)
  (mapcar #'digit-char-p
           (coerce (write-to-string n) 'list)))

(defun e (w n
          &optional (v (remove-duplicates w))
          &aux (a (car v)))
  (when v (if (= (count a w) n)
               (cons a (e w n (cdr v)))
               (e w n (cdr v)))))

(defun eeq (n m)
  (e (int-digits m) n))

> (eeq 3 1112)
(1)
> (eeq 3 111222)
(1 2)

```

Решение 6.18.7

```

(defun int-digits (n)
  (mapcar #'digit-char-p
    (concatenate 'list (write-to-string n))))

(defun e (w n
  &optional (v (remove-duplicates w))
  &aux (a (car v)))
  (when v (if (= (count a w) n)
    (cons a (e w n (cdr v)))
    (e w n (cdr v)))))

(defun eeq (n m)
  (e (int-digits m) n))

> (eeq 3 1112)
(1)
> (eeq 3 111222)
(1 2)

```

Решение 6.18.8

```

(defun int-digits (n)
  (mapcar #'digit-char-p
    (map 'list #'identity (write-to-string n))))

(defun e (w n
  &optional (v (remove-duplicates w))
  &aux (a (car v)))
  (when v (if (= (count a w) n)
    (cons a (e w n (cdr v)))
    (e w n (cdr v)))))

(defun eeq (n m)
  (e (int-digits m) n))

> (eeq 3 1112)
(1)
> (eeq 3 111222)
(1 2)

```

Решение 6.18.9

```

(defun int-digits (n)
  (mapcar #'parse-integer
    (map 'list #'string (write-to-string n))))

(defun e (w n
  &optional (v (remove-duplicates w))
  &aux (a (car v)))
  (when v (if (= (count a w) n)
    (cons a (e w n (cdr v)))
    (e w n (cdr v)))))

```

```
(defun eeq (n m)
  (e (int-digits m) n))
```

```
> (eeq 3 1112)
(1)
> (eeq 3 111222)
(1 2)
```

Решение 6.18.10

```
(defun int-digits (n)
  (mapcar #'parse-integer
    (map 'list #'string (format nil "~d" n))))
```

```
(defun e (w n
  &optional (v (remove-duplicates w))
  &aux (a (car v)))
  (when v (if (= (count a w) n)
    (cons a (e w n (cdr v)))
    (e w n (cdr v)))))
```

```
(defun eeq (n m)
  (e (int-digits m) n))
```

```
> (eeq 3 1112)
(1)
> (eeq 3 111222)
(1 2)
```

Решение 6.18.11

```
(defun int-digits (n)
  (loop with r do (setf (values n r) (truncate n 10))
    collect r into ac until (zerop n)
    finally (return (nreverse ac))))
```

```
(defun e (w n
  &optional (v (remove-duplicates w))
  &aux (a (car v)))
  (when v (if (= (count a w) n)
    (cons a (e w n (cdr v)))
    (e w n (cdr v)))))
```

```
(defun eeq (n m)
  (e (int-digits m) n))
```

```
> (eeq 3 1112)
(1)
> (eeq 3 111222)
(1 2)
```

Задача 6.19 unique-digits.lisp

All you have to do is write one true sentence. Write the truest sentence that you know.

Ernest Hemingway

Определить функцию, которая возвращает список всех трехзначных чисел, в записи которых нет одинаковых цифр.

Решение 6.19.1

```
(defun unique-digits (n m)
  (loop for a from n to m
        when (equal (remove-duplicates
                     (write-to-string a))
                     (write-to-string a))
        collect a))
```

```
> (unique-digits 100 999)
(102 103 104... 984 985 986 987)
```

Решение 6.19.2

```
(defun unique-digits (n m
                      &optional ac
                      &aux (a (write-to-string n)))
  (cond ((> n m) (nreverse ac))
        ((equal (remove-duplicates a) a)
         (unique-digits (1+ n) m (cons n ac)))
        ((unique-digits (1+ n) m ac))))
```

```
> (unique-digits 100 999)
(102 103 104... 984 985 986 987)
```

Решение 6.19.3

```
(defun unique-digits (n m)
  (remove-if-not
   #'(lambda (a)
        (equal (remove-duplicates
                 (write-to-string a))
                 (write-to-string a)))
   (loop for a from n to m collect a)))
```

```
> (unique-digits 100 999)
(102 103 104... 984 985 986 987)
```

Решение 6.19.4

```
(defun int-digits (n &optional ac)
```



```

(if (zerop n)
    ac
    (int-digits (truncate n 10)
                 (cons (rem n 10) ac))))

(defun unique-digits (n m)
  (cond ((> n m) nil)
        ((equal (remove-duplicates
                  (int-digits n))
                  (int-digits n))
         (cons n (unique-digits (1+ n) m)))
        ((unique-digits (1+ n) m))))

> (unique-digits 100 999)
(102 103 104... 984 985 986 987)

```

Решение 6.19.5

```

(defun unique-digits ()
  (loop
    for a from 1 to 9
    nconc
    (loop
      for b upto 9
      nconc
      (loop
        for c upto 9
        unless (or (= a b) (= a c) (= b c))
        collect (+ (* a 100) (* b 10) c))))))

> (unique-digits)
(102 103 104... 984 985 986 987)

```

Задача 6.20 range.lisp

Определить функцию, формирующую список всех четных чисел больше n , но меньше m .

Решение 6.20.1

```

(defun range (n m)
  (cond ((or (<= (- m n) 1)
            (and (evenp n)
                  (= (- m n) 2)))
        nil)
        ((r n m nil))))

(defun r (n m f)
  (cond ((>= n m) nil)
        (f (cons n (r (+ n 2) m t)))
        ((evenp n) (cons (+ n 2) (r (+ n 4) m t)))
        ((cons (+ n 1) (r (+ n 3) m t)))))

```

```

> (range 3 7)
(4 6)
> (range 3 4)
NIL
> (range 3 5)
(4)
> (range 4 5)
NIL
> (range 2 4)
NIL

```

Решение 6.20.2

```

(defun range (n m) (when (and (> (- m n) 1)
                              (not (and (evenp n)
                                         (= (- m n) 2))))
  (r n m nil)))

(defun r (n m f)
  (cond ((>= n m) nil)
        (f (cons n (r (+ n 2) m t)))
        ((evenp n) (cons (+ n 2) (r (+ n 4) m t)))
        ((cons (+ n 1) (r (+ n 3) m t)))))

```

```

> (range 3 7)
(4 6)
> (range 3 4)
NIL
> (range 3 5)
(4)
> (range 4 5)
NIL
> (range 2 4)
NIL

```

Решение 6.20.3

```

(defun range (n m) (when (> (- m n) 1) (r n m)))

(defun r (n m)
  (loop for a from (1+ n) below m
        when (evenp a) collect a))

> (range 3 7)
(4 6)
> (range 3 4)
NIL
> (range 3 5)
(4)
> (range 4 5)
NIL
> (range 2 4)

```

NIL

Задача 6.21 *reversed-primes.lisp*

Lisp has been around for a long time, though it has been transformed considerably since it was first invented (some say 'discovered'). Today, it is a modern, multi-paradigm language which has perhaps the most sophisticated features of all general-purpose languages in use (the object system, macros, reader macros, conditions, etc, etc). The first question one might ask is then "If Lisp is so good, why is it not more popular?". People who ask such a question typically assume that "good" implies "popular", and thus look for some part of Lisp that is NOT good, which would explain why it is not popular. However, there is absolutely no reason to assume that "good" implies "popular", so the question is really quite naive.

Robert Strandh

Определить функцию, возвращающую в порядке убывания все простые числа из диапазона от n до m .

Решение 6.21.1

```
(defun primep (n)
  (loop for i from 2 to (isqrt n) never (zerop (rem n i))))

(defun reversed-primes (n m)
  (loop for a downfrom m to n
        when (primep a) collect a))

> (reversed-primes 2 201)
(199 197 193 191 181 179 173 167 163 157 151 149 139 137 131 127 113
109 107 103 101 97 89 83 79 73 71 67 61 59 53 47 43 41 37 31 29 23 19
17 13 11 7 5 3 2)
```

Задача 6.22 *betwixt.lisp*

Даны целые числа n и m . Определить функцию, которая возвращает все целые числа x , отвечающие условию $n < x < m$.

Решение 6.22.1

```
(defun betwixt (n m)
  (when (< n m)
```

```
(loop for a from (1+ n) to (1- m)
      collect a)))
```

```
> (betwixt 4 7)
(5 6)
```

Решение 6.22.2

```
(defun betwixt (n m)
  (when (< n m)
    (loop for a from (1+ n) below m
          collect a)))
```

```
> (betwixt 4 7)
(5 6)
```

Решение 6.22.3

```
(defun betwixt (n m)
  (when (< n m) (betwixt- n m)))

(defun betwixt- (n m &aux (z (1+ n)))
  (when (< z m) (cons z (betwixt- (1+ n) m))))
```

```
> (betwixt 4 7)
(5 6)
```

Решение 6.22.4

```
(defun betwixt (n m)
  (when (< n m) (betwixt- n m)))

(defun betwixt- (n m &optional ac &aux (z (1+ n)))
  (if (< z m) (betwixt- (1+ n) m (cons z ac))
      (nreverse ac)))
```

```
> (betwixt 4 7)
(5 6)
```

Задача 6.23 *set-exclusive-random.lisp*

Определить функцию, которая формирует два списка целых чисел определенной длины. Вывести на печать числа, которые входят в один из списков, но не входят в оба списка.

Решение 6.23.1

```
(defun set-exclusive-random (n m)
  (set-exclusive-or (random-row n) (random-row m)))

(defun random-row (n)
  (loop for i from 1 to n collect (random 1000)))
```

```
> (set-exclusive-random 100 100)
(964 92 938 359 205 501 457 196 274 454 990 868 77 804 445 219 546 962
915 30 196 129 919 925 233 138 503 81 246 678 367 897 961 980 766 827
647 839 558 991 271 278 8 198 767 159 269 702 272 889 983 792 532 987
558 148 848 170 109 574 64 818 773 445 906 550 746 491 271 489 173 930
167 56 146 395 219 724 418 167 211 842 264 976 85 518 807 961 369 582
936 719 931 166 475 878 689 328 902 534 910 829 694 517 473 880 430
916 114 393 356 673 892 882 458 299 31 371 528 294 10 788 720 684 426
1 102 548 895 562 988 790 227 630 656 353 540 803 372 771 338 356 596
472 333 512 469 812 498 946 531 753 32 259 859 139 877 764 593 758 57
822 922 895 176 810 942 883 475 515 215 268 637 653 442 105 907 549
559 510)
```

Задача 6.24 *intersection-random.lisp*

It is one of the oldest high level programming languages and - it is different that c-based languages (c++, java, c#). It has been used for commercial purposes, but a lot for academic/teaching purposes. It requires a bit different thinking, brings some functional concepts/paradigms, but imperative too. In LISP everything is a list (OK not literally everything), but the name comes from LISt Processor: (+ 1 2 3) is 6 ... So Polish prefix (or postfix) notation - something different. Top athletes practice other sport arts in order to prevent overtraining, psychological one - lack of motivation. Executing other motions makes the body more capable of processing, learning, correcting motions characteristic in main sport art. And you still do move. The same can be applied to learning (in this case LISP) - do it. Even if you are java/c# developer it will relax you, bring some new "cerebral motions/movements".

Miljenko Cvjetko

Определить функцию, которая формирует два списка целых чисел определенной длины. Вывести на печать числа, встречающиеся в обоих списках.

Решение 6.24.1

```
(defun intersection-random (n m)
  (intersection (random-row n) (random-row m)))

(defun random-row (n) (loop for i from 1 to n collect (random 1000)))

> (intersection-random 200 200)
```

```
(653 603 444 926 775 410 77 974 779 633 844 336 613 334 337 878 91 430
842 17 180 360 829 862 994 317 560 718 51 898 115 705 354 77 135 979
815)
```

Задача 6.25 *make-function-number-plus.lisp*

Google's QPX System (acquired by buying ITA Software) for searching air fares is a very large system written in Lisp. Public estimates are that it consists of 1/2 million lines of Lisp code. It is used in production as the back end of Google Flight Search, so it is also surely the Lisp program that consumes the most CPU cycles anywhere.

Stavros Macrakis

Определить функцию, которая создает набор функций от n до m подобных встроенной функции `1+`.

Решение 6.25.1

```
(defun make-function-number-plus (n m)
  (loop for a from n to m do
    (make-function (add-plus a) '(z) `(+ z ,a))))

(defun add-plus (a)
  (read-from-string
   (concatenate 'string (write-to-string a) "+")))

(defun make-function (name prms body)
  (eval `(defun ,name ,prms ,body)))

> (make-function-number-plus 2 10)
NIL
> (7+ 1000)
1007
```

Решение 6.25.2 (автор - helter, www.cyberforum.ru)

```
(defun make-function-number-plus (n m)
  (loop for a from n to m do
    (let ((a a))
      (setf (symbol-function (add-plus a))
            (lambda (z) (+ z a))))))

(defun add-plus (a)
  (intern (format nil "~A+" a)))

> (make-function-number-plus 2 10)
NIL
> (7+ 1000)
```

1007

Задача 6.26 *primes.lisp*

Lots of things have been written in Lisp, and most people are unaware that CLOS was the first ANSI Standard OO language. So some things written in Lisp: Interleaf - a large document processing system. Think documents like millions of pages and volumes. Lisp-Machines used Lisp for everything but micro-code. AutoCad uses Lisp as its scripting language and large parts of it are written in Lisp. Microsoft Bob, while a commercial failure for numerous reasons, was coded in Franz Common Lisp. Emacs is largely written in eLisp except for small parts written in portable C. NASA - Do a Google query "NASA Lisp" and you'll find lots of examples, including one spacecraft that was patched for a bug on the way to its destination bc the code onboard was in Lisp. So they could upload new code on the fly. The original Interface Builder that ended up on the NeXT computer and is now on Mac OS X and iOS was written in Expertelligence ExperLisp (a Common Lisp) for the Mac and Steve Jobs saw it and hired the person who wrote it to come to NeXT and create it for NeXTStep. NoteCards - One of the early HyperText systems. Ran on Xerox InterLisp. Lisp.

Sam Griffith Jr.

Определить функцию, формирующую список простых чисел на заданном интервале.

Решение 6.26.1

```
(defun primes (n m)
  (loop for a from n to m
        when (primep a) collect a))
```

```
(defun primep (n)
  (loop for a from 2 to (isqrt n)
        never (zerop (mod n a)))))
```

```
> (primes 1 100)
(1 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97)
```

Решение 6.26.2

```

(defun primes (n m)
  (cond ((> n m) nil)
        ((primep n) (cons n (primes (1+ n) m)))
        (t (primes (1+ n) m))))

(defun primep (n)
  (loop for a from 2 to (isqrt n)
        never (zerop (mod n a))))

> (primes 1 100)
(1 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89
97)

```

Решение 6.26.3

```

(defun primes (n m &optional ac)
  (cond ((> n m) (nreverse ac))
        ((primep n) (primes (1+ n) m (cons n ac)))
        (t (primes (1+ n) m ac))))

(defun primep (n)
  (loop for a from 2 to (isqrt n)
        never (zerop (mod n a))))

> (primes 1 100)
(1 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89
97)

```

Решение 6.26.4

```

(defun primes (n m &optional ac)
  (cond ((> n m) (nreverse ac))
        ((primep n) (primes (1+ n) m (cons n ac)))
        (t (primes (1+ n) m ac))))

(defun primep (n &optional (a 2) (b (isqrt n)))
  (cond ((> a b)
        ((zerop (mod n a)) nil)
        (t (primep n (1+ a) b))))

> (primes 1 100)
(1 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89
97)

```

Задача 6.27 compose.lisp

Определить функцию, которая по двум числам формирует список из двух элементов. Первый элемент – это результат целочисленного деления первого числа на второе, второй – остаток.

Решение 6.27.1


```
(defun compose (a b)
  (when (plusp b) (list (floor (/ a b)) (rem a b))))

> (compose 11 5)
(2 1)
> (compose 11 0)
NIL
```

Решение 6.27.2

```
(defun compose (a b)
  (unless (zerop b) (list (floor (/ a b)) (rem a b))))

> (compose 11 5)
(2 1)
> (compose 11 0)
NIL
```

Задача 6.28 *symmetric.lisp*

Определить функцию, которая позволяет перечислить все целые положительные числа из заданного диапазона состоящие из двух и более цифр и которые читаются слева направо и справа налево одинаково, например: 121, 5665, 988889.

Решение 6.28.1

```
(defun symmetric (a b)
  (loop for n from (if (< a 10) 10 a) to (if (< b 10) 10 b)
    when (smp n) collect n))

(defun smp (n &aux (s (write-to-string n)))
  (equal s (reverse s)))

> (symmetric 1 2000)
(11 22 33 44 55 66 77 88 99 101 111 121 131 141 151 161 171 181 191
202 212 222 232 242 252 262 272 282 292 303 313 323 333 343 353 363
373 383 393 404 414 424 434 444 454 464 474 484 494 505 515 525 535
545 555 565 575 585 595 606 616 626 636 646 656 666 676 686 696 707
717 727 737 747 757 767 777 787 797 808 818 828 838 848 858 868 878
888 898 909 919 929 939 949 959 969 979 989 999 1001 1111 1221 1331
1441 1551 1661 1771 1881 1991)
```

Задача 6.29 *fizzbuzz.lisp*

Напишите программу, которая выводит на экран числа от 1 до 100. При этом вместо чисел, кратных трем, программа должна выводить слово Fizz, а вместо чисел, кратных пяти — слово Buzz. Если число кратно пятнадцати, то программа должна выводить слово FizzBuzz.

Решение 6.29.1 (автор - tmpValue, www.cyberforum.ru)

```
(defvar *fizzbuzz*
  ("FizzBuzz" nil nil "Fizz" nil "Buzz" "Fizz"
   nil nil "Fizz" "Buzz" nil "Fizz" nil nil))

(defun fizzbuzz ()
  (loop :for i :from 1 :to 100
        :collect
        (let ((tmp (nth (mod i 15) *fizzbuzz*)))
          (if tmp tmp i))))

> (fizzbuzz)
(1 2 "Fizz" 4 "Buzz" "Fizz" 7 8 "Fizz" "Buzz" 11 "Fizz" 13 14
 "FizzBuzz" 16 17 "Fizz" 19 "Buzz" "Fizz" 22 23 "Fizz" "Buzz" 26 "Fizz"
 28 29 "FizzBuzz" 31 32 "Fizz" 34 "Buzz" "Fizz" 37 38 "Fizz" "Buzz" 41
 "Fizz" 43 44 "FizzBuzz" 46 47 "Fizz" 49 "Buzz" "Fizz" 52 53 "Fizz"
 "Buzz" 56 "Fizz" 58 59 "FizzBuzz" 61 62 "Fizz" 64 "Buzz" "Fizz" 67 68
 "Fizz" "Buzz" 71 "Fizz" 73 74 "FizzBuzz" 76 77 "Fizz" 79 "Buzz" "Fizz"
 82 83 "Fizz" "Buzz" 86 "Fizz" 88 89 "FizzBuzz" 91 92 "Fizz" 94 "Buzz"
 "Fizz" 97 98 "Fizz" "Buzz")
```

Решение 6.29.2

```
(defconstant *fizzbuzz*
  ("FizzBuzz" nil nil "Fizz" nil "Buzz" "Fizz"
   nil nil "Fizz" "Buzz" nil "Fizz" nil nil))

(defun fizzbuzz (n m)
  (loop with tmp for i from n to m do
        (setf tmp (nth (mod i 15) *fizzbuzz*))
        collect (cond (tmp) (i))))

> (fizzbuzz 1 100)
(1 2 "Fizz" 4 "Buzz" "Fizz" 7 8 "Fizz" "Buzz" 11 "Fizz" 13 14
 "FizzBuzz" 16 17 "Fizz" 19 "Buzz" "Fizz" 22 23 "Fizz" "Buzz" 26 "Fizz"
 28 29 "FizzBuzz" 31 32 "Fizz" 34 "Buzz" "Fizz" 37 38 "Fizz" "Buzz" 41
 "Fizz" 43 44 "FizzBuzz" 46 47 "Fizz" 49 "Buzz" "Fizz" 52 53 "Fizz"
 "Buzz" 56 "Fizz" 58 59 "FizzBuzz" 61 62 "Fizz" 64 "Buzz" "Fizz" 67 68
 "Fizz" "Buzz" 71 "Fizz" 73 74 "FizzBuzz" 76 77 "Fizz" 79 "Buzz" "Fizz"
 82 83 "Fizz" "Buzz" 86 "Fizz" 88 89 "FizzBuzz" 91 92 "Fizz" 94 "Buzz"
 "Fizz" 97 98 "Fizz" "Buzz")
```

Задача 6.30 lace.lisp

Определить функцию от двух аргументов *e* и *n*, возвращающую список из *n* раз повторенных списков, состоящих из одного элемента *e*.

Решение 6.30.1

```
(defun lace (e n)
  (cond ((zerop n) nil)
```

```
(t (cons (list e) (lace e (1- n))))))
```

```
> (lace 'a 4)
((A) (A) (A) (A))
```

Решение 6.30.2

```
(defun lace (e n)
  (when (plusp n) (cons ` (,e) (lace e (1- n)))))
```

```
> (lace 'a 4)
((A) (A) (A) (A))
```

Решение 6.30.3

```
(defun lace (e n)
  (loop repeat n collect ` (,e)))
```

```
> (lace 'a 4)
((A) (A) (A) (A))
```

Задача 6.31 *grow-circles.lisp*

Определить функцию от двух аргументов *e* и *n*, возвращающую список вида *(e (e) ((e)) ...)*, где вложенность последнего элемента равна *n*.

Решение 6.31.1

```
(defun grow-circles (e n)
  (cond ((zerop n) (list e))
        (t (cons e (grow-circles (list e) (1- n))))))
```

```
> (grow-circles 'i 5)
(I (I) ((I)) (((I))) ((((I)))))
```

Решение 6.31.2

```
(defun grow-circles (e n)
  (if (zerop n) ` (,e) (cons e (grow-circles ` (,e) (1- n)))))
```

```
> (grow-circles 'i 5)
(I (I) ((I)) (((I))) ((((I)))))
```

Структура 7 (функция (СПИСОК АТОМ) > АТОМ

We can use emacs to receive/send mails, compile/debug programs, edit various file formats with advanced features support, write latex papers, take notes/diaries, browse web page and so on. We can do too many things in emacs. It is not just a text editor. It has the ability to integrate many other unix/linux tools, just a little bit like perl. Vi is still the best text editor in linux, but it is just a text editor.

Yan Tang

Elegance is necessarily unnatural, only achievable at great expense. If you just do something, it won't be elegant, but if you do it and then see what might be more elegant, and do it again, you might, after an unknown number of iterations, get something that is very elegant.

Erik Naggum

Within a couple weeks of learning Lisp I found programming in any other language unbearably constraining.

Paul Graham

Задача 7.1 *elms-num.lisp*

Определить функцию, подсчитывающую число вхождений элемента в список.

Решение 7.1.1

```
(defun elms-num (n w &aux (a (car w)) (d (cdr w)))  
  (cond ((null w) 0)  
        ((equal n a) (1+ (elms-num n d)))  
        ((elms-num n d))))
```

```
> (elms-num 2 '(1 2 2 4 2 3))  
3
```

Решение 7.1.2

```
(defun elms-num (n w) (loop for a in w count (equal n a)))
```

```
> (elms-num 2 '(1 2 2 4 2 3))
3
```

Решение 7.1.3

```
(defun elms-num (n w)
  (count n w))

> (elms-num 2 '(1 2 2 4 2 3))
3
```

Задача 7.2 *check-number.lisp*

Проверить числовой список *w* на наличие числа *n*.

Решение 7.2.1

```
(defun check-number (n w)
  (cond ((null w) nil)
        ((= n (car w)) t)
        ((check-number n (cdr w)))))

> (check-number 3 '(1 2 3 4 5))
T
> (check-number 0 '(1 2 3 4 5))
NIL
```

Решение 7.2.2

```
(defun check-number (n w)
  (some #'(lambda (a) (= n a)) w))

> (check-number 3 '(1 2 3 4 5))
T
> (check-number 0 '(1 2 3 4 5))
NIL
```

Решение 7.2.3

```
(defun check-number (n w)
  (loop for i in w thereis (= n i)))

> (check-number 3 '(1 2 3 4 5))
T
> (check-number 0 '(1 2 3 4 5))
NIL
```

Решение 7.2.4

```
(defun check-number (n w)
  (member n w))
```

```
> (check-number 3 '(1 2 3 4 5))
(3 4 5)
> (check-number 0 '(1 2 3 4 5))
NIL
```

Решение 7.2.5

```
(defun check-number (n w)
  (find n w))

> (check-number 3 '(1 2 3 4 5))
3
> (check-number 0 '(1 2 3 4 5))
NIL
```

Задача 7.3 *insider.lisp*

Проверить, входит ли заданный элемент в многоуровневый список.

Решение 7.3.1

```
(defun insider (n w)
  (when w ((lambda (a d)
              (cond ((null w) nil)
                    ((listp a) (insider n a))
                    ((eql n a) t)
                    ((insider n d))))
    (car w) (cdr w))))

> (insider 0 '(1(2(4 5)3(6))))
NIL
> (insider 4 '(1(2(4 5)3(6))))
T
```

Задача 7.4 *near-last.lisp*

Написать функцию, которая возвращает предпоследний элемент списка.

Решение 7.4.1

```
(defun near-last (w)
  (car (last (nbutlast w))))

> (near-last '(1 2 3 4 5 6))
5
```

Решение 7.4.2

```
(defun near-last (w)
  (cadr (nreverse w)))
```

```
> (near-last '(1 2 3 4 5 6))
5
```

Задача 7.5 value.lisp

Прайс компьютерного магазина описывается списком (наименование фирма параметры цена): ((Терминал Acer 15 4500) (Терминал Samsung 17 7000) (Матплата Asus 238 2500) (Процессор Intel 2.5 3000)). Определить функцию, возвращающую стоимость товаров заданной фирмы.

Решение 7.5.1

```
(defun value (b w &optional (p 0) &aux (a (car w)))
  (if w
      (if (eq (cadr a) b)
          (value b (cdr w) (+ p (caddr a)))
          (value b (cdr w) p)) p))

> (value 'intel '((terminal acer 15 4500) (terminal samsung 17 7000)
(motherboard intel 238 2500) (processor intel 2.5 3000)))
5500
```

Решение 7.5.2

```
(defun value (b w &optional (p 0) &aux (a (car w)))
  (cond ((null w) p)
        ((eq (cadr a) b) (value b (cdr w) (+ p (caddr a))))
        ((value b (cdr w) p))))

> (value 'intel '((terminal acer 15 4500) (terminal samsung 17 7000)
(motherboard intel 238 2500) (processor intel 2.5 3000)))
5500
```

Решение 7.5.3

```
(defun value (b w)
  (loop for a in w when (eq (cadr a) b) sum (caddr a)))

> (value 'intel '((terminal acer 15 4500) (terminal samsung 17 7000)
(motherboard intel 238 2500) (processor intel 2.5 3000)))
5500
```

Решение 7.5.4

```
(defun value (f w)
  (loop for (name firm parameters price) in w
        when (eq firm f) sum price))

> (value 'intel '((terminal acer 15 4500) (terminal samsung 17 7000)
(motherboard intel 238 2500) (processor intel 2.5 3000)))
5500
```

Задача 7.6 count-goods .lisp

Прайс компьютерного магазина описывается списком (наименование фирма параметры цена): ((Терминал Acer 15 4500) (Терминал Sumsung 17 7000) (Матплата Asus 238 2500) (Процессор Intel 2.5 3000)). Определить функцию возвращающую количество единиц выбранного наименования.

Решение 7.6.1

```
(defun count-goods (m w &optional (n 0))
  (cond ((null w) n)
        ((eq (caar w) m) (count-goods m (cdr w) (1+ n)))
        ((count-goods m (cdr w) n))))

> (count-goods 'terminal '((terminal acer 15 4500) (terminal samsung
17 7000) (motherboard intel 238 2500) (processor intel 2.5 3000)))
2
```

Решение 7.6.2

```
(defun count-goods (n w)
  (loop for (name firm parameters price) in w
        when (eq name n) count name))

> (count-goods 'terminal '((terminal acer 15 4500) (terminal samsung
17 7000) (motherboard intel 238 2500) (processor intel 2.5 3000)))
2
```

Решение 7.6.3

```
(defun count-goods (n w)
  (count n w :key #'car))

> (count-goods 'terminal '((terminal acer 15 4500) (terminal samsung
17 7000) (motherboard intel 238 2500) (processor intel 2.5 3000)))
2
```

Задача 7.7 last-n.lisp

Дан список. Определить функцию, которая выдавала бы элемент списка по заданному номеру с конца (функции length и nth не использовать).

Решение 7.7.1

```
(defun -nth (w n)
  (if (= n 1) (car w) (-nth (cdr w) (1- n))))

(defun len (w)
  (if (null w) 0 (1+ (len (cdr w)))))

(defun last-n (w n)
  (-nth w (- (len w) (1- n))))
```



```
> (last-n '(5 4 3 2 1) 5)
5
```

Решение 7.7.2

```
(defun len (w)
  (if (null w) 0 (1+ (len (cdr w)))))

(defun last-n (w n)
  (if (= (len w) n) (car w) (last-n (cdr w) n)))
```

```
> (last-n '(5 4 3 2 1) 5)
5
```

Решение 7.7.3

```
(defun last-n (w n) (-nth (reverse w) n))

(defun -nth (w n) (if (= n 1) (car w) (-nth (cdr w) (1- n))))
```

```
> (last-n '(5 4 3 2 1) 5)
5
```

Решение 7.7.4

```
(defun last-n (w n) (elt (reverse w) (1- n)))
```

```
> (last-n '(5 4 3 2 1) 5)
5
```

Решение 7.7.5

```
(defun last-n (w n) (car (last w n)))
```

```
> (last-n '(5 4 3 2 1) 2)
2
```

Решение 7.7.6

```
(defun last-n (w n)
  (if (null (nthcdr n w)) (car w) (last-n (cdr w) n)))
```

```
> (last-n '(5 4 3 2 1) 2)
2
```

Решение 7.7.7

```
(defun last-n (w n)
  (car (last (delete-if #'identity w :count (1- n) :from-end t))))
```

```
> (last-n '(5 4 3 2 1) 2)
2
```

Задача 7.8 last-a.lisp

Определить номер последнего вхождения атома в многоуровневый список.

Решение 7.8.1

```
(defun flat (w &optional acc)
  (cond ((null w) acc)
        ((atom w) (cons w acc))
        ((flat (car w) (flat (cdr w) acc)))))

(defun last-a (w a)
  (when (member a w) (1+ (position a (flat w) :from-end t))))

> (last-a '(a (a b) a (b a) v) 'a)
6
```

Решение 7.8.2

```
(defun flat (w)
  (loop for a in w
        if (listp a) append (flat a)
        else collect a))

(defun last-a (w a)
  (when (member a w) (1+ (position a (flat w) :from-end t))))

> (last-a '(a (a b) a (b a) v) 'a)
6
```

Решение 7.8.3

```
(defun flat (w)
  (mapcan #'(lambda (a) (if (listp a) (flat a) `(,a))) w))

(defun last-a (w a)
  (when (member a w) (1+ (position a (flat w) :from-end t))))

> (last-a '(a (a b) a (b a) v) 'a)
6
```

Решение 7.8.4

```
(defun flat (w)
  (typecase w
    (cons (mapcan #'flat w))
    (t `(,w))))

(defun last-a (w a)
  (when (member a w) (1+ (position a (flat w) :from-end t))))

> (last-a '(a (a b) a (b a) v) 'a)
```

6

Решение 7.8.5

```
(defun flat (w)
  (if (atom w) `(,w) (mapcan #'flat w)))

(defun last-a (w a)
  (when (member a w) (1+ (position a (flat w) :from-end t))))

> (last-a '(a (a b) a (b a) v) 'a)
6
```

Решение 7.8.6

```
(defun last-a (w n &aux (a (car w)) (d (cdr w)))
  (when w (if (listp a)
              (last-a (nconc a d) n)
              (let ((r (last-a d n)))
                (if r (1+ r) (when (equal a n) 1))))))

> (last-a '(a (a b) a (b a) v) 'a)
6
```

Решение 7.8.7

```
(defun last-a (w n &aux (a (car w)) (d (cdr w)))
  (cond ((null w) nil)
        ((atom a) ((lambda (r)
                      (if r (1+ r) (when (equal a n) 1)))
                   (last-a d n)))
        ((last-a (nconc a d) n))))

> (last-a '(a (a b) a (b a) v) 'a)
6
```

Решение 7.8.8

```
(defun last-a (w n &aux (a (car w)) (d (cdr w)))
  (cond ((null w) nil)
        ((atom a) (let ((r (last-a d n)))
                     (if r (1+ r) (when (equal a n) 1))))
        ((last-a (nconc a d) n))))

> (last-a '(a (a b) a (b a) v) 'a)
6
```

Решение 7.8.9

```
(defun last-a (w n &aux (a (car w)) (d (cdr w)))
  (cond ((null w) nil)
        ((listp a) (last-a (nconc a d) n))
        ((let ((r (last-a d n)))
```

```

      (if r (1+ r) (when (equal a n) 1))))))
> (last-a '(a (a b) a (b a) v) 'a)
6

```

Задача 7.9 *count-same.lisp*

Написать функцию, аргументом которой является список атомов, которая выдает значение истина - t, если в списке есть n и более одинаковых атома, и ложь - nil, в противном случае.

Решение 7.9.1

```

(defun count-same (w n &optional (v w))
  (cond ((null w) nil)
        ((>= (count (car w) v) n) t)
        ((count-same (cdr w) n v))))
> (count3 '(a b b b) 3)
T

```

Задача 7.10 *count-elm.lisp*

Найти число вхождений заданного элемента в список (на любом уровне вложенности).

Решение 7.10.1

```

(defun count-elm (w n &aux (a (car w)) (d (cdr w)))
  (cond ((null w) 0)
        ((atom a) (if (eql n a)
                       (1+ (count-elm d n))
                       (count-elm d n)))
        ((+ (count-elm a n) (count-elm d n)))))
> (count-elm '((1 2) 2 (4 2 3)) 2)
3

```

Решение 7.10.2

```

(defun count-elm (n w &aux (a (car w)) (d (cdr w)))
  (cond ((null w) 0)
        ((listp a) (+ (count-elm n a) (count-elm n d)))
        ((eql n a) (1+ (count-elm n d)))
        ((count-elm n d)))))
> (count-elm 2 '((1 2) 2 (4 2 3)))
3

```

Решение 7.10.3 (автор - Lambdik, www.cyberforum.ru)

```
(defun count-elm (n w)
  (cond ((eql n w) 1) ((atom w) 0)
        ((+ (count-elm n (car w)) (count-elm n (cdr w))))))

> (count-elm 2 '((1 2) 2 (4 2 3)))
3
```

Решение 7.10.4

```
(defun count-elm (n w)
  (if (eql n w) 1 (if (atom w) 0 (+ (count-elm n (car w))
                                     (count-elm n (cdr w))))))

> (count-elm 2 '((1 2) 2 (4 2 3)))
3
```

Задача 7.11 add&length.lisp

Добавить элемент в конец списка, если его нет в этом списке. Найти длину полученного списка. Если элемент есть в списке или список пустой, то значение функции nil.

Решение 7.11.1

```
(defun add&length (n w)
  (cond ((or (null w) (member n w)) nil)
        ((length (nconc w (list n))))))

> (add&length 4 '(0 1 2 3))
5
```

Задача 7.12 insider.lisp

Проверить, входит ли элемент в список.

Решение 7.12.1 (для одноуровневого списка)

```
(defun insider (w z)
  (cond ((null w) nil)
        ((eql (car w) z) t)
        ((insider (cdr w) z))))

> (insider '(a b) 'b)
T
```

Решение 7.12.2 (для многоуровневого списка)

```
(defun insider (w z &aux (a (car w)) (d (cdr w)))
  (cond ((null w) nil)
        ((consp a) (or (insider a z) (insider d z)))
        ((eql a z) t)))
```

```
((insider d z)))
```

```
> (insider '((a (b)) c) 'b)
T
```

Задача 7.13 *count-atoms.lisp*

Напишите функцию (count p w), которая подсчитывает, сколько атомов в многоуровневом списке w удовлетворяет предикату p.

Решение 7.13.1

```
(defun ft (w &optional ac)
  (cond ((null w) ac)
        ((atom w) (cons w ac))
        ((ft (car w) (ft (cdr w) ac)))))

(defun fn (w p &optional (n 0))
  (cond ((null w) n)
        ((funcall p (car w)) (fn (cdr w) p (+ n 1)))
        ((fn (cdr w) p n))))

(defun count-atoms (w p)
  (fn (ft w) p))

> (count-atoms '(1 (2 3) 4) #'evenp)
2
```

Задача 7.14 *elm.lisp*

Определить функцию, которая принимает два аргумента – число и список, и возвращает элемент из списка, который соответствует номеру переданного числа.

Решение 7.14.1

```
(defun elm (n w) (cond ((null w) nil)
                       ((= n 1) (car w))
                       ((elm (- n 1) (cdr w)))))

> (elm 2 '(a b c d e))
B
```

Решение 7.14.2

```
(defun elm (n w) (when w (if (= n 1) (car w) (elm (- n 1) (cdr w)))))

> (elm 2 '(a b c d e))
B
```

Решение 7.14.3


```
> (member-test 'd '(c a d))
T
> (member-test 'b '(c a d))
NIL
```

Задача 7.17 pos.lisp

Написать функцию, которая определяет на каком месте стоит заданный элемент в заданном списке.

Решение 7.17.1

```
(defun pos (a w &optional (p 0)) (cond ((null w) nil)
                                         ((eql (car w) a) p)
                                         ((pos a (cdr w) (+ p 1)))))

> (pos 2 '(0 1 2 3 4))
2
```

Решение 7.17.2

```
(defun pos (n w) (when (member n w) (os n w)))

(defun os (n w) (if (eq (car w) n) 0 (1+ (os n (cdr w)))))

> (pos 2 '(0 1 2 3 4))
2
> (pos 2 '(0 1 3 4))
NIL
```

Решение 7.17.3

```
(defun pos (n w)
  (when (member n w)
    (labels ((os (n w)
              (if (eq (car w) n) 0 (1+ (os n (cdr w)))))
      (os n w)))))

> (pos 2 '(0 1 3 4))
NIL
> (pos 2 '(0 1 2 3 4))
2
```

Решение 7.17.4

```
(defun pos (n w) (find n w))

> (pos 2 '(0 1 3 4))
NIL
> (pos 2 '(0 1 2 3 4))
2
```



```

> (check-console-primes "d:/new.txt")
11
12
13
14
quit
NIL

d:/new.txt:
T NIL T NIL

```

Задача 7.21 *check-file-primes.lisp*

Common Lisp is a big-city language. Spit out the hayseed, pronounce "shit" with one syllable and "shotgun" with two. You're not in Kansas, anymore. C is the language of the poor farmer village where the allocation of every seed and livestock matters, where taxes are low and public service non-existent. Appreciating the value of a large language is evidently hard for many people, just like many people find themselves miserable in the big city and go to great lengths to create a small village for themselves in the city where everything is like it used to be where they came from. Common Lisp can accommodate people who want to program in any old language and re-create what they are used to, but if they want to get the most out of it, the only way to do it is to adapt to the language and accept that somebody else may be better than you are at designing languages.

Erik Naggum

Определить функцию, которая проверяет, являются ли элементы списка *seq* простыми числами. Если элемент является простым числом, функция возвращает в результирующем списке значение *T*. Если элемент не является натуральным числом, функция возвращает в результирующем списке *NIL*. Ввод из файла, вывод значения на экран.

Решение 7.21.1

```

(defun check (x)
  (when (> x 1) (loop for i from 2 to (isqrt x)
                     never (zerop (mod x i)))))

(defun check-file-primes (path)
  (mapcar #'check
    (read-from-string

```

```

(concatenate
 'string "("
 (with-open-file (s path :direction :input)
  (read-line s nil :eof))
  ")"))))

> (check-fie-primes "d:/old.txt")
(T NIL T NIL)

d:/old.txt:
11 12 13 14

```

Задача 7.22 *exceed-number.lisp*

Создать программу, вычисляющую количество элементов больших *n* из введенного списка.

Решение 7.22.1

```

(defun exceed-number (n w)
  (cond ((null w) 0)
        ((> (car w) n) (+ 1 (exceed-number n (cdr w))))
        ((exceed-number n (cdr w)))))

> (exceed-number 5 '(0 5 6 7 8))
3

```

Задача 7.23 *dive-count.lisp*

Написать программу определения количества атомов на заданном уровне в произвольном исходном списке. Атомы, содержащиеся в исходном списке, имеют уровень 0, атомы, содержащиеся в следующем вложенном списке, имеют уровень 1 и т. д.

Решение 7.23.1

```

(defun dive-elms (n w &aux (a (car w)) (d (cdr w)))
  (cond ((null w) nil)
        ((= n 0) w)
        ((atom a) (dive-elms n d))
        ((nconc (dive-elms (1- n) a) (dive-elms n d)))))

(defun dive-count (n w)
  (count-if #'atom (dive-elms n w)))

> (dive-count 0 '(a (b (c 1) d e) f ((2 (g)) 3)))
2
> (dive-count 1 '(a (b (c 1) d e) f ((2 (g)) 3)))
4
> (dive-count 2 '(a (b (c 1) d e) f ((2 (g)) 3)))
3

```

Решение 7.23.2

```
(defun dive-count (n w &aux (a (car w)) (d (cdr w)))
  (cond ((null w) 0)
        ((= n 0) (count-if #'atom w))
        ((atom a) (dive-count n d))
        ((+ (dive-count (1- n) a) (dive-count n d)))))

> (dive-count 0 '(a (b (c 1) d e) f ((2 (g)) 3)))
2
> (dive-count 1 '(a (b (c 1) d e) f ((2 (g)) 3)))
4
> (dive-count 2 '(a (b (c 1) d e) f ((2 (g)) 3)))
3
```

Задача 7.24 any.lisp

Определить функционал, возвращающий истину, если определенным свойством обладает хотя бы один элемент списка.

Решение 7.24.1

```
(defun any (p w) (cond ((null w) nil)
                        ((funcall p (car w)) t)
                        ((any p (cdr w)))))

> (any #'minusp '(-1 1))
T
> (any #'minusp '(1 1))
NIL
```

Решение 7.24.2

```
(defun any (p w) (when w (or (funcall p (car w)) (any p (cdr w)))))

> (any #'minusp '(-1 1))
T
> (any #'minusp '(1 1))
NIL
```

Решение 7.24.3

```
(defun any (p w) (loop for a in w thereis (funcall p a)))

> (any #'minusp '(-1 1))
T
> (any #'minusp '(1 1))
NIL
```

Решение 7.24.4

```
(defun any (p w) (find-if p w))

> (any #'minusp '(1 1))
NIL
> (any #'minusp '(-1 1))
-1
```

Решение 7.24.5

```
(defun any (p w) (some p w))

> (any #'minusp '(-1 1))
T
> (any #'minusp '(1 1))
NIL
```

Задача 7.25 count-after.lisp

Определить функцию, возвращающую количество элементов списка, следующих после заданного символа.

Решение 7.25.1

```
(defun count-after (m w &optional f)
  (cond ((null w) 0)
        ((eq (car w) m) (count-after m (cdr w) t))
        (f (1+ (count-after m (cdr w) t)))
        ((count-after m (cdr w)))))

> (count-after '* '(a b c * 1 2 3 4))
4
> (count-after '+ '(a b c * 1 2 3 4))
0
```

Решение 7.25.2

```
(defun count-after (m w &aux (n (position m w)))
  (if n (length (nthcdr (1+ (position m w)) w)) 0))

> (count-after '* '(a b c * 1 2 3 4))
4
> (count-after '+ '(a b c * 1 2 3 4))
0
```

Решение 7.25.3

```
(defun count-after (m w &aux (n (position m w)))
  (if n (count-if #'identity w :start (1+ n)) 0))

> (count-after '* '(a b c * 1 2 3 4))
4
> (count-after '+ '(a b c * 1 2 3 4))
```

0

Решение 7.25.4

```
(defun count-after (m w &aux (v (member m w)))
  (if v (1- (length v)) 0))

> (count-after '* '(a b c * 1 2 3 4))
4
> (count-after '+ '(a b c * 1 2 3 4))
0
```

Задача 7.26 rhymes.lisp

Определить функцию, удаляющую каждый n-й элемента в списке, пока не останется один элемент (считалка).

Решение 7.26.1

```
(defun rhymes (w n &optional acc)
  (if (nth n w)
      (rhymes (nthcdr n w) n (nconc acc (subseq w 0 (1- n))))
      (drop-thin (nconc w acc) n)))

(defun drop-thin (w n)
  (if (cdr w)
      (drop-thin (drop-n w (rem n (length w))) n)
      (car w)))

(defun drop-n (w n)
  (print w)
  (if (> n 0)
      (nconc (nthcdr n w) (subseq w 0 (1- n)))
      (butlast w)))

> (rhymes '(a b c d e f g h) 5)
(F G H A B C D)
(C D F G H A)
(A C D F G)
(A C D F)
(C D F)
(F C)
C
> (rhymes '(a b c d e f g h i j k) 7)
(H I J K A B C D E F)
(D E F H I J K A B)
(A B D E F H I J)
(J A B D E F H)
(J A B D E F)
(A B D E F)
(D E F A)
(A D E)
```

```
(D E)
E
```

Решение 7.26.2

```
(defun rhymes (w n &optional acc)
  (if (nth n w)
      (rhymes (nthcdr n w) n (nconc acc (subseq w 0 (1- n))))
      (drop-thin (nconc w acc) n)))

(defun drop-thin (w n &aux (m (rem n (length w))))
  (print w)
  (if (cdr w)
      (drop-thin (if (> m 0)
                     (nconc (nthcdr m w) (subseq w 0 (1- m)))
                     (butlast w))
                 n)
      (car w)))
```

```
> (rhymes '(a b c d e f g h) 5)
(F G H A B C D)
(C D F G H A)
(A C D F G)
(A C D F)
(C D F)
(F C)
(C)
C
> (rhymes '(a b c d e f g h i j k) 7)
(H I J K A B C D E F)
(D E F H I J K A B)
(A B D E F H I J)
(J A B D E F H)
(J A B D E F)
(A B D E F)
(D E F A)
(A D E)
(D E)
(E)
E
```

Решение 7.26.3

```
(defun rhymes (w n &optional acc)
  (if (nth n w)
      (rhymes (subseq w n) n (nconc acc (subseq w 0 (1- n))))
      (drop-thin (nconc w acc) n)))

(defun drop-thin (w n &aux (m (rem n (length w))))
  (print w)
  (if (cdr w)
      (drop-thin (if (> m 0)
                     (nconc (subseq w m) (subseq w 0 (1- m)))
```

```

                                (butlast w))
                                n)
    (car w)))

> (rhymes '(a b c d e f g h) 5)
(F G H A B C D)
(C D F G H A)
(A C D F G)
(A C D F)
(C D F)
(F C)
(C)
C
> (rhymes '(a b c d e f g h i j k) 7)
(H I J K A B C D E F)
(D E F H I J K A B)
(A B D E F H I J)
(J A B D E F H)
(J A B D E F)
(A B D E F)
(D E F A)
(A D E)
(D E)
(E)
E

```

Решение 7.26.4

```

(defun rhymes (w n &optional acc)
  (if (nth n w)
      (rhymes (subseq w n) n (nconc acc (subseq w 0 (1- n))))
      (drop (nconc w acc) n)))

(defun drop (w n &aux (m (rem n (length w))))
  (if (cdr w)
      (drop (if (> m 0)
                 (nconc (subseq w m) (subseq w 0 (1- m)))
                 (butlast w))
            n)
      w))

> (rhymes '(a b c d e f g h i j k) 7)
(E)
> (rhymes '(a b c d e f g h) 5)
(C)

```

Решение 7.26.5

```

(defun rhymes (w n &optional acc)
  (if (nth n w)
      (rhymes (subseq w n) n (nconc acc (subseq w 0 (1- n))))
      (drop (nconc w acc) n)))

```



```

(defun drop (w n &aux (m (rem n (length w))))
  (if (cdr w)
      (drop (if (> m 0)
                (nconc (subseq w m) (subseq w 0 (1- m)))
                (butlast w))
          n)
      (car w)))

> (rhymes '(a b c d e f g h) 5)
C
> (rhymes '(a b c d e f g h i j k) 7)
E

```

Задача 7.27 *_some.lisp*

Реализовать функционал, который проверяет, удовлетворяет ли хотя бы один элемент списка заданному предикату.

Решение 7.27.2

```

(defun _some (p w)
  (loop for a in w when (funcall p a) return t))

> (_some #'numberp '(a 1))
T

```

Задача 7.28 *n-elm-n.lisp*

Определить функцию, которая выбирает на верхнем уровне списка n -ый элемент, отсчитанный от начала списка, если $n > 0$, или от конца списка, если $n < 0$.

Решение 7.28.1

```

(defun n-elm-n (w n)
  (nth (1- (abs n)) (if (plusp n) w (reverse w))))

> (n-elm-n '(1 2 3 4 5) 1)
1
> (n-elm-n '(1 2 3 4 5) -1)
5

```

Задача 7.29 *previous-z.lisp*

Определить функцию, возвращающую атом списка, предшествующий заданному.

Решение 7.29.1

```

(defun previous-z (z w)
  (cond ((null w) nil)

```

```
((eq (cadr w) z) (car w))
((previous-z z (cdr w)))))
```

```
> (previous-z 1 '(1 2 3))
NIL
> (previous-z 2 '(1 2 3))
1
> (previous-z 3 '(1 2 3))
2
> (previous-z 4 '(1 2 3))
NIL
```

Решение 7.29.2

```
(defun previous-z (z w)
  (when w (if (eq (cadr w) z)
              (car w)
              (previous-z z (cdr w)))))
```

```
> (previous-z 1 '(1 2 3))
NIL
> (previous-z 2 '(1 2 3))
1
> (previous-z 3 '(1 2 3))
2
> (previous-z 4 '(1 2 3))
NIL
```

Решение 7.29.3 (автор - transformator.t, www.cyberforum.ru)

```
(defun previous-z (z w)
  (loop for (a b) on w while b
        if (eq z b) do (return a)))
```

```
> (previous-z 1 '(1 2 3))
NIL
> (previous-z 2 '(1 2 3))
1
> (previous-z 3 '(1 2 3))
2
> (previous-z 4 '(1 2 3))
NIL
```

Решение 7.29.4

```
(defun previous-z (z w &aux (p (position z w)))
  (when (and p (plusp p)) (nth (1- p) w)))
```

```
> (previous-z 1 '(1 2 3))
NIL
> (previous-z 2 '(1 2 3))
1
> (previous-z 3 '(1 2 3))
```

```
2
> (previous-z 4 '(1 2 3))
NIL
```

Задача 7.30 *max-depth-of-atom.lisp*

Определить функцию, возвращающую максимальную глубину вхождения атома в список.

Решение 7.30.1

```
(defun max-depth-of-atom (w z &optional (c 0) (m -1)
                          &aux (a (car w)) (d (cdr w)))
  (cond ((null w) m)
        ((eq z a) (max-depth-of-atom d z c (if (> c m) c m)))
        ((listp a) (max (max-depth-of-atom a z (1+ c) m)
                         (max-depth-of-atom d z c m)))
        ((max-depth-of-atom d z c m))))

> (max-depth-of-atom '(1 2 3 (4 5 6 7) (((7)))))) 1)
0
> (max-depth-of-atom '(1 2 3 (4 5 6 7) (((7)))))) 4)
1
> (max-depth-of-atom '(1 2 3 (4 5 6 7) (((7)))))) 7)
4
> (max-depth-of-atom '(1 2 3 (4 5 6 7) (((7)))))) 8)
-1
```

Решение 7.30.2

```
(defun max-depth-of-atom (w z &optional (c 0) (m -1)
                          &aux (a (car w)) (d (cdr w)))
  (cond ((null w) m)
        ((eq z a) (max-depth-of-atom d z c (if (> c m) c m)))
        ((atom a) (max-depth-of-atom d z c m))
        ((max (max-depth-of-atom a z (1+ c) m)
               (max-depth-of-atom d z c m)))))

> (max-depth-of-atom '(1 2 3 (4 5 6 7) (((7)))))) 1)
0
> (max-depth-of-atom '(1 2 3 (4 5 6 7) (((7)))))) 4)
1
> (max-depth-of-atom '(1 2 3 (4 5 6 7) (((7)))))) 7)
4
> (max-depth-of-atom '(1 2 3 (4 5 6 7) (((7)))))) 8)
-1
```

Задача 7.31 *elm-num.lisp*

Определить функцию $f(a\ w)$, где a — s -выражение, а w — список, которая выдает номер первого из элементов списка w , совпадающих с a , и nil , если такой элемент не найден.

Решение 7.31.1

```
(defun elm-num (a w &optional (n 0))
  (cond ((null w) nil)
        ((equalp (car w) a) n)
        (t (elm-num a (cdr w) (1+ n)))))
```

```
> (elm-num 'c '(a b c d))
2
```

Решение 7.31.2

```
(defun elm-num (a w &optional (n 0))
  (when w (if (equalp (car w) a) n (elm-num a (cdr w) (1+ n)))))
```

```
> (elm-num 'c '(a b c d))
2
```

Решение 7.31.3

```
(defun elm-num (a w)
  (loop for e in w for i from 0
        when (equalp e a) return i))
```

```
> (elm-num 'c '(a b c d))
2
```

Решение 7.31.4

```
(defun elm-num (a w) (position a w))
```

```
> (elm-num 'c '(a b c d))
2
```

Задача 7.32 low.lisp

Определить функцию $f(a\ w)$, где a – число, а w – список чисел, которая выдает количество чисел списка w , не превосходящих a .

Решение 7.32.1

```
(defun low (a w)
  (cond ((null w) 0)
        ((<= (car w) a) (1+ (low a (cdr w)))))
  (t (low a (cdr w)))))
```

```
> (low 4 '(1 2 3 4 5 6 7))
4
```

Решение 7.32.2

```
(defun low (a w) (loop for e in w when (<= e a) count e))

> (low 4 '(1 2 3 4 5 6 7))
4
```

Решение 7.32.3

```
(defun low (a w) (count-if #'(lambda (e) (<= e a)) w))

> (low 4 '(1 2 3 4 5 6 7))
4
```

Задача 7.33 *n-prime.lisp*

By listing the first six prime numbers: 2, 3, 5, 7, 11, and 13, we can see that the 6th prime is 13. What is the 10 001st prime number?

Выписав первые шесть простых чисел, получим 2, 3, 5, 7, 11 и 13. Очевидно, что 6-ое простое число - 13. Какое число является 10001-ым простым числом?

Решение 7.33.1

```
(defun n-prime (n &optional (c 1) (m 3) (ac '(2)))
  (cond ((= c n) (~ m 2))
        ((loop for a in ac never (zerop (rem m a)))
         (n-prime n (1+ c) (+ m 2) (cons m ac)))
        (t (n-prime n c (+ m 2) ac))))

> (time (n-prime 10001))
Real time: 87.91196 sec.
Run time: 87.641365 sec.
Space: 80000 Bytes
104743
```

Решение 7.33.2

```
(defun primep (n)
  (loop for a from 2 to (isqrt n) never (zerop (mod n a))))

(defun n-prime (n &aux (c 1) (m 2))
  (loop while (< c n) do (when (primep (incf m)) (incf c))
    finally (return m)))

> (time (n-prime 10001))
Real time: 1.5756028 sec.
Run time: 1.57561 sec.
Space: 0 Bytes
104743
```

Задача 7.34 large-sum.lisp

Work out the first ten digits of the sum of the following one-hundred 50-digit numbers.

Найдите первые десять цифр суммы следующих ста 50-значных чисел.

```

37107287533902102798797998220837590246510135740250
46376937677490009712648124896970078050417018260538
74324986199524741059474233309513058123726617309629
91942213363574161572522430563301811072406154908250
23067588207539346171171980310421047513778063246676
89261670696623633820136378418383684178734361726757
28112879812849979408065481931592621691275889832738
44274228917432520321923589422876796487670272189318
47451445736001306439091167216856844588711603153276
70386486105843025439939619828917593665686757934951
62176457141856560629502157223196586755079324193331
64906352462741904929101432445813822663347944758178
92575867718337217661963751590579239728245598838407
58203565325359399008402633568948830189458628227828
80181199384826282014278194139940567587151170094390
35398664372827112653829987240784473053190104293586
86515506006295864861532075273371959191420517255829
71693888707715466499115593487603532921714970056938
54370070576826684624621495650076471787294438377604
53282654108756828443191190634694037855217779295145
36123272525000296071075082563815656710885258350721
4587657617241097644733911060721826523687722363045
17423706905851860660448207621209813287860733969412
81142660418086830619328460811191061556940512689692
51934325451728388641918047049293215058642563049483
62467221648435076201727918039944693004732956340691
15732444386908125794514089057706229429197107928209
55037687525678773091862540744969844508330393682126
18336384825330154686196124348767681297534375946515
80386287592878490201521685554828717201219257766954
78182833757993103614740356856449095527097864797581
16726320100436897842553539920931837441497806860984
48403098129077791799088218795327364475675590848030
87086987551392711854517078544161852424320693150332
59959406895756536782107074926966537676326235447210
69793950679652694742597709739166693763042633987085
41052684708299085211399427365734116182760315001271
65378607361501080857009149939512557028198746004375
35829035317434717326932123578154982629742552737307
94953759765105305946966067683156574377167401875275
88902802571733229619176668713819931811048770190271
25267680276078003013678680992525463401061632866526
36270218540497705585629946580636237993140746255962
24074486908231174977792365466257246923322810917141
91430288197103288597806669760892938638285025333403

```

34413065578016127815921815005561868836468420090470
23053081172816430487623791969842487255036638784583
11487696932154902810424020138335124462181441773470
63783299490636259666498587618221225225512486764533
67720186971698544312419572409913959008952310058822
95548255300263520781532296796249481641953868218774
7608537213285723110424803456124867697064507995236
37774242535411291684276865538926205024910326572967
23701913275725675285653248258265463092207058596522
29798860272258331913126375147341994889534765745501
18495701454879288984856827726077713721403798879715
38298203783031473527721580348144513491373226651381
34829543829199918180278916522431027392251122869539
40957953066405232632538044100059654939159879593635
29746152185502371307642255121183693803580388584903
41698116222072977186158236678424689157993532961922
62467957194401269043877107275048102390895523597457
23189706772547915061505504953922979530901129967519
8618808822587531452958409925120382900940770775672
11306739708304724483816533873502340845647058077308
82959174767140363198008187129011875491310547126581
97623331044818386269515456334926366572897563400500
42846280183517070527831839425882145521227251250327
55121603546981200581762165212827652751691296897789
32238195734329339946437501907836945765883352399886
75506164965184775180738168837861091527357929701337
62177842752192623401942399639168044983993173312731
32924185707147349566916674687634660915035914677504
99518671430235219628894890102423325116913619626622
73267460800591547471830798392868535206946944540724
76841822524674417161514036427982273348055556214818
97142617910342598647204516893989422179826088076852
87783646182799346313767754307809363333018982642090
10848802521674670883215120185883543223812876952786
71329612474782464538636993009049310363619763878039
62184073572399794223406235393808339651327408011116
66627891981488087797941876876144230030984490851411
60661826293682836764744779239180335110989069790714
85786944089552990653640447425576083659976645795096
66024396409905389607120198219976047599490197230297
64913982680032973156037120041377903785566085089252
16730939319872750275468906903707539413042652315011
94809377245048795150954100921645863754710598436791
78639167021187492431995700641917969777599028300699
15368713711936614952811305876380278410754449733078
40789923115535562561142322423255033685442488917353
44889911501440648020369068063960672322193204149535
41503128880339536053299340368006977710650566631954
81234880673210146739058568557934581403627822703280
82616570773948327592232845941706525094512325230608
22918802058777319719839450180888072429661980811197
77158542502016545090413245809786882778948721859617
72107838435069186155435662884062257473692284509516

20849603980134001723930671666823555245252804609722
 53503534226472524250874054075591789781264330331690

Решение 7.34.1

```
(defun large-sum (n w)
  (parse-integer
   (subseq
    (write-to-string
     (reduce #' + w) 0 n)))

> (large-sum 10 '(
37107287533902102798797998220837590246510135740250
46376937677490009712648124896970078050417018260538
74324986199524741059474233309513058123726617309629
91942213363574161572522430563301811072406154908250
23067588207539346171171980310421047513778063246676
89261670696623633820136378418383684178734361726757
28112879812849979408065481931592621691275889832738
44274228917432520321923589422876796487670272189318
47451445736001306439091167216856844588711603153276
70386486105843025439939619828917593665686757934951
62176457141856560629502157223196586755079324193331
64906352462741904929101432445813822663347944758178
92575867718337217661963751590579239728245598838407
58203565325359399008402633568948830189458628227828
80181199384826282014278194139940567587151170094390
35398664372827112653829987240784473053190104293586
86515506006295864861532075273371959191420517255829
71693888707715466499115593487603532921714970056938
54370070576826684624621495650076471787294438377604
53282654108756828443191190634694037855217779295145
36123272525000296071075082563815656710885258350721
45876576172410976447339110607218265236877223636045
17423706905851860660448207621209813287860733969412
81142660418086830619328460811191061556940512689692
51934325451728388641918047049293215058642563049483
62467221648435076201727918039944693004732956340691
15732444386908125794514089057706229429197107928209
55037687525678773091862540744969844508330393682126
18336384825330154686196124348767681297534375946515
80386287592878490201521685554828717201219257766954
78182833757993103614740356856449095527097864797581
16726320100436897842553539920931837441497806860984
48403098129077791799088218795327364475675590848030
87086987551392711854517078544161852424320693150332
59959406895756536782107074926966537676326235447210
69793950679652694742597709739166693763042633987085
41052684708299085211399427365734116182760315001271
65378607361501080857009149939512557028198746004375
35829035317434717326932123578154982629742552737307
94953759765105305946966067683156574377167401875275
88902802571733229619176668713819931811048770190271
```


25267680276078003013678680992525463401061632866526
36270218540497705585629946580636237993140746255962
24074486908231174977792365466257246923322810917141
91430288197103288597806669760892938638285025333403
34413065578016127815921815005561868836468420090470
230530811728164304876237919698424872550366384583
11487696932154902810424020138335124462181441773470
63783299490636259666498587618221225225512486764533
67720186971698544312419572409913959008952310058822
95548255300263520781532296796249481641953868218774
76085327132285723110424803456124867697064507995236
37774242535411291684276865538926205024910326572967
23701913275725675285653248258265463092207058596522
29798860272258331913126375147341994889534765745501
18495701454879288984856827726077713721403798879715
38298203783031473527721580348144513491373226651381
34829543829199918180278916522431027392251122869539
40957953066405232632538044100059654939159879593635
29746152185502371307642255121183693803580388584903
41698116222072977186158236678424689157993532961922
62467957194401269043877107275048102390895523597457
23189706772547915061505504953922979530901129967519
86188088225875314529584099251203829009407770775672
11306739708304724483816533873502340845647058077308
82959174767140363198008187129011875491310547126581
97623331044818386269515456334926366572897563400500
42846280183517070527831839425882145521227251250327
55121603546981200581762165212827652751691296897789
32238195734329339946437501907836945765883352399886
75506164965184775180738168837861091527357929701337
62177842752192623401942399639168044983993173312731
32924185707147349566916674687634660915035914677504
99518671430235219628894890102423325116913619626622
73267460800591547471830798392868535206946944540724
76841822524674417161514036427982273348055556214818
97142617910342598647204516893989422179826088076852
87783646182799346313767754307809363333018982642090
10848802521674670883215120185883543223812876952786
71329612474782464538636993009049310363619763878039
62184073572399794223406235393808339651327408011116
66627891981488087797941876876144230030984490851411
60661826293682836764744779239180335110989069790714
85786944089552990653640447425576083659976645795096
66024396409905389607120198219976047599490197230297
64913982680032973156037120041377903785566085089252
16730939319872750275468906903707539413042652315011
94809377245048795150954100921645863754710598436791
78639167021187492431995700641917969777599028300699
15368713711936614952811305876380278410754449733078
40789923115535562561142322423255033685442488917353
44889911501440648020369068063960672322193204149535
41503128880339536053299340368006977710650566631954
81234880673210146739058568557934581403627822703280

```

82616570773948327592232845941706525094512325230608
22918802058777319719839450180888072429661980811197
77158542502016545090413245809786882778948721859617
72107838435069186155435662884062257473692284509516
20849603980134001723930671666823555245252804609722
53503534226472524250874054075591789781264330331690) )
5537376230
10

```

Задача 7.35 *_every.lisp*

Определите функциональный предикат, который истинен в том и только том случае, когда, являющийся функциональным аргументом предикат истинен для всех элементов списка.

Решение 7.35.1

```

(defun _every (p w)
  (cond ((null w) t)
        ((funcall p (car w)) (_every p (cdr w)))))

> (_every #'evenp '(2 4 6 8))
T
> (_every #'evenp '(1 4 6 8))
NIL

```

Решение 7.35.2

```

(defun _every (p w)
  (cond ((null w) t)
        ((funcall p (car w)) (_every p (cdr w)))))

> (_every #'evenp '(2 4 6 8))
T
> (_every #'evenp '(1 4 6 8))
NIL

```

Решение 7.35.3

```

(defun _every (p w)
  (if w (when (funcall p (car w)) (_every p (cdr w))) t))

> (_every #'evenp '(2 4 6 8))
T
> (_every #'evenp '(1 4 6 8))
NIL

```

Решение 7.35.4

```

(defun _every (p w)
  (if w (and (funcall p (car w)) (_every p (cdr w))) t))

```

```
> (_every #'evenp '(2 4 6 8))
T
> (_every #'evenp '(1 4 6 8))
NIL
```

Решение 7.35.5

```
(defun _every (p w)
  (or (null w) (and (funcall p (car w)) (_every p (cdr w)))))

> (_every #'evenp '(2 4 6 8))
T
> (_every #'evenp '(1 4 6 8))
NIL
```

Решение 7.35.6

```
(defun _every (p w) (eval (cons 'and (mapcar p w))))

> (_every #'evenp '(2 4 6 8))
T
> (_every #'evenp '(1 4 6 8))
NIL
```

Решение 7.35.7

```
(defun _every (p w) (loop for a in w always (funcall p a)))

> (_every #'evenp '(2 4 6 8))
T
> (_every #'evenp '(1 4 6 8))
NIL
```

Задача 7.36 *dive-product.lisp*

Определить функцию, которая вычисляет произведение числовых элементов списка *w*, находящихся на *n*-ном уровне списка

Решение 7.36.1

```
(defun dive-elms (n w &aux (a (car w)) (d (cdr w)))
  (cond ((null w) nil)
        ((= n 0) w)
        ((atom a) (dive-elms n d))
        ((nconc (dive-elms (1- n) a) (dive-elms n d)))))

(defun dive-product (n w)
  (reduce #'* (remove-if-not #'numberp (dive-elms n w))))

> (dive-elms 2 '(a (b (c 1) d e) f ((2 (g)) 3)))
(C 1 2 (G))
> (dive-product 1 '(a (b (c 1) d e) f ((2 (g)) 3)))
```

```
3
> (dive-product 2 '(a (b (c 1) d e) f ((2 (g)) 3)))
2
```

Решение 7.36.2

```
(defun dive-product (n w &aux (a (car w)) (d (cdr w)))
  (cond ((null w) 1)
        ((= n 0) (reduce #'* (remove-if-not #'numberp w)))
        ((atom a) (dive-product n d))
        ((* (dive-product (1- n) a) (dive-product n d)))))

> (dive-elms 2 '(a (b (c 1) d e) f ((2 (g)) 3)))
(C 1 2 (G))
> (dive-product 1 '(a (b (c 1) d e) f ((2 (g)) 3)))
3
> (dive-product 2 '(a (b (c 1) d e) f ((2 (g)) 3)))
2
```

Задача 7.37 bin-decim.lisp

Определить функцию, которая возвращает число в десятичной системе счисления, образованное первыми m элементами списка w , состоящего из нулей и единиц.

Решение 7.37.1

```
(defun bin-decim (m w &optional (v (subseq w 0 m)) (n 0))
  (if (null v) n (bin-decim m w (cdr v) (+ (car v) n n))))

> (bin-decim 3 '(1 0 0 0 1))
4
> (bin-decim 4 '(1 0 0 0 1))
8
> (bin-decim 5 '(1 0 0 0 1))
17
```

Решение 7.37.2

```
(defun bin-deci (m w &optional (n 0))
  (if (null w) n (bin-deci m (cdr w) (+ (car w) n n))))

(defun bin-decim (m w) (bin-deci m (subseq w 0 m)))

> (bin-decim 3 '(1 0 0 0 1))
4
> (bin-decim 4 '(1 0 0 0 1))
8
> (bin-decim 5 '(1 0 0 0 1))
17
```

Решение 7.37.3

```

(defun bin-deci (m w &optional (n 0))
  (if w (bin-deci m (cdr w) (+ (car w) n n)) n))

(defun bin-decim (m w) (bin-deci m (subseq w 0 m)))

> (bin-decim 3 '(1 0 0 0 1))
4
> (bin-decim 4 '(1 0 0 0 1))
8
> (bin-decim 5 '(1 0 0 0 1))
17

```

Задача 7.38 *count-exept.lisp*

During the years we worked on Viaweb I read a lot of job descriptions. A new competitor seemed to emerge out of the woodwork every month or so. The first thing I would do, after checking to see if they had a live online demo, was look at their job listings. After a couple years of this I could tell which companies to worry about and which not to. The more of an IT flavor the job descriptions had, the less dangerous the company was. The safest kind were the ones that wanted Oracle experience. You never had to worry about those. You were also safe if they said they wanted C++ or Java developers. If they wanted Perl or Python programmers, that would be a bit frightening - that's starting to sound like a company where the technical side, at least, is run by real hackers. If I had ever seen a job posting looking for Lisp hackers, I would have been really worried.

Paul Graham «Beating the Averages»

Даны атом и список. Определить функцию, вычисляющую, сколько всего атомов в списке исключая заданный атом.

Решение 7.38.1

```

(defun count-exept (a w &optional (n 0))
  (cond ((null w) n)
        ((atom w) (if (not (eq w a)) (1+ n) n))
        ((count-exept a (car w) (count-exept a (cdr w) n)))))

> (count-exept 2 '(1 2 (3 (2 4)) (5 2)))
4

```

Решение 7.38.2

```
(defun count-exept (a w &optional (n 0))
  (cond ((null w) n)
        ((atom w) (if (eq w a) n (1+ n)))
        ((count-exept a (car w) (count-exept a (cdr w) n)))))

> (count-exept 2 '(1 2 (3 (2 4)) (5 2)))
4
```

Решение 7.38.3

```
(defun count-exept (a w)
  (cond ((null w) 0)
        ((eq a (car w)) (count-exept a (cdr w)))
        ((listp (car w)) (+ (count-exept a (car w))
                             (count-exept a (cdr w))))
        ((+ 1 (count-exept a (cdr w))))))

> (count-exept 2 '(1 2 (3 (2 4)) (5 2)))
4
```

Решение 7.38.4

```
(defun count-exept (a w)
  (cond ((null w) 0)
        (t (+ (cond ((eq a (car w)) 0)
                     ((listp (car w)) (count-exept a (car w)))
                     (t 1))
               (count-exept a (cdr w))))))

> (count-exept 2 '(1 2 (3 (2 4)) (5 2)))
4
```

Решение 7.38.5

```
(defun count-exept (a w)
  (if w
      (+ (cond ((eq a (car w)) 0)
              ((listp (car w)) (count-exept a (car w)))
              (t 1))
         (count-exept a (cdr w)))
      0))

> (count-exept 2 '(1 2 (3 (2 4)) (5 2)))
4
```

Решение 7.38.6

```
(defun count-exept (a w)
  (if w
      (+ (cond ((eq a (car w)) 0)
              ((listp (car w)) (count-exept a (car w)))
```

```

        (1))
      (count-exept a (cdr w)))
    0))

> (count-exept 2 '(1 2 (3 (2 4)) (5 2)))
4

```

Решение 7.38.7

```

(defun count-exept (a w)
  (if w
      (+ (cond ((eq a (car w)) 0)
               ((atom (car w)) 1)
               ((count-exept a (car w))))
        (count-exept a (cdr w)))
    0))

> (count-exept 2 '(1 2 (3 (2 4)) (5 2)))
4

```

Решение 7.38.8

```

(defun flat (w &optional acc)
  (cond ((null w) acc)
        ((atom w) (cons w acc))
        ((flat (car w) (flat (cdr w) acc)))))

(defun count-exept (a w)
  (count-if-not #'(lambda (e) (eq e a)) (flat w)))

> (count-exept 2 '(1 2 (3 (2 4)) (5 2)))
4

```

Решение 7.38.9

```

(defun flat (w)
  (loop for a in w
        if (listp a) append (flat a)
        else collect a))

(defun count-exept (a w)
  (loop for e in (flat w) unless (eq e a) count e))

> (count-exept 2 '(1 2 (3 (2 4)) (5 2)))
4

```

Задача 7.39 count-bigger-than-this.lisp

Определить функцию, которая считает количество элементов в списке, больших *n*.

Решение 7.39.1

```
(defun count-bigger-than-this (w n)
  (cond ((null w) 0)
        ((> (car w) n) (1+ (count-bigger-than-this (cdr w) n)))
        ((count-bigger-than-this (cdr w) n))))
```

```
> (count-bigger-than-this '(1 2 3 1 2 3 4) 2)
3
```

Решение 7.39.2

```
(defun count-bigger-than-this (w n) (loop for a in w count (> a n)))
```

```
> (count-bigger-than-this '(1 2 3 1 2 3 4) 2)
3
```

Решение 7.39.3

```
(defun count-bigger-than-this (w n)
  (count-if #'(lambda (a) (> a n)) w))
```

```
> (count-bigger-than-this '(1 2 3 1 2 3 4) 2)
3
```

Задача 7.40 counter.lisp

Определить функцию, которая будет считать сколько раз заданный элемент встречается в списке.

Решение 7.40.1

```
(defun counter (a w)
  (cond ((null w) 0)
        ((equal (car w) a) (1+ (counter a (cdr w))))
        ((counter a (cdr w)))))
```

```
> (counter 'a '(a b c a b a))
3
```

Решение 7.40.2

```
(defun counter (a w)
  (loop for e in w count (equal e a)))
```

```
> (counter 'a '(a b c a b a))
3
```

Решение 7.40.3

```
(defun counter (a w) (count a w))
```

```
> (counter 'a '(a b c a b a))
```


3

Задача 7.41 *dive-count-list.lisp*

Определить функцию, которая возвращает количество подсписков, находящихся на указанном уровне для заданного списка.

Решение 7.41.1

```
(defun dive-count-list (n w &aux (a (car w)) (d (cdr w)))
  (cond ((null w) 0)
        ((= n 0) (count-if #'listp w))
        ((atom a) (dive-count-list n d))
        ((+ (dive-count-list (1- n) a) (dive-count-list n d)))))

> (dive-count-list 0 '(a (b (c 1) d e) f ((2 (g)) 3)))
2
```

Решение 7.41.2

```
(defun dive-count-list (n w &aux (a (car w)))
  (cond ((null w) 0)
        ((= n 0) (count-if #'listp w))
        ((+ (if (atom a) 0 (dive-count-list (1- n) a))
              (dive-count-list n (cdr w))))))

> (dive-count-list 0 '(a (b (c 1) d e) f ((2 (g)) 3)))
2
```

Задача 7.42 *_some.lisp*

Определить функцию, которая проверяет, удовлетворяет ли хотя бы один элемент списка заданному предикату.

Решение 7.42.1

```
(defun _some (p w)
  (loop for a in w when (funcall p a) return t))

> (_some #'numberp '(a 1))
Т
```

Решение 7.42.2

```
(defun _some (p w)
  (loop for a in w thereis (funcall p a)))

> (_some #'numberp '(a 1))
Т
```

Решение 7.42.3

```
(defun _some (p w)
  (when w (or (funcall p (car w)) (_some p (cdr w)))))

> (_some #'numberp '(a 1))
T
```

Задача 7.43 *count-number.lisp*

Lisp is the first programming language to be designed around a non Turing model of computation. Instead, it was designed around the lambda calculus model. The Turing model is the idea of a machine built from an infinite tape, where each strip of the tape is data, and the machine transitions from one strip to another based on the data on the strip. The transitions over the tape follow a very imperative style, where basically all instructions is a kind of goto to somewhere else on the tape. Lisp being from Lambda Calculus instead takes a very different model, it says that with a function, in the mathematical sense, and a few properties, like allowing the function to call itself, you can compute the same things that you can with the Turing model. This leads to more of a functional style of programming. Apart from that massive difference, Lisp was first to invent Dynamicism, where you don't specify types when you code, it was first to introduce the idea of interpreted languages, where the compilation happens at runtime, so you can write macros, or program altering functions which alter the program at runtime. It was first to introduce homoi-conicity, a syntax that mimics the AST of a language, which means it was first to be composed of only expressions, no statements. It was also the first to introduce the concept of a repl, where you can interactively write code at a command prompt. All these things make Lisp special.

Didier Adot

Даны число и s-выражение, состоящее из чисел. Определить функцию, возвращающую количество элементов на всех уровнях этого s-выражения, которые равны данному числу.

Решение 7.43.1

```
(defun count-number (a w)
```

```
(cond ((null w) 0)
      ((atom w) (if (= w a) 1 0))
      (t (+ (count-number a (car w))
             (count-number a (cdr w)))))

> (count-number 5 '(4 5 ((3 5 (4)) 2) 2 5 6))
3
```

Решение 7.43.2

```
(defun count-number (a w)
  (if w (if (atom w) (if (= w a) 1 0)
          (+ (count-number a (car w))
             (count-number a (cdr w))))
    0))

> (count-number 5 '(4 5 ((3 5 (4)) 2) 2 5 6))
3
```

Задача 7.44 pull.lisp

Some may say Ruby is a bad rip-off of Lisp or Smalltalk, and I admit that. But it is nicer to ordinary people.

Yukihiro Matsumoto

Определить функцию, которая принимает число и возвращает из заданного списка с подсписками и массивами наименьшее из четных чисел, больших аргумента.

Решение 7.44.1

```
(defun pull (w n)
  (when w
    (reduce #'min
            (remove-if #'(lambda (a) (or (oddp a) (<= a n)))
                    (flatten w)))))

(defun flat-arr (arr)
  (make-array
   (apply #'* (array-dimensions arr))
   :displaced-to arr))

(defun flatten (w &optional ac)
  (cond ((null w) ac)
        ((atom w) (cons w ac))
        ((flatten (if (arrayp (car w))
                      (array-list (flat-arr (car w)))
                      (car w))
                  (flatten (cdr w) ac)))))
```

```
(defun array-list (a)
  (loop for i below (array-total-size a)
        collect (row-major-aref a i)))

> (pull ' (57 (57 71) #2A((11 44 55 55) (22 33 44 55))) 33)
44
```

Задача 7.45 *summ.lisp*

The main advantage of S-expressions is that it exposes the structure of programs. This doesn't matter too much if you're focusing on what the program is supposed to do, but once you get used to it, it doesn't disturb either — certainly no more than any other syntax, especially on a computer. But since you know how simple and regular the syntax is, you can easily write programs that process your programs — for example, send them to many machines for parallel execution, analyze to check some particular properties, or transform to have them executed as various different platforms. In particular, it allows programmers to fix the flaws in the programming language they are using rather easily (if they are using a LISP variant with a sufficiently developed macro system, they don't even need to modify their workflow — which is necessary if you're generating code in other languages).

Panicz Godek

Определить функцию, возвращающую сумму положительных четных чисел списка меньше n.

Решение 7.45.1

```
(defun summ (w n)
  (if w
      (+ (if (and (integerp (car w))
                  (< 0 (car w) n))
          (car w)
          0)
         (summ (cdr w) n))
      0))

> (summ ' (-1 1 2 3.5 4) 4)
3
```

Решение 7.45.2

```
(defun summ (w n &aux (a (car w)))
  (if w
    (+ (if (and (integerp a)
                (< 0 a n))
          a
          0)
      (summ (cdr w) n))
    0))
```

```
> (summ '(-1 1 2 3.5 4) 4)
3
```

Решение 7.45.3

```
(defun summ (w n)
  (loop for a in w
        when (and (integerp a) (< 0 a n))
        sum a))
```

```
> (summ '(-1 1 2 3.5 4) 4)
3
```

Задача 7.46 *elt-from-end.lisp*

Emacs is the world's best text editor. It's not just the best for editing program source; it's the best for any kind of text-editing. Mastering Emacs will make you more effective at writing and editing email, documentation drafts, blogs, HTML pages, XML files, and virtually everything else that requires any typing.

Steve Yegge

Определить функцию, возвращающую элемент списка по заданному номеру, считая от конца, учитывая элементы подсписков. Например, результатом применения функции к аргументам 2 '(a (b c (d)) e) должен быть d (второй с конца).

Решение 7.46.1

```
(defun elt-from-end (n w) (car (last (flat w nil) n)))

(defun flat (w acc)
  (cond ((null w) acc)
        ((atom w) (cons w acc))
        ((flat (car w) (flat (cdr w) acc)))))
```

```
> (elt-from-end 2 '( 1 2 3 4 5))
4
> (elt-from-end 2 '( 1 2 (3 (4)) 5))
```

4

Решение 7.46.2

```
(defun elt-from-end (n w) (nth (1- n) (reverse (flat w nil))))

(defun flat (w)
  (loop for a in w
        if (listp a) append (flat a)
        else collect a))

> (elt-from-end 2 '( 1 2 3 4 5))
4
> (elt-from-end 2 '( 1 2 (3 (4)) 5))
4
```

Решение 7.46.3

```
(defun elt-from-end (n w) (nth (1- n) (reverse (flat w nil))))

(defun flat (w) (if (atom w) `(,w) (mapcan #'flat w)))

> (elt-from-end 2 '( 1 2 3 4 5))
4
> (elt-from-end 2 '( 1 2 (3 (4)) 5))
4
```

Задача 7.47 our-position.lisp

The most useful tool I have come across till now is the treatment of functions like first-class-citizens. The way Lisp programmers think of data and functions is perhaps unique. They make no distinction at all. And no other language that is "mainstream" has yet succeeded in incorporating functions in data-structures like Lisp has.

Hrishikesh S

Определить функцию, возвращающую положение атома в списке (возвращала бы порядковый номер атома в списке).

Решение 7.47.1

```
(defun our-position (a w &optional (n 0))
  (cond ((null w) nil)
        ((equal (car w) a) n)
        (t (our-position a (cdr w) (1+ n)))))

> (our-position 'b '(a b c))
1
```

Задача 7.48 *our-nth.lisp*

I often tell people there are two languages they should learn to become better programmers: C and Lisp. With C you learn a lot of low level details about how computers work and this is helpful in all areas of computer programming. Even Flink boasts they do C++-style memory management on the JVM in Scala and for the things one uses Flink for that's a good thing. C is also a really great tool for really thinking through problems and imperative solutions. Sloppy C code is a product of a sloppy mind, and if you cannot program C responsibly, what a mess you will create with Scala! The second language on the list is Lisp because there you learn how to think about code as data and data as code. This becomes the basis of functional programming in many other languages and FP in C is, well, annoying and not well abstracted. With these two languages you get a really good overview of a lot of aspects of computing theory and practice, and it makes you a better programmer when it comes to learning other languages, whether newer imperative languages like Go or newer FP languages like Haskell.

Chris Travers

Определить функцию, возвращающую элемент списка по заданному номеру.

Решение 7.48.1

```
(defun our-nth (n w)
  (cond ((null w) nil)
        ((zerop n) (car w))
        (t (our-nth (1- n) (cdr w)))))

> (our-nth 1 '(a b c))
B
```

Задача 7.49 *rems-n.lisp*

Определить функцию, которая в исходном списке заменяет целые числа остатками от их деления на целое *n*.

Решение 7.49.1

```
(defun rems-n (w n)
  (cond ((null w) nil)
```

```
((integerp (car w)) (cons (rem (car w) n) (rem-n (cdr w) n)))
((cons (car w) (rem-n (cdr w) n))))
```

```
> (rem-n '(a 2 3) 2)
(A 0 1)
```

Решение 7.49.2

```
(defun rem-n (w n)
  (cond ((null w) nil)
        ((listp (car w)) (cons (rem-n (car w) n) (rem-n (cdr w) n)))
        ((integerp (car w)) (cons (rem (car w) n) (rem-n (cdr w) n)))
        ((cons (car w) (rem-n (cdr w) n)))))
```

```
> (rem-n '(a 2 (3)) 2)
(A 0 (1))
```

Решение 7.49.3

```
(defun rem-n (w n)
  (loop for a in w
        if (listp a) collect (rem-n a n)
        else if (integerp a) collect (rem a n)
        else collect a))
```

```
> (rem-n '(a 2 (3)) 2)
(A 0 (1))
```

Задача 7.50 cntt.lisp

Rewarding incompetence and ignorance increases the number of incompetent programmers. Designing programming languages and tools so incompetent programmers can feel better about themselves is not the way to go.

Erik Naggum

Определить функцию, считающую число элементов больших n в списке w на всех уровнях.

Решение 7.50.1

```
(defun cntt (w b &aux (a (car w)))
  (cond ((null w) 0)
        ((listp a) (+ (cntt a b) (cntt (cdr w) b)))
        (t (+ (if (> a b) 1 0) (cntt (cdr w) b)))))
```

```
> (cntt '(1 ((2 ((3 (4))))) 5)) 2)
3
```

Решение 7.50.3


```
(defun cntt (w b &aux (a (car w)))
  (if w
    (+ (if (listp a)
          (cnttt a b)
          (if (> a b)
              1
              0))
      (cntt (cdr w) b))
    0))

> (cntt '(1 ((2 ((3 (4))))) 5)) 2)
3
```

Задача 7.51 *deep-nth.lisp*

For people in the Operations Research industry — broadly trained mathematicians with a knack for applying their skills in practice, who lack the training, patience and passion for the nitty-gritty of Computer Science — Common Lisp is the answer. Prototyping to test an idea or algorithm — to get quick results for your publication if you will — would not and should not have to incur the overhead of “researching”. Common Lisp allows you to focus on what OR people do best; designing that algorithm.

Marc Kuo

Определить функцию, возвращающую n -ый атом многоуровневого списка.

Решение 7.51.1

```
(defun deep-nth (w n)
  (nth (1- n) (flatten w)))

(defun flatten (w)
  (mapcan #'(lambda (a) (if (listp a) (flatten a)
                             `(,a)))
          w))

> (deep-nth '((a k) ((b d) (q w) f) g) 5)
Q
```

Задача 7.52 *count<=.lisp*

Определить функцию, возвращающую количество элементов списка w , не превосходящих n .

Решение 7.52.1

```
(defun count<= (n w)
  (+ (if (> (car w) n) 0 1)
     (if (> (cadr w) n) 0 1)
     (if (> (caddr w) n) 0 1)))

> (count<= 2 '(1 2 3))
2
```

Решение 7.52.2

```
(defun count<= (n w)
  (+ (if (> (first w) n) 0 1)
     (if (> (second w) n) 0 1)
     (if (> (third w) n) 0 1)))

> (count<= 2 '(1 2 3))
2
```

Решение 7.52.3

```
(defun count<= (n w) (loop for a in w when (<= a n) count a))

> (count<= 2 '(1 2 3))
2
```

Решение 7.52.4

```
(defun count<= (n w) (loop for a in w unless (> a n) count a))

> (count<= 2 '(1 2 3))
2
```

Решение 7.52.5

```
(defun count<= (n w) (count-if #'(lambda (a) (<= a n)) w))

> (count<= 2 '(1 2 3))
2
```

Решение 7.52.6

```
(defun count<= (n w) (count-if-not #'(lambda (a) (> a n)) w))

> (count<= 2 '(1 2 3))
2
```

Решение 7.52.7

```
(defun count<= (n w)
  (cond ((null w) 0)
        ((<= (car w) n) (1+ (count<= n (cdr w)))))
```

```
(t (count<= n (cdr w))))
```

```
> (count<= 2 '(1 2 3))
2
```

Решение 7.52.8

```
(defun count<= (n w)
  (cond ((null w) 0)
        ((> (car w) n) (count<= n (cdr w)))
        (t (1+ (count<= n (cdr w))))))
```

```
> (count<= 2 '(1 2 3))
2
```

Решение 7.52.9

```
(defun count<= (n w)
  (if w (+ (if (<= (car w) n) 1 0)
          (count<= n (cdr w)))
      0))
```

```
> (count<= 2 '(1 2 3))
2
```

Решение 7.52.10

```
(defun count<= (n w)
  (if w (+ (if (> (car w) n) 0 1)
          (count<= n (cdr w)))
      0))
```

```
> (count<= 2 '(1 2 3))
2
```

Решение 7.52.11

```
(defun count<= (a w)
  (reduce #'+ w :key #'(lambda (e) (if (<= e a) 1 0))))
```

```
> (count<= 2 '(1 2 3))
2
```

Решение 7.52.12

```
(defun count<= (a w)
  (reduce #'+ w :key #'(lambda (e) (if (> e a) 0 1))))
```

```
> (count<= 2 '(1 2 3))
2
```

Решение 7.52.13

```
(defun count<= (n w &optional (m 0))
  (loop for a in w do (when (<= a n) (incf m))
    finally (return m)))
```

```
> (count<= 2 '(1 2 3))
2
```

Решение 7.52.14

```
(defun count<= (n w &optional (m 0))
  (loop for a in w do (unless (> a n) (incf m))
    finally (return m)))
```

```
> (count<= 2 '(1 2 3))
2
```

Задача 7.53 cnt-object.lisp

I love emacs even though I now there are a lot more "user friendly" ways to do the things I do in emacs. I love it for exactly the same reason you are writing about which is the challenge. I'll also admit that there is a bit of an ego thing for me as well. Total geek cred when I tell people I use emacs to post to my blog, organize my life, take notes, code in Python and UniBASIC, listen to music, and chat on IRC.

~

Дан простой список. Определить функцию, возвращающую число заданных объектов.

Решение 7.53.1

```
(defun cnt-object (a w)
  (cond ((null w) 0)
        ((+ (if (equalp a (car w)) 1 0) (cnt-object a (cdr w))))))
```

```
> (cnt-object 'z '(z a z b z))
3
```

Решение 7.53.2

```
(defun cnt-object (a w)
  (loop for e in w
    when (equalp a e) count e))
```

```
> (cnt-object 'z '(z a z b z))
3
```

Задача 7.54 *inside.lisp*

Dijkstra - I remember it line-by-line, and I have clear idea what it is about and what each particular line does. Floyd-Warshall - I remember it line-by-line (not so hard, since there are only 4 lines), and I have clear idea what it does and why, and don't have to think why the order of cycles is like that etc. Maximum flow - the only one I remember is Dinic. I know it line-by-line, and even though I have general idea about what it does and why, and what's the point of each particular line, I don't bother myself thinking about it when coding it - it's just some code to type. I have an idea about several other maximum flow algorithms, like Ford-Fulkerson, but it will take me some effort to code them, though I believe I can do it by myself in pretty reasonable time. HLD - I only coded it ~2 times in my life, last one being definitely more than a year ago, back when I was doing competitive programming seriously and taking part in ICPC. I think I'll manage to implement it by myself if needed, but it will take notable amount of time and effort; also, I don't recall how proper implementation should look like (even remotely), so if you'll ask me to actually implement it without checking any prewritten implementation - the outcome is likely going to be rather ugly. I'm not sure what is the exact point of your question; you are curious if they know these algorithms and can code them or just take prewritten implementation? It depends - some people can implement pretty much everything, some people don't bother remembering algorithms in details and only know how to use them as black box for particular problem. Most of those who have library of prewritten algorithms can actually implement these algorithms by themselves if needed, but simply prefer using prewritten stuff to save time or to avoid "boring" part.

Bohdan Pryshchenko, SEERC 2014 winner

Написать функцию для списка с произвольной иерархией, которая определяет принадлежность атома *e* списку *w*.

Решение 7.54.1

```
(defun inside (e w)
  (cond ((null w) nil) ((equal (car w) e))
        ((listp (car w)) (or (inside e (car w)) (inside e (cdr w))))
        ((inside e (cdr w)))))
```

```
> (inside 'a '(q (a w) e r))
T
```

Решение 7.54.2

```
(defun inside (e w &aux (a (car w)) (d (cdr w)))
  (cond ((null w) nil) ((equal a e))
        ((listp a) (or (inside e a) (inside e d)))
        ((inside e d)))))
```

```
> (inside 'a '(q (a w) e r))
T
```

Решение 7.54.3

```
(defun inside (e w)
  (when w (cond ((equal a e))
                 ((listp a) (or (inside e a) (inside e d)))
                 ((inside e d)))))
```

```
> (inside 'a '(q (a w) e r))
T
```

Решение 7.54.4

```
(defun inside (e w)
  (loop for a in w thereis
        (or (equal a e) (when (listp a) (inside e a)))))
```

```
> (inside 'a '(q (a w) e r))
T
```

Задача 7.55 w-pull.lisp

Определить функцию, которая принимает число и возвращает из заданного списка чисел и символов с подсписками и массивами наименьшее из чётных чисел, больших аргумента.

Решение 7.55.1

```
(defun w-pull (w n)
  (when w (reduce #'min
                  (remove-if #'(lambda (a) (or (oddp a) (<= a n)))
                           (remove-if #'symbolp (flatten w))))))
```

```
(defun flat-arr (arr)
```

```

(make-array (apply #'* (array-dimensions arr))
            :displaced-to arr))

(defun flatten (w &optional ac)
  (cond ((null w) ac)
        ((atom w) (cons w ac))
        ((flatten (if (arrayp (car w))
                      (array-list (flat-arr (car w)))
                      (car w))
                 (flatten (cdr w) ac)))))

(defun array-list (a)
  (loop for i below (array-total-size a)
        collect (row-major-aref a i)))

> (w-pull '(57 a (57 71) #2A((11 a 44 55 55) (22 a 33 44 55))) 33)
44

```

Решение 7.55.2

```

(defun w-pull (w n)
  (when w (reduce #'min
                  (remove-if #'(lambda (a) (or (oddp a) (<= a n)))
                          (flatten w)))))

(defun flat-arr (arr)
  (make-array (apply #'* (array-dimensions arr))
              :displaced-to arr))

(defun flatten (w &optional ac)
  (cond ((null w) ac)
        ((atom w) (if (symbolp w) ac (cons w ac)))
        ((flatten (if (arrayp (car w))
                      (array-list (flat-arr (car w)))
                      (car w))
                 (flatten (cdr w) ac)))))

(defun array-list (a)
  (loop for i below (array-total-size a)
        collect (row-major-aref a i)))

> (w-pull '(57 a (57 71) #2A((11 a 44 55 55) (22 a 33 44 55))) 33)
44

```

Задача 7.56 *diagonal-difference.lisp*

С клавиатуры последовательно вводятся: в первой строке n (размер квадратной матрицы), в последующих строках вводятся строки матрицы – числа, разделенные пробелами. Определить функцию, выводящую на экран абсолютную разницу сумм диагоналей матрицы.

Решение 7.56.1

```
(defun diagonal-difference ()
  (dd (loop for a from 1 to (read) collect
    (read-from-string
      (concatenate 'string "("
                    (read-line)
                    ")")))))
```

```
(defun dd (w)
  (abs (- (matrix-main w)
    (matrix-main (reverse w)))))
```

```
(defun matrix-main (w &optional (n 0))
  (if (null w) 0
    (+ (nth n (car w))
      (matrix-main (cdr w) (1+ n)))))
```

```
> (write (diagonal-difference))
3
11 2 4
4 5 6
10 8 -12
15
15
```

Решение 7.56.2

```
(defun diagonal-difference ()
  (format t "~a"
    (dd (loop for a from 1 to (read) collect
      (read-from-string
        (concatenate 'string
                      "("
                      (read-line)
                      ")")))))
```

```
(defun dd (w)
  (abs (- (matrix-main w)
    (matrix-main (reverse w)))))
```

```
(defun matrix-main (w &optional (n 0))
  (if (null w) 0
    (+ (nth n (car w))
      (matrix-main (cdr w) (1+ n)))))
```

```
> (diagonal-difference)
3
11 2 4
4 5 6
10 8 -12
15
NIL
```


Задача 7.57 twice.lisp

I mostly don't care how popular Lisp is. I am not using Lisp in order to score better in a popularity contest. I am using Lisp because it is the best programming language I know for the kind of programs I write. I don't think there is anything particularly or seriously wrong with Lisp. It might be that at the moment it does not provide what some people have come to expect from a programming language (free cross-platform implementation with all the libraries you could possibly want). Whether this will ever happen, I don't know, and I don't care (though I respect that others might). I do think I know that some of the people who want this to happen will have to get their hands dirty and just do it. No amount of lamenting is going to magically create any libraries.

Robert Strandh

Определить функцию, которая возвращает Т, если в списке w хотя бы дважды встречается элемент a.

Решение 7.57.1

```
(defun twice (w a &optional (n 2))
  (cond ((zerop n)
        ((null w) nil)
        ((equalp (car w) a) (twice (cdr w) a (1- n)))
        ((twice (cdr w) a n))))
```

```
> (twice '(a b b) 'b)
T
```

Решение 7.57.2

```
(defun twice (w a &optional (n 2))
  (cond ((zerop n)
        ((when w
            (twice (cdr w)
                    a
                    (- n (if (equalp (car w) a) 1 0)))))))
```

```
> (twice '(a b b) 'b)
T
```

Решение 7.57.3

```
(defun twice (w a)
  (if (cdr (remove-if-not #'(lambda (e) (equalp e a)) w)) t))

> (twice '(a b b) 'b)
T
```

Решение 7.57.4

```
(defun twice (w a)
  (loop for e in w
        when (equalp e a) count e into n
        when (= n 2) return t))

> (twice '(a b b) 'b)
T
```

Решение 7.57.5

```
(defun twice (w a &optional (n 2) &aux (v (member a w)))
  (cond ((zerop n)
        ((when v (twice (cdr v) a (1- n))))))

> (twice '(a b b) 'b)
T
```

Решение 7.57.6

```
(defun twice (w a &optional (n 2) &aux (v (member a w)))
  (if (zerop n) t
      (when v (twice (cdr v) a (1- n)))))

> (twice '(a b b) 'b)
T
```

Решение 7.57.7

```
(defun twice (w a) (> (count a w) 1))

> (twice '(a b b) 'b)
T
```

Задача 7.58 cnt.lisp

Определить функцию, которая подсчитывает количество вхождений элемента в список.

Решение 7.58.1

```
(defun cnt (a w)
  (if w (+ (if (equal (car w) a) 1 0) (cnt a (cdr w))) 0))

> (cnt 'a '(a b a b a a c a))
```

5

Решение 7.58.2

```
(defun cnt (a w)
  (loop for e in w
        when (equal e a) count e))

> (cnt 'a '(a b a b a a c a))
5
```

Решение 7.58.3

```
(defun cnt (a w) (count a w))

> (cnt 'a '(a b a b a a c a))
5
```

Задача 7.59 cnt-sine.lisp

Победы техники наших дней зиждутся на математике; никакой доблести в плохой успеваемости по этому предмету не осталось вовсе — прошли те времена. Школьная математика — отдельное занятие, нацеленное, в основном, на сдачу экзаменов. Плотное расписание тоже не способствует погружению в предмет, а математика не терпит поспешности. Для впитывания математических идей требуется время. Некоторые величайшие математики бывали до крайности непроворны в своих попытках постичь глубинные принципы этой дисциплины.

Тони Крили «Математика. 50 идей, о которых нужно знать»

Определить функцию, вычисляющую, сколько всего атомов в данном списке, исключая заданный атом.

Решение 7.59.1

```
(defun cnt-sine (a w)
  (length (remove a (ftn w))))

(defun ftn (w)
  (loop for a in w
        if (listp a) append (ftn a)
        else collect a)
  0))

> (cnt-sine 2 '(1 2 (3 (2 4)) (5 2)))
4
```

Решение 7.59.2

```
(defun cnt-sine (e w &aux (a (car w)))
  (if w (cond ((listp a) (+ (cnt-sine e a) (cnt-sine e (cdr w))))
        ((= a e) (cnt-sine e (cdr w)))
        ((1+ (cnt-sine e (cdr w))))
    0))

> (cnt-sine 2 '(1 2 (3 (2 4)) (5 2)))
4
```

Решение 7.59.3

```
(defun cnt-sine (e w &aux (a (car w)))
  (if w (+ (cond ((listp a) (cnt-sine e a))
              ((= a e) 0)
              (t 1))
    (cnt-sine e (cdr w)))
  0))

> (cnt-sine 2 '(1 2 (3 (2 4)) (5 2)))
4
```

Структура 8 (функция (СПИСОК) АТОМ) > (СПИСОК)

If, however, one factor is too successful, it will continue to be the winning factor regardless of the variation in the other factors over the range of variation in the conditions, and therefore will stifle the development of other advantageous factors until the conditions change sufficiently that it no longer is the winning factor. At this point, the whole population is ill prepared for the change, and may well perish entirely if the winning factor accidentally becomes the matching factor for a disease or a predator.

Erik Naggum

Чтобы обнаруживать ошибки, программист должен иметь ум, которому доставляет удовольствие находить изъяны там, где, казалось, царят красота и совершенство.

Фредерик Брукс

Книги наши толще, а вера наша полнее...

Николай Лесков «Сказ о тульском косом левше и о стальной блохе»

Задача 8.1 *_mapcar.lisp*

Определить функцию применения другой переданной в качестве параметра функции ко всем элементам заданного списка.

Решение 8.1.1

```
(defun _mapcar (p w)
  (when w (cons (funcall p (car w)) (_mapcar p (cdr w)))))

> (_mapcar #'1- '(1 2 3))
(0 1 2)
```

До тех пор - when, пока не опустел (существует) список w включаем - cons результат применения функции p, переданной в качестве параметра определяемой функции _mapcar, к первому элементу (car w) списка w в начало списка, полученного в результате вызова определяемой функции _mapcar с параметрами: p и хвостом (cdr w) списка w.

Задача 8.2 multiply.lisp

Определить функцию, которая умножает элементы списка (числа) на заданное число.

Решение 8.2.1

```
(defun multiply (w n)
  (cond ((null w) nil)
        ((cons (* (car w) n) (multiply (cdr w) n)))))
```

```
> (multiply '(1 2) 2)
(2 4)
```

Комментарии (построчно):

1. Определяем функцию `multiply` с параметрами: список **w** и число **n**.
2. Если (**cond**) пустой (**null**) список **w**, функция `multiply` возвращает **nil**.
3. При невыполнении условия **вставляет** (**cons**) произведение первого элемента (**car w**) и числа **n** в результат вызова `multiply` с параметрами: списком без первого элемента (**cdr w**) и числом **n**.

Решение 8.2.2

```
(defun multiply (w n)
  (if (null w) nil (cons (* (car w) n) (multiply (cdr w) n))))
```

```
> (multiply '(1 2) 2)
(2 4)
```

Решение 8.2.3

```
(defun multiply (w n)
  (when w (cons (* (car w) n) (multiply (cdr w) n))))
```

```
> (multiply '(1 2) 2)
(2 4)
```

Решение 8.2.4

```
(defun multiply (w n) (mapcar #'(lambda (a) (* a n)) w))
```

```
> (multiply '(1 2) 2)
(2 4)
```

Решение 8.2.5

```
(defun multiply (w n) (loop for a in w collect (* a n)))
```

```
> (multiply '(1 2) 2)
(2 4)
```

Задача 8.3 *drop-elm.lisp*

Определить функцию, которая удаляет из списка все совпадающие с данным атомом элементы и возвращает в качестве значения список из остатка элементов.

Решение 8.3.1

```
(defun drop-elm (n w)
  (cond ((null w) nil)
        ((eql (car w) n) (drop-elm n (cdr w)))
        ((cons (car w) (drop-elm n (cdr w))))))
```

```
> (drop-elm 3 '(1 2 3))
(1 2)
```

Комментарии (построчно):

1. Определяем функцию **drop-elm** с двумя параметрами: атом **n** и список **w**.
2. Если (**cond**) пустой (**null**) список **w** функция **drop-elm** возвращает **nil**
3. Если первый элемент списка (**car w**) равен **n** вызывается функция **drop-elm** с параметрами: **n** и список без первого элемента (**cdr w**).
4. При невыполнении 1 и 2 условий вставляет (**cons**) первый элемент (**car w**) в результат вызова **drop-elm** с параметрами: **n** и списком без первого элемента (**cdr w**).

Решение 8.3.2

```
(defun drop-elm (a w)
  (when w (if (equal a (car w))
              (drop-elm a (cdr w))
              (cons (car w) (drop-elm a (cdr w))))))
```

```
> (drop-elm 3 '(1 2 3))
(1 2)
```

Задача 8.4 *add-elt.lisp*

Проверить есть ли символ в списке, если нет, то добавить его в список.

Решение 8.4.1

```
(defun add-elt (e w)
  (cond ((member e w) ((cons e w))))
```

```
> (add-elt '1 '(2 3))
(1 2 3)
> (add-elt 'a '(b c))
(A B C)
> (add-elt 'b '(b c))
(B C)
```

Решение 8.4.2

```
(defun add-elt (e w) (if (member e w) w (cons e w)))

> (add-elt '1 '(2 3))
(1 2 3)
> (add-elt 'a '(b c))
(A B C)
> (add-elt 'b '(b c))
(B C)
```

Решение 8.4.3

```
(defun add-elt (e w) (adjoin e w))

> (add-elt '1 '(2 3))
(1 2 3)
> (add-elt 'a '(b c))
(A B C)
> (add-elt 'b '(b c))
(B C)
```

Задача 8.5 *group2.lisp*

Написать функцию, которая разбивает исходный список на пары (a1 a2) (a3 a4)...

Решение 8.5.1

```
(defun group2 (w)
  (cond ((cadr w) (cons (list (car w) (cadr w)) (group2 (cddr w))))
        (w (cons w nil))))

> (group2 '(a b c d e f))
((A B) (C D) (E F))
```

Решение 8.5.2

```
(defun group2 (w)
  (loop for v on w by #'cddr collect
        (if (cadr v) (subseq v 0 2) (subseq v 0))))

> (group '(a b c d e f))
((A B) (C D) (E F))
```

Задача 8.6 *blast-els.lisp*

Необходимо удалить n последних элементов из списка и вывести первоначальный список без удаленных элементов.

Решение 8.6.1


```
(defun blast-els (w n)
  (delete-if #'identity w :count n :from-end t))

> (blast-els '(1 2 3 4 5) 2)
(1 2 3)
```

Решение 8.6.2

```
(defun blast-els (w n)
  (butlast w n))

> (blast-els '(1 2 3 4 5) 2)
(1 2 3)
```

Задача 8.7 delete-first.lisp

Определить функцию, которая удаляет из списка первое вхождение заданного атома на верхнем уровне.

Решение 8.7.1

```
(defun delete-first (a w)
  (cond ((null w) nil)
        ((equal a (car w)) (cdr w))
        ((cons (car w) (delete-first a (cdr w))))))

> (delete-first 'a '(a b c (a b c)))
(B C (A B C))
> (delete-first 'a '((a b) c a b c))
((A B) C B C)
```

1. Если (null w), то достигли конца списка, вернули nil - пустой список (к нему в глубине рекурсии будет присоединен результат выполнения третьего условия, если элемент не будет найден);
2. Если (equal a (car w)) - найден нужный элемент, то вернули (cdr w) хвост списка;
3. Если не выполнилось ни одно из условий, то (cons - создаем список из (car w) - головы списка и (delete-first a (cdr w)) результата вызова функции с параметрами элемент и хвост списка.

Решение 8.7.2

```
(defun delete-first (a w)
  (cond ((or (null w) (eql a (car w))) (cdr w))
        ((cons (car w) (delete-first a (cdr w))))))

> (delete-first 'a '((a b) c a b c))
((A B) C B C)
> (delete-first 'a '(a b c (a b c)))
(B C (A B C))
> (delete-first 'a '((a b) c (a b c)))
((A B) C (A B C))
```

Решение 8.7.3

```
(defun delete-first (a w)
  (if (or (null w) (eql a (car w)))
      (cdr w)
      (cons (car w) (delete-first a (cdr w)))))

> (delete-first 'a '((a b) c (a b c)))
((A B) C (A B C))
> (delete-first 'a '(a b c (a b c)))
(B C (A B C))
> (delete-first 'a '((a b) c a b c))
((A B) C B C)
```

Задача 8.8 frequent.lisp

Дан список. Определить функцию, находящую элементы в списке встречающиеся ровно n раз.

Решение 8.8.1

```
(defun frequent (w n)
  (remove-if-not
   #'(lambda (a) (= (count a w) n)) (remove-duplicates w)))

> (time (frequent '(1 2 3 4 5 5 4 5 1 2 1 2) 3))
Real time: 0.0 sec.
Run time: 0.0 sec.
Space: 520 Bytes
(5 1 2)
```

Решение 8.8.2

```
(defun frequent (w n)
  (delete-if-not
   #'(lambda (a) (= (count a w) n)) (remove-duplicates w)))

> (time (frequent '(1 2 3 4 5 5 4 5 1 2 1 2) 3))
Real time: 0.0 sec.
Run time: 0.0 sec.
Space: 496 Bytes
(5 1 2)
```

Решение 8.8.3

```
(defun frequent (w n)
  (loop for a in (remove-duplicates w)
        when (= (count a w) n) collect a))

> (time (frequent '(1 2 3 4 5 5 4 5 1 2 1 2) 3))
Real time: 0.0 sec.
```

```
Run time: 0.0 sec.
Space: 468 Bytes
(5 1 2)
```

Решение 8.8.4

```
(defun frequent (w n &optional (v (remove-duplicates w))
                  &aux (a (car v)))
  (cond ((null v) nil)
        ((= (count a w) n) (cons a (frequent w n (cdr v))))
        ((frequent w n (cdr v)))))

> (time (frequent '(1 2 3 4 5 5 4 5 1 2 1 2) 3))
Real time: 0.0 sec.
Run time: 0.0 sec.
Space: 468 Bytes
(5 1 2)
```

Решение 8.8.5

```
(defun frequent (w n &optional (v (remove-duplicates w))
                  &aux (a (car v)))
  (when v (if (= (count a w) n)
              (cons a (frequent w n (cdr v)))
              (frequent w n (cdr v)))))

> (time (frequent '(1 2 3 4 5 5 4 5 1 2 1 2) 3))
Real time: 0.0 sec.
Run time: 0.0 sec.
Space: 468 Bytes
(5 1 2)
```

Решение 8.8.6

```
(defun quent (v w n)
  (loop for a in v when (= (count a w) n) collect a))

(defun frequent (w n) (quent (remove-duplicates w) w n))

> (time (frequent '(1 2 3 4 5 3 3 3 5 4 5 1 2 1 2) 3))
Real time: 0.0010001 sec.
Run time: 0.0 sec.
Space: 468 Bytes
(5 1 2)
```

Решение 8.8.7

```
(defun frequent (w n)
  (cond ((null w) nil)
        ((= n (count (car w) w))
         (cons (car w) (frequent (remove (car w) (cdr w)) n)))
        ((frequent (remove (car w) (cdr w)) n))))
```

```
> (time (frequent '(1 2 3 4 5 3 3 3 5 4 5 1 2 1 2) 3))
Real time: 0.0 sec.
Run time: 0.0 sec.
Space: 316 Bytes
(1 2 5)
```

Решение 8.8.8

```
(defun frequent (w n)
  (cond ((null w) nil)
        ((= n (count (car w) w))
         (cons (car w) (frequent (delete (car w) (cdr w)) n)))
        ((frequent (delete (car w) (cdr w)) n))))

> (time (frequent '(1 2 3 4 5 3 3 3 5 4 5 1 2 1 2) 3))
Real time: 0.0 sec.
Run time: 0.0 sec.
Space: 84 Bytes
(1 2 5)
```

Задача 8.9 mapping.lisp

Определить функцию, аргументами которой являются функция и список, рассматриваемый как множество, а результат применения – множество (список без повторяющихся элементов), которое можно получить применением функции-аргумента, к каждому элементу списка. Порядок элементов не имеет значения.

Решение 8.9.1

```
(defun mapping (f w) (remove-duplicates (mapcar f w)))

> (mapping #'(lambda (z) (/ (- z (rem z 10)) 10)) '(1 57 101 102 53 6))
(10 5 0)
```

Решение 8.9.2

```
(defun sine-duos (w)
  (cond ((null w) nil)
        ((member (car w) (cdr w)) (sine-duos (cdr w)))
        ((cons (car w) (sine-duos (cdr w)))))

(defun mapping (f w)
  (reverse
   (sine-duos (mapcar #'(lambda (a) (funcall f a)) w))))

> (mapping (lambda (z) (/ (- z (rem z 10)) 10)) '(1 57 101 102 53 6))
(0 5 10)
```

Решение 8.9.3

```

(defun -member (a w)
  (cond ((null w) nil)
        ((equalp a (car w)) (cons (car w) (cdr w)))
        ((member a (cdr w)))))

(defun duplicates- (w)
  (cond ((null w) nil)
        ((-member (car w) (cdr w)) (duplicates- (cdr w)))
        ((cons (car w) (duplicates- (cdr w)))))

(defun mapping (f w)
  (reverse
   (duplicates- (mapcar #'(lambda (a) (funcall f a)) w))))

> (mapping #'(lambda (z) (/ (- z (rem z 10)) 10)) '(1 57 101 102 53
6))
(0 5 10)

```

Задача 8.10 group-n.lisp

Создать функцию которая группирует список в n-элементные подписки.

Решение 8.10.1

```

(defun group-n (w n
                &optional acc ac (m n)
                &aux (a (car w)) (d (cdr w)))
  (cond ((null w) (reverse (if ac (cons (reverse ac) acc) acc)))
        ((= m 1) (group-n d n (cons (reverse (cons a ac)) acc) nil n))
        ((group-n d n acc (cons a ac) (1- m)))))

> (group-n '(1 2 3 4 5 6 7 8 9 10 11) 3)
((1 2 3) (4 5 6) (7 8 9) (10 11))
> (group-n '(1 2 3 4 5 6 7 8 9 10 11) 2)
((1 2) (3 4) (5 6) (7 8) (9 10) (11))

```

Решение 8.10.2

```

(defun group-n (w n
                &optional acc ac (m n)
                &aux (v (cons (car w) ac)) (z (= m 1)))
  (if w
      (group-n (cdr w)
                n
                (if z (cons (reverse v) acc) acc)
                (unless z v)
                (if z n (1- m)))
      (reverse (if ac (cons (reverse ac) acc) acc))))

> (group-n '(1 2 3 4 5 6 7 8 9 10 11) 3)
((1 2 3) (4 5 6) (7 8 9) (10 11))
> (group-n '(1 2 3 4 5 6 7 8 9 10 11) 2)

```

```
((1 2) (3 4) (5 6) (7 8) (9 10) (11))
```

Решение 8.10.3

```
(defun group-n (w n)
  (when w (if (< (length w) n)
              (list w)
              (cons (subseq w 0 n)
                    (group-n (subseq w n) n))))))
```

```
> (group-n '(1 2 3 4 5 6 7 8 9 10 11) 3)
((1 2 3) (4 5 6) (7 8 9) (10 11))
> (group-n '(1 2 3 4 5 6 7 8 9 10 11) 2)
((1 2) (3 4) (5 6) (7 8) (9 10) (11))
```

Задача 8.11 *quadros.lisp*

Дан список *w* и число *n*. Разбить список на подсписки длины *n*.

Решение 8.11.1

```
(defun quadros (w n &optional ac acc (m n))
  (cond ((null w) (reverse (cons (reverse ac) (reverse acc))))
        ((= m 1) (quadros (cdr w)
                           n
                           nil
                           (cons (reverse (cons (car w) ac)) acc)
                           n))
        ((quadros (cdr w) n (cons (car w) ac) acc (1- m)))))
```

```
> (quadros '(A B (C D) 1 2 NIL) 4)
((A B (C D) 1) (2 NIL))
```

Задача 8.12 *_remove-if-not.lisp*

Определите фильтр (удалить-если-не предикат список), удаляющий из списка элементы, которые не обладают свойством, наличие которого про-
веряет предикат.

Решение 8.12.1

```
(defun _remove-if-not (p w)
  (cond ((null w) nil)
        ((funcall p (car w))
         (cons (car w) (_remove-if-not p (cdr w))))
        ((_remove-if-not p (cdr w)))))
```

```
> (_remove-if-not #'evenp '(1 2))
(2)
```

Решение 8.12.2

```
(defun _remove-if-not (p w)
  (mapcan #'(lambda (a) (when (funcall p a) (list a))) w))

> (_remove-if-not #'evenp '(1 2))
(2)
```

Решение 8.12.3

```
(defun _remove-if-not (p w)
  (loop for a in w when (funcall p a) collect a))

> (_remove-if-not #'evenp '(1 2))
(2)
```

Задача 8.13 dive-elms.lisp

Написать программу с двумя параметрами (n – целое число, w – список), чтобы получить из списка w новый список из элементов, находящихся на уровне n . $n=1$ самый верхний уровень, возвращается сам список.

Решение 8.13.1

```
(defun dive-elms (n w) (if (= n 1) w (dive-elms (1- n) (elms w))))

(defun elms (w)
  (cond ((null w) nil)
        ((atom (car w)) (elms (cdr w)))
        ((nconc (car w) (elms (cdr w))))))

> (dive-elms 1 '((a ((b (c) d))) (a ((b (c) d)))))
((A ((B (C) D))) (A ((B (C) D))))
> (dive-elms 2 '((a ((b (c) d))) (a ((b (c) d)))))
(A ((B (C) D)) A ((B (C) D)))
> (dive-elms 3 '((a ((b (c) d))) (a ((b (c) d)))))
((B (C) D) (B (C) D))
> (dive-elms 4 '((a ((b (c) d))) (a ((b (c) d)))))
(B (C) D B (C) D)
> (dive-elms 5 '((a ((b (c) d))) (a ((b (c) d)))))
(C C)
```

Решение 8.13.2

```
(defun dive-elms (n w)
  (labels ((elms (w)
            (cond ((null w) nil)
                  ((atom (car w)) (elms (cdr w)))
                  ((nconc (car w) (elms (cdr w)))))))
    (if (= n 1) w (dive-elms (1- n) (elms w)))))

> (dive-elms 1 '((a ((b (c) d))) (a ((b (c) d)))))
((A ((B (C) D))) (A ((B (C) D))))
```

```

> (dive-elms 2 '((a ((b (c) d))) (a ((b (c) d)))))
(A ((B (C) D)) A ((B (C) D)))
> (dive-elms 3 '((a ((b (c) d))) (a ((b (c) d)))))
((B (C) D) (B (C) D))
> (dive-elms 4 '((a ((b (c) d))) (a ((b (c) d)))))
(B (C) D B (C) D)
> (dive-elms 5 '((a ((b (c) d))) (a ((b (c) d)))))
(C C)

```

Решение 8.13.3

```

(defun dive-elms (n w &aux (a (car w)) (d (cdr w)))
  (cond ((null w) nil)
        ((= n 1) w)
        ((atom a) (dive-elms n d))
        ((nconc (dive-elms (1- n) a) (dive-elms n d)))))

> (dive-elms 1 '((a ((b (c) d))) (a ((b (c) d)))))
(A ((B (C) D))) (A ((B (C) D)))
> (dive-elms 2 '((a ((b (c) d))) (a ((b (c) d)))))
(A ((B (C) D)) A ((B (C) D)))
> (dive-elms 3 '((a ((b (c) d))) (a ((b (c) d)))))
((B (C) D) (B (C) D))
> (dive-elms 4 '((a ((b (c) d))) (a ((b (c) d)))))
(B (C) D B (C) D)
> (dive-elms 5 '((a ((b (c) d))) (a ((b (c) d)))))
(C C)

```

Задача 8.14 *get-while.lisp*

Напишите функцию `(function w p)`, которая в качестве результата выдает список, состоящий из всех элементов списка `w`, с начала списка и до первого элемента, не удовлетворяющего данному предикату `p`. Например: `(function '(2 4 6 7 8 9) #'evenp)` возвращает `(2 4 6)`.

Решение 8.14.1

```

(defun get-while (w p)
  (when w (when (funcall p (car w))
                (cons (car w) (get-while (cdr w) p)))))

> (get-while '(2 4 6 5 7 4) #'evenp)
(2 4 6)

> (get-while nil #'evenp)
NIL

```

Решение 8.14.2

```

(defun get-while (w p)
  (when (and w (funcall p (car w)))
    (cons (car w) (get-while (cdr w) p))))

```



```
> (get-while '(2 4 6 5 7 4) #'evenp)
(2 4 6)
> (get-while nil #'evenp)
NIL
```

Решение 8.14.3

```
(defun get-while (w p)
  (cond ((null w) nil)
        ((funcall p (car w)) (cons (car w) (get-while (cdr w) p)))
        (nil)))

> (get-while '(2 4 6 5 7 4) #'evenp)
(2 4 6)
> (get-while nil #'evenp)
NIL
```

Решение 8.14.4

```
(defun get-while (w p)
  (let ((a (car w)))
    (cond ((null w) nil)
          ((funcall p a) (cons a (get-while (cdr w) p)))
          (nil))))

> (get-while '(2 4 6 5 7 4) #'evenp)
(2 4 6)
> (get-while nil #'evenp)
NIL
```

Решение 8.14.5

```
(defun get-while (w p &aux (a (car w)))
  (cond ((null w) nil)
        ((funcall p a) (cons a (get-while (cdr w) p)))
        (nil)))

> (get-while '(2 4 6 5 7 4) #'evenp)
(2 4 6)
> (get-while nil #'evenp)
NIL
```

Задача 8.15 *dump-level.lisp*

Написать функцию, удаляющую из исходного списка подписки заданной глубины. При описании функций нельзя использовать функции `set`, `setq`, `setf` и циклы.

Решение 8.15.1

```
(defun dump-level (w n)
```

```

(when w ((lambda (a d)
  (cond ((atom a) (cons a (dump-level d n)))
        ((= n 0) (dump-level d n))
        ((cons (dump-level a (1- n)) (dump-level d n))))))
(car w) (cdr w)))

> (dump-level '(5 (4 (6)) (1 7) (8 (3 (5)))) 0)
(5)
> (dump-level '(5 (4 (6)) (1 7) (8 (3 (5)))) 1)
(5 (4) (1 7) (8))
> (dump-level '(5 (4 (6)) (1 7) (8 (3 (5)))) 2)
(5 (4 (6)) (1 7) (8 (3)))
> (dump-level '(5 (4 (6)) (1 7) (8 (3 (5)))) 3)
(5 (4 (6)) (1 7) (8 (3 (5))))

```

Решение 8.15.2

```

(defun dump-level (w n &aux (a (car w)) (d (cdr w)))
  (cond ((null w) nil)
        ((atom a) (cons a (dump-level d n)))
        ((= n 0) (dump-level d n))
        ((cons (dump-level a (1- n)) (dump-level d n)))))

> (dump-level '(5 (4 (6)) (1 7) (8 (3 (5)))) 0)
(5)
> (dump-level '(5 (4 (6)) (1 7) (8 (3 (5)))) 1)
(5 (4) (1 7) (8))
> (dump-level '(5 (4 (6)) (1 7) (8 (3 (5)))) 2)
(5 (4 (6)) (1 7) (8 (3)))
> (dump-level '(5 (4 (6)) (1 7) (8 (3 (5)))) 3)
(5 (4 (6)) (1 7) (8 (3 (5))))

```

Задача 8.16 addn.lisp

Определите функцию (f w n), которая из списка чисел s создает новый список, прибавляя к каждому атому число n. Исходный список не предполагается одноуровневым.

Решение 8.16.1

```

(defun addn (w n &aux (a (car w)))
  (cond ((null w) nil)
        ((consp a) (cons (addn a n) (add (cdr w) n)))
        ((cons (+ a n) (addn (cdr w) n)))))

> (addn '(1 (2) (3 (4)) 5) 10)
(11 (12) (13 (14)) 15)

```

Решение 8.16.2

```

(defun addn (w n)
  (mapcar #'(lambda (a) (if (atom a)

```

```

                (+ a n)
                (addn a n)))
        w))

> (addn '(1 (2) (3 (4)) 5) 10)
(11 (12) (13 (14)) 15)

```

Задача 8.17 add.lisp

Определить функцию, добавляющую к каждому числовому элементу списка *w* заданное параметром число *n*.

Решение 8.17.1

```

(defun add (w n &aux (a (car w)))
  (cond ((null w) nil)
        ((numberp a) (cons (+ a n) (add (cdr w) n)))
        (t (cons a (add (cdr w) n)))))

> (add '(a -1 6 v 3) 3)
(A 2 9 V 6)

```

Решение 8.17.2

```

(defun add (w n &aux (a (car w)))
  (when w (cons (if (numberp a) (+ a n) a) (add (cdr w) n))))

> (add '(a -1 6 v 3) 3)
(A 2 9 V 6)

```

Решение 8.17.3

```

(defun add (w n)
  (loop for a in w if (numberp a) collect (+ a n) else collect a))

> (add '(a -1 6 v 3) 3)
(A 2 9 V 6)

```

Решение 8.17.4

```

(defun add (w n)
  (loop for a in w collect (if (numberp a) (+ a n) a)))

> (add '(a -1 6 v 3) 3)
(A 2 9 V 6)

```

Решение 8.17.5

```

(defun add (w n)
  (mapcar #'(lambda (a) (if (numberp a) (+ a n) a)) w))

> (add '(a -1 6 v 3) 3)

```

```
(A 2 9 V 6)
```

Задача 8.18 *plunge-in.lisp*

Добавить число в упорядоченный в порядке возрастания список без нарушения порядка. Например: (функция 7 '(1 3 5 8 11)) результат: (1 3 5 7 8 11).

Решение 8.18.1

```
(defun plunge-in (n w)
  (when w (let ((a (car w)) (d (cdr w)))
    (cond ((atom d) (list a n))
          ((> n a) (cons a (plunge-in n d)))
          ((cons n w))))))
```

```
> (plunge-in 7 '(1 3 5 8 11))
(1 3 5 7 8 11)
```

Решение 8.18.2

```
(defun plunge-in (n w)
  (when w ((lambda (a d)
    (cond ((atom d) (list a n))
          ((> n a) (cons a (plunge-in n d)))
          ((cons n w))))
    (car w) (cdr w))))
```

```
> (plunge-in 7 '(1 3 5 8 11))
(1 3 5 7 8 11)
```

Решение 8.18.3

```
(defun squeeze-in (n w)
  (let ((e))
    (loop for i from 0 below (length w) do
      ((lambda (u)
        (when (and (<= n u) (or (= i 0) (> n (nth (1- i) w))))
          (push n e))
        (push u e)
        (when (and (> n u) (= (length w) (1+ i)))
          (push n e)))
        (nth i w)))
    (nreverse e)))
```

```
> (squeeze-in 7 '(1 3 5 8 11))
(1 3 5 7 8 11)
```

Решение 8.18.4

```
(defun cram-in (n w)
  (sort (cons n w) #'<))
```

```
> (cram-in 7 '(1 3 5 8 11))
(1 3 5 7 8 11)
```

Решение 8.18.5

```
(defun set-in (n w)
  (cond ((null w) (cons n nil))
        ((> n (car w)) (cons (car w) (set-in n (cdr w))))
        ((cons n w))))

> (set-in 7 '(1 3 5 8 11))
(1 3 5 7 8 11)
```

Решение 8.18.6

```
(defun set-in (n w)
  (cond ((null w) `(,n))
        ((> n (car w)) (cons (car w) (set-in n (cdr w))))
        ((cons n w))))

> (set-in 7 '(1 3 5 8 11))
(1 3 5 7 8 11)
```

Решение 8.18.7

```
(defun set-in (n w &aux (a (car w)))
  (cond ((null w) `(,n))
        ((> n a) (cons a (set-in n (cdr w))))
        ((cons n w))))

> (set-in 7 '(1 3 5 8 11))
(1 3 5 7 8 11)
```

Решение 8.18.8

```
(defun set-in (n w &aux (a (car w)))
  (if w (if (> n a) (cons a (set-in n (cdr w))) (cons n w)) `(,n)))

> (set-in 7 '(1 3 5 8 11))
(1 3 5 7 8 11)
```

Решение 8.18.9 (автор - Bernhard Pfahringer, comp.lang.lisp)

```
(defun plunge-in (n w)
  (merge 'list (list n) w #'<))

> (plunge-in 7 '(1 3 5 8 11))
(1 3 5 7 8 11)
```

Решение 8.18.10

```
(defun plunge-in (n w) (merge 'list `(,n) w #'<))
```

```
> (plunge-in 7 '(1 3 5 8 11))
(1 3 5 7 8 11)
```

Задача 8.19 *substitute-first-or-last.lisp*

Есть список и некоторый объект. Написать функцию, которая возвращает новый список, в котором объект заменяет первый элемент списка, если первый элемент списка и объект является атомами, иначе заменяет последний элемент списка.

Решение 8.19

```
(defun substitute-first-or-last (w v)
  (if (and (atom (car w)) (atom v)) (cons v (cdr w))
      (append (butlast w) (list v))))

> (substitute-first-or-last '(a b c) 'z)
(Z B C)
```

Задача 8.20 *mask-list.lisp*

Дан список атомов. Вывести все элементы по заданной маске с указанием индекса элемента. Пример - дан: (1 a b 3 a c f a), маска (a), результат: (2 a 5 a 8 a).

Решение 8.20.1

```
(defun mask-list
  (w z &optional (n 1) &aux (a (car w)) (d (cdr w)))
  (cond ((null w) nil)
        ((eq a z) (cons n (cons z (mask-list d z (1+ n)))))
        ((mask-list d z (1+ n)))))

> (mask-list '(1 a b 3 a c f a) 'a)
(2 A 5 A 8 A)
```

Решение 8.20.2

```
(defun ne (w a n)
  (cond ((null w) nil)
        ((eq a (car w)) (cons n (cons a (ne (cdr w) a (1+ n)))))
        ((ne (cdr w) a (1+ n)))))

(defun mask-list (w a)
  (ne w a 1))

> (mask-list '(1 a b 3 a c f a) 'a)
(2 A 5 A 8 A)
```

Задача 8.21 repeat-elts.lisp

Напишите функцию `(f w n)`, которая выдает список, получающийся из списка `w` увеличением вхождения каждого элемента на `n`, например: `(f '(1 2 3) 4) = (1 1 1 1 2 2 2 2 3 3 3 3)`.

Решение 8.21.1

```
(defun repeat-elts (w n &optional (m n))
  (cond ((null w) nil)
        ((zerop m) (repeat-elts (cdr w) n n))
        ((cons (car w) (repeat-elts w n (1- m))))))

> (repeat-elts '(1 2 3) 4)
(1 1 1 1 2 2 2 2 3 3 3 3)
```

Задача 8.22 del-last-ns.lisp

Напишите функцию `(f w n)`, которая удаляет `n` элементов с конца списка `w`.

Решение 8.22.1

```
(defun del-last-ns (w n)
  (delete-if #'identity w :count n :from-end t))

> (del-last-ns '(a b) 1)
(A)
```

Решение 8.22.2

```
(defun del-last-ns (w n) (butlast w n))

> (del-last-ns '(a b) 1)
(A)
```

Задача 8.23 shift-left.lisp

Осуществить циклический сдвиг элементов списка влево, количество шагов задаётся.

Решение 8.23.1

```
(defun shift-left (w n)
  (append (nthcdr n w) (subseq w 0 n)))

> (shift-left '(a b c d e) 2)
(C D E A B)
```

Задача 8.24 *remove-i+n.lisp*

Дан список *w* и число *n*. Реализовать функцию, которая удаляет все *i+n* элементы списка.

Решение 8.24.1

```
(defun remove-i+n (w n)
  (when (and w (plusp n)) (cons (car w) (wx (cdr w) (1- n)))))

> (remove-i+n '(1 2) 1)
(1)
```

Задача 8.25 *insert-@.lisp*

Написать функционал, который ставит символ @ перед каждым элементом списка отвечающего критерию, например каждым элементом, который больше 0 или каждым нечетным элементом.

Решение 8.25.1

```
(defun insert-@ (w f &aux (a (car w)) (d (cdr w)))
  (cond ((null w) nil)
        ((funcall f a) (cons '@ (cons a (insert-@ d f))))
        ((cons a (insert-@ d f)))))

> (insert-@ '(1 0 -2 3 0 -4 5) #'evenp)
(1 @ 0 @ -2 3 @ 0 @ -4 5)
> (insert-@ '(1 0 -2 3 0 -4 5) #'(lambda (x) (>= x 0)))
(@ 1 @ 0 -2 @ 3 @ 0 -4 @ 5)
```

Переменная *w* – список; переменная *f* – функция; &aux предвеляет, вспомогательные параметры (*a* (*car w*)) (*d* (*cdr w*)), от "auxiliary" – вспомогательный, вместо него можно применить *let* в теле функции, или просто заменить все *a* на (*car w*) и все *d* на (*cdr w*).

Задача 8.26 *hide**.lisp***

Определить функционал, заменяющий все элементы списка, не обладающие определенным свойством, на символ *. Функционал для предикатов:

- число;
- неположительное число (при вызове используйте лямбда-функцию).

Решение 8.26.1

```
(defun hide**** (w f &aux (a (car w)) (d (cdr w)))
  (cond ((null w) nil)
        ((funcall f a) (cons a (hide**** d f)))
        ((cons '*' (hide**** d f)))))

> (hide**** '(a 1) #'numberp)
```



```
(* 1)

> (hide**** '(1 0) #'(lambda (x) (< x 0)))
(* *)
```

Задача 8.27 del-over-n.lisp

Удалить из подсписков числа больше n , с использованием спискоразрушающих функций. Например:

```
> (f '(1 2 (3 4 7) 8 9 (5 1)) 5)
(1 2 (3 4) 8 9 (5 1))
```

Решение 8.27.1

```
(defun del-over-n (w m &aux (a (car w)) (d (cdr w)))
  (cond((null w) nil)
        ((atom a) (cons a (del-over-n d m)))
        ((cons (delete-if #'(lambda (z) (> z m)) a)
                 (del-over-n d m)))))

> (del-over-n '(1 2 (3 4 7) 8 9 (5 1)) 5)
(1 2 (3 4) 8 9 (5 1))
```

Задача 8.28 delete-elt.lisp

Определить функцию, удаляющую заданный атом из списка.

Решение 8.28.1.

```
(defun delete-elt (w a)
  (cond ((null w) nil)
        ((eq (car w) a) (delete-elt (cdr w) a))
        ((cons (car w) (delete-elt (cdr w) a)))))

> (delete-elt '(a b a c) 'a)
(b c)
```

Задача 8.29 shift-right.lisp

Осуществить циклический сдвиг элементов списка вправо, количество шагов задаётся.

Решение 8.29.1

```
(defun shift-right (w n)
  (append (last w n) (butlast w n)))

> (shift-right '(a b c d e) 2)
(d e a b c)
```

Решение 8.29.2

```
(defun shift-right (w n &aux (m (if (minusp n) 0 n)))
  (append (last w m) (butlast w m)))

> (shift-right '(a b c d e) -1)
(A B C D E)
> (shift-right '(a b c d e) 0)
(A B C D E)
> (shift-right '(a b c d e) 1)
(E A B C D)
> (shift-right '(a b c d e) 2)
(D E A B C)
```

Решение 8.29.3

```
(defun shift-right (w n)
  ((lambda (w m)
    (append (last w m) (butlast w m)))
   w (if (minusp n) 0 n)))

> (shift-right '(a b c d e) -1)
(A B C D E)
```

Решение 8.29.4

```
(defun shift-right (w n)
  (let ((m (if (minusp n) 0 n)))
    (append (last w m) (butlast w m))))

> (shift-right '(a b c d e) -1)
(A B C D E)
```

Задача 8.30 rotate>.lisp

Осуществить n циклических сдвигов вправо элементов исходного списка.

Решение 8.30.1

```
(defun rotate> (w n)
  (if (zerop n) w
      (rotate> (nconc (last w) (butlast w)) (1- n))))

> (rotate> '(1 2 3 4 5) 2)
(4 5 1 2 3)
```

Решение 8.30.2

```
(defun rotate> (w n)
  (if (zerop n) w
      (rotate> (cons (car (last w)) (butlast w)) (1- n))))
```

```
> (rotate> '(1 2 3 4 5) 2)
(4 5 1 2 3)
```

Решение 8.30.3

```
(defun rotate> (w n)
  (nconc (last w n) (butlast w n)))

> (rotate> '(1 2 3 4 5) 2)
(4 5 1 2 3)
```

Задача 8.31 cut-n-elms.lisp

Написать программу формирующую список, состоящий из не более чем n элементов исходного списка.

Решение 8.31.1

```
(defun cut-n-elms (n w)
  (loop for a in w
        for b downfrom n unless (zerop b) collect a))

> (cut-n-elms 2 '(1 2 3))
(1 2)
```

Задача 8.32 ante-del.lisp

Удалите из списка перед каждым вхождением x один элемент, если такой имеется и отличен от x .

Решение 8.32.1

```
(defun ante-del (a w)
  (cond ((null w) nil)
        ((and (equalp (cadr w) a) (not (equalp (car w) a)))
         (ante-del a (cdr w)))
        ((cons (car w) (ante-del a (cdr w))))))

> (ante-del 7 '(7 2 7 7 3))
(7 7 7 3)
```

Решение 8.32.2

```
(defun ante-del (a w)
  (when w (if (and (equalp (cadr w) a) (not (equalp (car w) a)))
              (ante-del a (cdr w))
              (cons (car w) (ante-del a (cdr w))))))

> (ante-del 7 '(7 2 7 7 3))
(7 7 7 3)
```

Задача 8.33 list-a.lisp

Дан список (1 2 3 4) и число 5. Определить рекурсивную функцию, которая помещает 5 в конец списка (1 2 3 4).

Решение 8.33.1

```
(defun list-a (w a)
  (if (null w) `(,a) (cons (car w) (list-a (cdr w) a))))

> (list-a '(1 2 3 4) 5)
(1 2 3 4 5)
```

Задача 8.34 downsize.lisp

Дан список (1 2 3 4 5 6 7). Необходимо составить новый список из элементов, имеющих нечетные порядковые номера, уменьшенных в n-раз.

Решение 8.34.1

```
(defun downsize (n w)
  (when w (cons (/ (car w) n) (downsize n (cddr w)))))

> (downsize 3 '(1 2 3 4 5 6 7))
(1/3 1 5/3 7/3)
```

Задача 8.35 drop-num.lisp

Дан список и позиция элемента. Удалить элемент с заданным номером.

Решение 8.35.1

```
(defun drop-num (n w)
  (cond ((null w) nil)
        ((zerop n) (cdr w))
        ((cons (car w) (drop-num (1- n) (cdr w)))))

> (drop-num 3 '(0 1 2 3 4))
(0 1 2 4)
```

Решение 8.35.2

```
(defun drop-num (n w)
  (when w (if (zerop n)
              (cdr w)
              (cons (car w) (drop-num (1- n) (cdr w)))))

> (drop-num 3 '(0 1 2 3 4))
(0 1 2 4)
```



```

                                (dive n (cdr w))))))
      (dive n w))))))

> (dive-atoms 0 '(a (b)))
(A)
> (dive-atoms 1 '(a (b)))
(B)

```

Решение 8.36.4

```

(defun dive-atoms (n w)
  (labels ((atoms (w)
             (remove-if #'listp w))
            (dive (n w)
                   (cond ((null w) nil)
                         ((= n 0) w)
                         ((atom (car w)) (dive n (cdr w)))
                         ((append (dive (1- n) (car w))
                                   (dive n (cdr w)))))))
    (dive n w))))

> (dive-atoms 1 '(a (b)))
(B)

```

Решение 8.36.5

```

(defun dive-atoms (n w)
  (remove-if #'listp
    (labels ((dive (n w)
               (cond ((null w) nil)
                     ((= n 0) w)
                     ((atom (car w)) (dive n (cdr w)))
                     ((append (dive (1- n) (car w))
                               (dive n (cdr w)))))))
      (dive n w))))

> (dive-atoms 0 '(a (b)))
(A)
> (dive-atoms 1 '(a (b)))
(B)

```

Решение 8.36.6

```

(defun dive-atoms (n w)
  (cond ((null w) nil)
        ((= n 0) (labels ((atoms (w)
                              (cond ((null w) nil)
                                    ((atom (car w))
                                     (cons (car w) (atoms (cdr w))))
                                    ((atoms (cdr w))))))
            (atoms w)))
        ((atom (car w)) (dive-atoms n (cdr w)))
        ((append (dive-atoms (1- n) (car w))
                  (dive-atoms n (cdr w))))))

```

```
(dive-atoms n (cdr w))))))
```

```
> (dive-atoms 0 '(a (b)))
(A)
> (dive-atoms 1 '(a (b)))
(B)
```

Решение 8.36.7

```
(defun dive-atoms (n w)
  (cond ((null w) nil)
        ((= n 0) (remove-if #'listp w))
        ((atom (car w)) (dive-atoms n (cdr w)))
        ((nconc (dive-atoms (1- n) (car w))
                  (dive-atoms n (cdr w))))))

> (dive-atoms 0 '(a (b)))
(A)
> (dive-atoms 1 '(a (b)))
(B)
```

Задача 8.37 member-down.lisp

Определить функцию (f a w), где a – s-выражение, а w – список чисел, которая уменьшает числа списка w на единицу, если число a является элементом списка w, и возвращает исходный список в противном случае.

Решение 8.37.1

```
(defun member-down (a x) (if (member a x) (mapcar #'1- x) x))

> (member-down 2 '(1 2 3 4 5 6))
(0 1 2 3 4 5)
```

Задача 8.38 expt-elms.lisp

Определить функцию возведения в квадрат списка чисел.

Решение 8.38.1

```
(defun expt-elms (w f)
  (when w (cons (funcall f (car w)) (expt-elms (cdr w) f))))

(defun x^2 (x) (* x x))

> (expt-elms '(1 2 3) #'x^2)
(1 4 9)
```

Решение 8.38.2

```
(defun expt-elms (w)
```

```
(when w (cons (expt (car w) 2) (expt-elms (cdr w)))))

> (expt-elms '(1 2 3))
(1 4 9)
```

Решение 8.38.3

```
(defun expt-elms (w) (mapcar #'(lambda (a) (expt a 2)) w))

> (expt-elms '(1 2 3))
(1 4 9)
```

Задача 8.39 *swing-elms.lisp*

Написать программу, которая позволит увеличить значение всех положительных элементов одноуровневого числового списка на заданное число и уменьшить значения всех отрицательных элементов того же массива на то же число.

Решение 8.39.1

```
(defun swing-elms (w n &aux (a (car w)))
  (cond ((null w) nil)
        ((plusp a) (cons (+ a n) (swing-elms (cdr w) n)))
        ((minusp a) (cons (- a n) (swing-elms (cdr w) n)))
        ((cons a (swing-elms (cdr w) n)))))

> (swing-elms '(-1 0 1) 1)
(-2 0 2)
```

Решение 8.39.2

```
(defun swing-elms (w n &aux (a (car w)))
  (when w (cons (cond ((zerop a) a)
                    ((plusp a) (+ a n))
                    ((- a n)))
                (swing-elms (cdr w) n))))

> (swing-elms '(-1 0 1) 1)
(-2 0 2)
```

Задача 8.40 *knapsack.lisp*

Дан упорядоченный по убыванию список целых положительных чисел и некоторое число. Необходимо найти последовательности списка, сумма элементов которых равна этому числу, т.н. задача о ранце - название своё получила от максимизационной задачи укладки как можно большего числа ценных вещей в рюкзак при условии, что общий объём (или вес) всех предметов, способных поместиться в рюкзак, ограничен.

Решение 8.40.1


```

(defun knapsack (w n)
  (mapcar #'reverse (remove-duplicates (knap w n) :test #'equal)))

(defun knap (w n &optional acc ac
  &aux (a (car w)) (d (cdr w)) (z (apply #'+ ac)))
  (cond ((null w) acc)
        ((= (+ a z) n) (knap d n (cons (cons a ac) acc) ac))
        ((knap d n (knap d n acc ac)
          (if (< (+ a z) n) (cons a ac) ac)))))

> (knapsack '(9 8 7 6 5 4 3 2 1) 13)
((9 3 1) (9 4) (8 4 1) (8 3 2) (8 5) (7 5 1) (7 4 2) (7 3 2 1) (7 6)
(6 5 2) (6 4 2 1) (6 4 3) (5 4 3 1))

```

Решение 8.40.2

```

(defun knapsack (w n &optional acc ac
  &aux (a (car w)) (d (cdr w)) (b (cons a ac))
  (z (apply #'+ ac)))
  (cond ((null w) acc)
        ((= (+ a z) n)
         (knapsack d n (if (member (reverse b) acc
                                     :test #'equal)
                           acc
                           (cons (reverse b) acc))
          ac))
        ((knapsack d n (knapsack d n acc ac)
          (if (< (+ a z) n) b ac)))))

> (knapsack '(9 8 7 6 5 4 3 2 1) 13)
((9 3 1) (9 4) (8 4 1) (8 3 2) (8 5) (7 5 1) (7 4 2) (7 3 2 1) (7 6)
(6 5 2) (6 4 2 1) (6 4 3) (5 4 3 1))

```

Решение 8.40.3

```

(defun knapsack (w n &optional acc ac
  &aux (a (car w)) (d (cdr w)) (b (cons a ac))
  (z (apply #'+ ac)))
  (cond ((null w) acc)
        ((= (+ a z) n)
         (knapsack d n (let ((c (reverse b)))
                          (if (member c acc :test #'equal)
                              acc
                              (cons c acc))
                          ac))
        ((knapsack d n (knapsack d n acc ac)
          (if (< (+ a z) n) b ac)))))

> (knapsack '(9 8 7 6 5 4 3 2 1) 13)
((9 3 1) (9 4) (8 4 1) (8 3 2) (8 5) (7 5 1) (7 4 2) (7 3 2 1) (7 6)
(6 5 2) (6 4 2 1) (6 4 3) (5 4 3 1))

```

Решение 8.40.4

```
(defun knapsack (w n &optional acc ac
  &aux (a (car w)) (d (cdr w)) (b (cons a ac))
  (z (apply #' + ac)))
  (if w
    (if (= (+ a z) n)
      (knapsack d n (let ((c (reverse b)))
        (if (member c acc :test #'equal)
            acc
            (cons c acc))))
      ac)
    (knapsack d n (knapsack d n acc ac)
      (if (< (+ a z) n) b ac)))
  acc))

> (knapsack '(9 8 7 6 5 4 3 2 1) 13)
((9 3 1) (9 4) (8 4 1) (8 3 2) (8 5) (7 5 1) (7 4 2) (7 3 2 1) (7 6)
(6 5 2) (6 4 2 1) (6 4 3) (5 4 3 1))
```

Решение 8.40.5

```
(defun knapsack (w n &optional acc ac
  &aux (a (car w)) (d (cdr w)) (b (cons a ac))
  (z (apply #' + ac)))
  (if w (knapsack d n (if (= (+ a z) n)
    (let ((c (reverse b)))
      (if (member c acc :test #'equal)
          acc
          (cons c acc)))
    (knapsack d n acc ac))
    (if (< (+ a z) n) b ac))
  acc))

> (knapsack '(9 8 7 6 5 4 3 2 1) 13)
((9 3 1) (9 4) (8 4 1) (8 3 2) (8 5) (7 5 1) (7 4 2) (7 3 2 1) (7 6)
(6 5 2) (6 4 2 1) (6 4 3) (5 4 3 1))
```

Решение 8.40.6

```
(defun knapsack (w n &optional acc ac
  &aux (a (car w)) (d (cdr w)) (b (cons a ac))
  (z (apply #' + ac)))
  (if w (knapsack d n (if (= (+ a z) n)
    (adjoin (reverse b) acc :test #'equal)
    (knapsack d n acc ac))
    (if (< (+ a z) n) b ac))
  acc))

> (knapsack '(9 8 7 6 5 4 3 2 1) 13)
((9 3 1) (9 4) (8 4 1) (8 3 2) (8 5) (7 5 1) (7 4 2) (7 3 2 1) (7 6)
(6 5 2) (6 4 2 1) (6 4 3) (5 4 3 1))
```

Решение 8.40.7

```
(defun knapsack (w n &optional acc ac
                &aux (a (car w)) (d (cdr w)) (b (cons a ac))
                (z (apply #'+ ac)))
  (cond ((null w) acc)
        ((= (+ a z) n)
         (knapsack d n (adjoin (reverse b) acc :test #'equal) ac))
        ((knapsack d n (knapsack d n acc ac)
                     (if (< (+ a z) n) b ac)))))

> (knapsack '(9 8 7 6 5 4 3 2 1) 13)
((9 3 1) (9 4) (8 4 1) (8 3 2) (8 5) (7 5 1) (7 4 2) (7 3 2 1) (7 6)
(6 5 2) (6 4 2 1) (6 4 3) (5 4 3 1))
```

Решение 8.40.8

```
(defun knapsack (w n &optional acc ac
                &aux (a (car w)) (d (cdr w)) (b (cons a ac)))
  (cond ((null w) acc)
        ((= (apply #'+ b) n)
         (knapsack d n (adjoin (reverse b) acc :test #'equal) ac))
        ((knapsack d n (knapsack d n acc ac)
                     (if (< (apply #'+ b) n) b ac)))))

> (knapsack '(9 8 7 6 5 4 3 2 1) 13)
((9 3 1) (9 4) (8 4 1) (8 3 2) (8 5) (7 5 1) (7 4 2) (7 3 2 1) (7 6)
(6 5 2) (6 4 2 1) (6 4 3) (5 4 3 1))
```

Решение 8.40.9

```
(defun knapsack (w n &optional acc ac
                &aux (a (car w)) (d (cdr w)) (b (cons a ac)))
  (if w
      (knapsack d
                 n
                 (if (= (apply #'+ b) n)
                     (adjoin (reverse b) acc :test #'equal)
                     (knapsack d n acc ac))
                 (if (< (apply #'+ b) n) b ac))
      acc))

> (knapsack '(9 8 7 6 5 4 3 2 1) 13)
((9 3 1) (9 4) (8 4 1) (8 3 2) (8 5) (7 5 1) (7 4 2) (7 3 2 1) (7 6)
(6 5 2) (6 4 2 1) (6 4 3) (5 4 3 1))
```

Решение 8.40.10

```
(defun knapsack (w n &optional acc ac
                &aux (a (car w)) (d (cdr w)) (b (cons a ac)))
  (if w
      (let ((z (apply #'+ b)))
        (knapsack d
```

```

      n
      (if (= z n)
          (adjoin (reverse b) acc :test #'equal)
          (knapsack d n acc ac))
      (if (< z n) b ac)))
  acc))

> (knapsack '(9 8 7 6 5 4 3 2 1) 13)
((9 3 1) (9 4) (8 4 1) (8 3 2) (8 5) (7 5 1) (7 4 2) (7 3 2 1) (7 6)
(6 5 2) (6 4 2 1) (6 4 3) (5 4 3 1))

```

Задача 8.41 *insert-atom.lisp*

Реализовать функцию (insert atom list), возвращающую список list, в котором в начало каждого подсписка добавлен атом atom.

Пример: (insert-atom 'a '((b c) (b (c d)) (c d) nil))
 ((A B C) (A B (A C D)) (A C D) (A))

Решение 8.41.1

```

(defun insert-atom (a w)
  (mapcar #'(lambda (e) (if (atom e)
                            (if e e (list a))
                            (cons a (insert-atom a e)))) w))

> (insert-atom 'a '((b c) (b (c d)) (c d) nil))
((A B C) (A B (A C D)) (A C D) (A))

```

Решение 8.41.2

```

(defun insert-atom (a w)
  (mapcar #'(lambda (e) (if (listp e)
                            (cons a (insert-atom a e))
                            e))
          w))

> (insert-atom 'a '((b c) (b (c d)) (c d) nil))
((A B C) (A B (A C D)) (A C D) (A))

```

Решение 8.41.3

```

(defun insert-atom (a w)
  (cond ((null w) nil)
        ((listp (car w))
         (cons (cons a (car w)) (insert-atom a (cdr w))))
        ((cons (car w) (insert-atom a (cdr w))))))

> (insert-atom 'a '((b c) (b (c d)) (c d) nil))
((A B C) (A B (A C D)) (A C D) (A))

```

Задача 8.42 purgatory.lisp

Удалите из списка перед каждым вхождением n один элемент, если такой имеется и отличен от n .

Решение 8.42.1

```
(defun purgatory (n w)
  (cond ((null w) nil)
        ((and (eq (cadr w) n) (not (eq (car w) (cadr w))))
         (purgatory n (cdr w)))
        ((cons (car w) (purgatory n (cdr w))))))

> (purgatory 1 '(1 2 1 1 3))
(1 1 1 3)
```

Задача 8.43 not-dictionary.lisp

Дано предложение и словарь. Определить функцию, возвращающую слова предложения, которые не входят в словарь.

Решение 8.43.1

```
(defun not-dictionary (s v)
  (remove-duplicates
   (loop for a in
         (read-from-string
          (concatenate
           'string "("
           (delete-if-not #'(lambda (x)
                               (or (alpha-char-p x)
                                   (equal x #\space)
                                   (equal x #\(-)))
                             s) ")"))
         unless (member a v) collect a)))

> (not-dictionary "He has plundered our seas, ravaged our coasts,
burnt our towns, and destroyed the lives of our people." '(plundered
coasts towns destroyed lives people))
(HE HAS SEAS RAVAGED BURNT AND THE OF OUR)
```

Задача 8.44 zeka.lisp

Определить функцию $(f\ n\ m)$, где $n < m$, которая сначала выводит строку чисел без пробелов $n, n+1, n+2, \dots, m$, а затем на другой строке выводит значение в виде списка чисел $(n\ n+1\ n+2, \dots, m)$.

Решение 8.44.1

```
(defun zeka (n m)
  (mapcar #'(princ (loop for a from n to m collect a)))
```

```
> (zeka 1 5)
12345
(1 2 3 4 5)
```

Решение 8.44.1

```
(defun zeka (n m)
  (loop for a from n to m do (princ a) collect a))

> (zeka 1 5)
12345
(1 2 3 4 5)
```

Задача 8.45 cska.lisp

Lisp has pretty much the simplest syntax while being perhaps the most extensible language and treating data like programs and programs like data. All this MEANS that you can write very powerful programs in Lisp much easier than in most other languages. It also means that Lisp supports pretty much any "style" of programming you can imagine without having the go to extreme measures to make a new style "fit". Object Oriented, Functional, Declarative, Logic programming, Aspect programming, Agents and on an one. When each of these became popular the Lisp programmers were able to IMMEDIATELY use the new or newly popular concepts without having to first alter the language standard.

Herb Martin

Определить функцию (f n m), где $n < m$, которая сначала выводит числа в колонку $n, n+1, n+2, \dots, m$, а затем с новой строки выводит значение в виде списка чисел $(n \ n+1 \ n+2 \dots m)$.

Решение 8.45.1

```
(defun cska (n m)
  (loop for a from n to m
        do (format t "~a~%" a)
        collect a))

> (cska 1 5)
1
2
3
4
5
```

```
(1 2 3 4 5)
```

Решение 8.45.1

```
(defun cska (n m)
  (loop for a from n to m do (print a) collect a))

> (cska 1 5)
1
2
3
4
5
(1 2 3 4 5)
```

Задача 8.46 inserta.lisp

Используя функцию DEFUN определить функцию *f* (*a w*), где *a* - *s*-выражение, а *w* - список, состоящий из 2-3 элементов, которая вставляет *A* между первым и вторым элементом списка *w*, если третий элемент равен нулю. В остальных случаях возвращает исходный список *w*.

Решение 8.46.1

```
(defun inserta (a w)
  (if (zerop (caddr w)) (cons (car w) (cons a (cdr w))) w))

> (inserta 'v '(1 2 0))
(1 v 2 0)
```

Задача 8.47 dump-levels.lisp

Написать функцию, удаляющую из исходного списка заданное число под-списков заданной глубины.

Решение 8.47.1

```
(defun dump-levels (w n m)
  (labels ((next (w n)
    (cond ((null w) nil)
          ((atom (car w)) (cons (car w) (next (cdr w) n)))
          ((and (= n 0) (plusp m))
           (setf m (1- m)) (next (cdr w) n))
          ((cons (next (car w) (1- n)) (next (cdr w) n))))))
    (next w n)))

> (dump-levels '(5 (4 (6)) (1 7) (8 (3 (5)))) 0 1)
(5 (1 7) (8 (3 (5))))
> (dump-levels '(5 (4 (6)) (1 7) (8 (3 (5)))) 1 1)
(5 (4) (1 7) (8 (3 (5))))
```

Задача 8.48 keys+.lisp

Написать функцию, выбирающую из ассоциативного списка все значения, которые больше введенного числа. Собрать в список их ключи.

Решение 8.48.1

```
(defun keys+ (w n)
  (mapcar #'cdr (remove-if #'(lambda (a) (<= a n)) w :key #'cdr)))

> (keys+ '((a . 1) (b . 2) (c . 3) (d . 4) (e . 5) (f . 6) (g . 7)) 4)
(5 6 7)
```

Решение 8.48.2

```
(defun keys+ (w n)
  (mapcar #'cdr
    (remove-if-not #'(lambda (a) (> a n))
      w
      :key #'cdr)))

> (keys+ '((a . 1) (b . 2) (c . 3) (d . 4) (e . 5) (f . 6) (g . 7)) 4)
(5 6 7)
```

Задача 8.49 drop-n.lisp

Удалить из списка n-й элемент.

Решение 8.49.1

```
(defun drop-n (n w)
  (cond ((null w) nil)
        ((= n 2) (cons (car w) (cddr w)))
        ((cons (car w) (drop-n (1- n) (cdr w))))))

> (drop-n 14 '(1 2 3 4 5 6 7 8 9 10 11 12 13))
(1 2 3 4 5 6 7 8 9 10 11 12 13)
> (drop-n 14 '(1 2 3 4 5 6 7 8 9 10 11 12 13 14))
(1 2 3 4 5 6 7 8 9 10 11 12 13)
R> (drop-n 14 '(1 2 3 4 5 6 7 8 9 10 11 12 13 14 15))
(1 2 3 4 5 6 7 8 9 10 11 12 13 15)
```

Задача 8.50 with-pairs.lisp

Дан одноуровневый числовой список с четным числом элементов. Определить функционал, который применяет функцию последовательно к каждому двум соседним элементам списка.

Решение 8.50.1

```
(defun with-pairs (f w)
```



```

    (when w (cons (funcall f (car w) (cadr w))
                  (with-pairs f (cddr w)))))

> (with-pairs #'list '(1 2 3 4 5 6))
((1 2) (3 4) (5 6))
> (with-pairs #'+' '(1 2 3 4 5 6))
(3 7 11)
> (with-pairs #'- '(1 2 3 4 5 6))
(-1 -1 -1)
> (with-pairs #'* '(1 2 3 4 5 6))
(2 12 30)
> (with-pairs #'expt '(1 2 3 4 5 6))
(1 81 15625)

```

Решение 8.50.2

```

(defun with-pairs (f w)
  (loop for (a b) on w by #'cddr
        collect (funcall f a b)))

> (with-pairs #'list '(1 2 3 4 5 6))
((1 2) (3 4) (5 6))
> (with-pairs #'+' '(1 2 3 4 5 6))
(3 7 11)
> (with-pairs #'- '(1 2 3 4 5 6))
(-1 -1 -1)
> (with-pairs #'* '(1 2 3 4 5 6))
(2 12 30)
> (with-pairs #'expt '(1 2 3 4 5 6))
(1 81 15625)

```

Задача 8.51 drop-zero-rem-a-n.lisp

You become what you do. How and what you become depends on environmental influence so you become who you hang around. Raise the standard your peers must meet and you'll raise your expectations of yourself. If your environment is not making you better, change it.

Mark Twight

Дан одноуровневый числовой список с четным числом элементов. Определить функционал, который применяет функцию последовательно к каждому двум соседним элементам списка.

Решение 8.51.1

```

(defun drop-zero-rem-a-n (w n &aux (a (car w)))
  (cond ((null w) nil)
        ((zerop (rem a n)) (drop-zero-rem-a-n (cdr w) n))
        ((cons a (drop-zero-rem-a-n (cdr w) n)))))

```

```
> (drop-zero-rem-a-n '(1 2 3 4 5 6 7 8 9 10) 3)
(1 2 4 5 7 8 10)
```

Решение 8.51.2

```
(defun drop-zero-rem-a-n (w n &aux (a (car w)))
  (when w (if (zerop (rem a n))
              (drop-zero-rem-a-n (cdr w) n)
              (cons a (drop-zero-rem-a-n (cdr w) n)))))
```

```
> (drop-zero-rem-a-n '(1 2 3 4 5 6 7 8 9 10) 3)
(1 2 4 5 7 8 10)
```

Решение 8.51.3

```
(defun drop-zero-rem-a-n (w n)
  (loop for a in w
        unless (zerop (rem a n))
        collect a))
```

```
> (drop-zero-rem-a-n '(1 2 3 4 5 6 7 8 9 10) 3)
(1 2 4 5 7 8 10)
```

Решение 8.51.4

```
(defun drop-zero-rem-a-n (w n)
  (remove-if #'(lambda (a) (zerop (rem a n))) w))
```

```
> (drop-zero-rem-a-n '(1 2 3 4 5 6 7 8 9 10) 3)
(1 2 4 5 7 8 10)
```

Задача 8.52 *del-upto-a.lisp*

Определить функцию, удаляющую начало списка до заданного элемента (включительно).

Решение 8.52.1

```
(defun del-upto-a (a w)
  (cond ((null w) nil)
        ((equal a (car w)) (cdr w))
        ((del-upto-a a (cdr w)))))
```

```
> (del-upto-a 'a '((a b) c a b c))
(B C)
```

Решение 8.52.2

```
(defun del-upto-a (a w)
  (cond ((null w) nil)
        ((equal a (car w)) (cdr w))
```

```

      ((del-upto-a a (cdr w))))))
> (del-upto-a 'a '((a b) c a b c))
(B C)

```

Задача 8.53 *n-elm.lisp*

Создать список, в котором каждый из элементов исходного списка повторяется *n* раз.

Решение 8.53.1

```

(defun n-elm (n w &optional (m n))
  (cond ((null w) nil)
        ((= m 1) (cons (car w) (n-elm n (cdr w) n)))
        (t (cons (car w) (n-elm n w (1- m))))))
> (n-elm 3 '(a b c))
(A A A B B B C C C)

```

Задача 8.54 *_every.lisp*

Определите функциональный предикат, который истинен в том и только в том случае, когда являющийся функциональным аргументом предикат истинен для всех элементов списка.

Решение 8.54.1

```

(defun _every (p w)
  (cond ((null w) t)
        ((funcall p (car w)) (_every p (cdr w)))
        (t nil)))
> (_every #'numberp '(1 2 3))
T
> (_every #'numberp '(1 2 a))
NIL
> (_every #'numberp '())
T

```

Решение 8.54.2

```

(defun _every (p w)
  (if w (when (funcall p (car w)) (_every p (cdr w))) t))
> (_every #'numberp '(1 2 3))
T
> (_every #'numberp '(1 2 a))
NIL
> (_every #'numberp '())
T

```

Решение 8.54.3

```
CL-USER> (defpackage :good (:use :common-lisp))
#<PACKAGE GOOD>
CL-USER> (in-package :good)
#<PACKAGE GOOD>
GOOD> (defun every (p w)
  (if w (when (funcall p (car w)) (every p (cdr w))) t))
EVERY
GOOD> (every #'numberp '(1 2 3))
T
```

Задача 8.55 *sum-n-elms.lisp*

Найти сумму первых n положительных элементов одномерного числового массива.

Решение 8.55.1

```
(defun sum-n-elms (w n)
  (cond ((zerop n) 0)
        ((plusp (car w)) (+ (car w) (sum-n-elms (cdr w) (1- n))))
        (t (sum-n-elms (cdr w) n))))

> (sum-n-elms '(-10 -10 1 2 3 4) 2)
3
> (sum-n-elms '(-10 -10 1 2 3 4) 4)
10
```

Решение 8.55.1 (автор - nullxdth, www.cyberforum.ru)

```
(defun sum-n-elms (w n)
  (reduce #'+ (remove-if-not #'plusp w) :end n))

> (sum-n-elms '(-10 -10 1 2 3 4) 2)
3
> (sum-n-elms '(-10 -10 1 2 3 4) 4)
10
```

Задача 8.56 *cut-n-elms.lisp*

Напишите функцию, "обрывающую" список, если он состоит более чем из n элементов.

Решение 8.56.1

```
(defun cut-n-elms (n w)
  (cond ((or (zerop n) (null w)) nil)
        (t (cons (car w) (cut-n-elms (1- n) (cdr w))))))
```

```
> (cut-n-elms 4 '(1 -2 3 -4 -7 4 5))
(1 -2 3 -4)
```

Решение 8.56.2

```
(defun cut-n-elms (n w)
  (when (and w (> n 0))
    (cons (car w) (cut-n-elms (1- n) (cdr w)))))

> (cut-n-elms 4 '(1 -2 3 -4 -7 4 5))
(1 -2 3 -4)
```

Решение 8.56.3

```
(defun cut-n-elms (n w)
  (when (and (plusp n) w)
    (cons (car w) (cut-n-elms (1- n) (cdr w)))))

> (cut-n-elms 4 '(1 -2 3 -4 -7 4 5))
(1 -2 3 -4)
```

Задача 8.57 drop<n.lisp

Определите функцию, зависящую от двух аргументов *u* и *n*, которая по данному списку строит список его элементов, встречающихся в нем не менее *n* раз.

Решение 8.57.1

```
(defun drop<n (w n)
  (remove-duplicates
   (remove-if #'(lambda (a) (< (count a w) n)) w)))

> (drop<n '(40 2 8 9 4 2 8 9 4 1 4 1 2 8 7 9 7) 2)
(4 1 2 8 9 7)
```

Решение 8.57.2

```
(defun drop<n (w n &aux (v (remove-duplicates w)))
  (remove-if #'(lambda (a) (< (count a w) n)) v))

> (drop<n '(40 2 8 9 4 2 8 9 4 1 4 1 2 8 7 9 7) 2)
(4 1 2 8 9 7)
```

Задача 8.58 streak.lisp

Из произвольного списка выделить серии из *n* и более стоящих подряд одинаковых элементов.

Решение 8.58.1

```
(defun streak (w n &optional (b (car w)) ac &aux (a (car w)))
  (cond ((null w) (when (>= (length ac) n) (list ac)))
        ((eql a b) (streak (cdr w) n b (cons b ac)))
        ((< (length ac) n) (streak (cdr w) n a (list a)))
        ((cons ac (streak (cdr w) n a (list a))))))
```

```
> (streak '(1 2 2 2 3 4 5 5 5 6 7 7 7 7 8) 2)
((2 2 2) (5 5 5) (7 7 7 7 7))
> (streak '(1 2 2 2 3 4 5 5 5 6 7 7 7 7 8) 4)
((7 7 7 7 7))
```

Решение 8.58.2

```
(defun streak (w n &optional (b (car w)) ac
                &aux (a (car w)) (p (>= (length ac) n)))
  (cond ((null w) (when p (list ac)))
        ((eql a b) (streak (cdr w) n b (cons b ac)))
        (p (cons ac (streak (cdr w) n a (list a))))
        ((streak (cdr w) n a (list a)))))
```

```
> (streak '(1 2 2 2 3 4 5 5 5 6 7 7 7 7 8) 2)
((2 2 2) (5 5 5) (7 7 7 7 7))
> (streak '(1 2 2 2 3 4 5 5 5 6 7 7 7 7 8) 4)
((7 7 7 7 7))
```

Задача 8.59 add-abs.lisp

Дано s-выражение, состоящее из положительных и отрицательных чисел. Определить функцию, прибавляющую n к модулю каждого элемента этого s-выражения на всех уровнях.

Решение 8.59.1

```
(defun add-abs (w n)
  (cond ((null w) nil)
        ((atom (car w)) (cons (if (< (car w) 0)
                                   (- (car w) n)
                                   (+ (car w) n))
                               (add-abs (cdr w) n))))
  (t (cons (add-abs (car w) n)
            (add-abs (cdr w) n)))))
```

```
> (add-abs '(1 -2 ((3 (-4)) -5) 6) 1)
(2 -3 ((4 (-5)) -6) 7)
```

Решение 8.59.2

```
(defun add-abs (w n &aux (a (car w)))
  (cond ((null w) nil)
        ((atom a) (cons (if (minusp a)
                              (- a n)
                              (+ a n))
                          (add-abs (cdr w) n)))))
```

```

      (add-abs (cdr w) n)))
  ((cons (add-abs a n) (add-abs (cdr w) n)))))

```

```

> (add-abs '(1 -2 ((3 (-4)) -5) 6) 1)
(2 -3 ((4 (-5)) -6) 7)

```

Решение 8.59.3

```

(defun add-abs (w n &aux (a (car w)))
  (when w (if (atom a)
              (cons (if (minusp a)
                        (- a n)
                        (+ a n))
                    (add-abs (cdr w) n))
              (cons (add-abs a n)
                    (add-abs (cdr w) n)))))

```

```

> (add-abs '(1 -2 ((3 (-4)) -5) 6) 1)
(2 -3 ((4 (-5)) -6) 7)

```

Решение 8.59.4

```

(defun add-abs (w n &aux (a (car w)))
  (when w (cons (if (atom a)
                    (if (minusp a)
                        (- a n)
                        (+ a n))
                    (add-abs a n))
                (add-abs (cdr w) n))))

```

```

> (add-abs '(1 -2 ((3 (-4)) -5) 6) 1)
(2 -3 ((4 (-5)) -6) 7)

```

Решение 8.59.5

```

(defun add-abs (w n)
  (cond ((listp w) (mapcar #'(lambda (a) (add-abs a n)) w))
        ((minusp w) (- w n))
        ((+ w n))))

```

```

> (add-abs '(1 -2 ((3 (-4)) -5) 6) 1)
(2 -3 ((4 (-5)) -6) 7)

```

Решение 8.59.6

```

(defun add-abs (w n)
  (cond ((listp w) (loop for a in w collect (add-abs a n)))
        ((minusp w) (- w n))
        ((+ w n))))

```

```

> (add-abs '(1 -2 ((3 (-4)) -5) 6) 1)
(2 -3 ((4 (-5)) -6) 7)

```

Решение 8.59.7

```
(defun add-abs (w n)
  (cond ((listp w) (mapcar #'(lambda (a) (add-abs a n)) w))
        ((eval `(, (if (minusp w) '- '+) ,w ,n)))))

> (add-abs '(1 -2 ((3 (-4)) -5) 6) 1)
(2 -3 ((4 (-5)) -6) 7)
```

Решение 8.59.8

```
(defun add-abs (w n)
  (cond ((listp w) (loop for a in w collect (add-abs a n)))
        ((eval `(, (if (minusp w) '- '+) ,w ,n)))))

> (add-abs '(1 -2 ((3 (-4)) -5) 6) 1)
(2 -3 ((4 (-5)) -6) 7)
```

Решение 8.59.9

```
(defun add-abs (w n)
  (if (listp w)
      (mapcar #'(lambda (a) (add-abs a n)) w)
      (eval `(, (if (minusp w) '- '+) ,w ,n))))

> (add-abs '(1 -2 ((3 (-4)) -5) 6) 1)
(2 -3 ((4 (-5)) -6) 7)
```

Решение 8.59.10

```
(defun add-abs (w n)
  (if (listp w)
      (loop for a in w collect (add-abs a n))
      (eval `(, (if (minusp w) '- '+) ,w ,n))))

> (add-abs '(1 -2 ((3 (-4)) -5) 6) 1)
(2 -3 ((4 (-5)) -6) 7)
```

Решение 8.59.11

```
(defun add-abs (w n)
  (when w (if (atom w)
              (if (minusp w)
                  (- w n)
                  (+ w n))
              (cons (add-abs (car w) n)
                    (add-abs (cdr w) n)))))

> (add-abs '(1 -2 ((3 (-4)) -5) 6) 1)
(2 -3 ((4 (-5)) -6) 7)
> (add-abs '(1 -2 ((3 (-4)) -5) 6) 2)
(3 -4 ((5 (-6)) -7) 8)
```


Задача 8.60 *slow-moving-n-numbers-sum.lisp*

Дан список из m чисел и натуральное число $n < m$. Для каждой группы из n элементов, которые находятся рядом, вычислить ее сумму. Написать функцию, которая выдает список из всех возможных сумм. Пример: (7 1 4 2 3), $n = 3$, результат (12 7 9).

Решение 8.60.1

```
(defun slow-moving-n-numbers-sum (w n)
  (cond ((null (nth (1- n) w)) nil)
        (t (cons (apply #'(subseq w 0 n))
                  (slow-moving-n-numbers-sum (cdr w) n))))))

> (slow-moving-n-numbers-sum '(7 1 4 2 3) 3)
(12 7 9)
```

Решение 8.60.2

```
(defun slow-moving-n-numbers-sum (w n)
  (when (nth (1- n) w)
    (cons (apply #'(subseq w 0 n))
          (slow-moving-n-numbers-sum (cdr w) n))))

> (slow-moving-n-numbers-sum '(7 1 4 2 3) 3)
(12 7 9)
```

Решение 8.60.3

```
(defun slow-moving-n-numbers-sum (w n)
  (loop for d on w
        when (nth (1- n) d)
        collect (apply #'(subseq d 0 n))))

> (slow-moving-n-numbers-sum '(7 1 4 2 3) 3)
(12 7 9)
```

Задача 8.61 *anarchy-nums.lisp*

Определить функцию, вычисляющую количество последовательностей n -поряд идущих элементов числового списка, в каждой из которых все элементы различны.

Решение 8.61.1

```
(defun anarchy (w)
  (= (length w)
     (length (remove-duplicates w))))

(defun anarchy-nums (w n)
  (cond ((null (cddr w)) 0)
        ((anarchy (subseq w 0 n))
```

```

      (+ 1 (anarchy-nums (cdr w) n)))
      ((anarchy-nums (cdr w) n)))

> (anarchy-nums '(1 1 3 5 4 5 2 3 2 2 1 1 1) 3)
4

```

Задача 8.62 *dropp.lisp*

Определить функцию, которая из исходного простого списка удаляет все атомы, обладающие заданным свойством.

Решение 8.62.1

```

(defun dropp (p w)
  (cond ((null w) nil)
        ((funcall p (car w)) (dropp p (cdr w)))
        (t (cons (car w) (dropp p (cdr w))))))

> (dropp #'oddp '(1 2 3 4))
(2 4)

```

Решение 8.62.2

```

(defun dropp (p w)
  (when w (if (funcall p (car w))
              (dropp p (cdr w))
              (cons (car w) (dropp p (cdr w))))))

> (dropp #'oddp '(1 2 3 4))
(2 4)

```

Решение 8.62.3

```

(defun dropp (p w)
  (loop for a in w unless (funcall p a) collect a))

> (dropp #'oddp '(1 2 3 4))
(2 4)

```

Задача 8.63 *complex-dropp.lisp*

Определить функцию, которая из исходного сложного списка удаляет все атомы, обладающие заданным свойством.

Решение 8.63.1

```

(defun complex-dropp (p w)
  (cond ((null w) nil)
        ((listp (car w)) (cons (complex-dropp p (car w))
                                (complex-dropp p (cdr w))))
        ((funcall p (car w)) (complex-dropp p (cdr w))))

```

```

      (t (cons (car w) (complex-dropp p (cdr w)))))

> (complex-dropp #'oddp '((1) (2 (((3)))) ((4))))
(NIL (2 ((NIL))) ((4)))

```

Решение 8.63.2

```

(defun complex-dropp (p w)
  (loop for a in w
        if (listp a) collect (complex-dropp p a)
        else unless (funcall p a) collect a))

> (complex-dropp #'oddp '((1) (2 (((3)))) ((4))))
(NIL (2 ((NIL))) ((4)))

```

Задача 8.64 overlap-a.lisp

Заменить повторяющиеся элементы матрицы на заданный элемент.

Решение 8.64.1

```

(defun overlap-a (w a &aux (v (reduce #'append w)))
  (loop for e in w collect (change-a e a v)))

(defun change-a (w a v)
  (loop for e in w
        if (> (count e v) 1) collect a
        else collect e))

> (overlap-a '((1 2 3) (4 5 6) (1 2 3)) 7)
((7 7 7) (4 5 6) (7 7 7))

```

Задача 8.65 aftershock.lisp

Определить функцию, которая по списку и атому возвращает множество всех объектов, непосредственно следующих за вхождениями атома в список.

Решение 8.65.1

```

(defun aftershock (w z)
  (remove-duplicates (next w z)))

(defun next (w z &aux (a (car w)) (d (cdr w)))
  (cond ((null d) nil)
        ((equalp a z) (cons (car d) (next d z)))
        ((next d z))))

> (aftershock '(a b a b a c a) 'a)
(B C)

```

Решение 8.65.2

```
(defun aftershock (w z &optional ac &aux (a (car w)) (d (cdr w)))
  (cond ((null d) (nreverse ac))
        ((or (not (equalp a z)) (member (car d) ac))
         (aftershock d z ac))
        ((aftershock d z (cons (car d) ac)))))

> (aftershock '(a b a b a c a) 'a)
(B C)
```

Решение 8.65.3

```
(defun aftershock (w z) (remove-duplicates (next w z)))

(defun next (w z)
  (loop for a in w
        for b in (cdr w)
        when (equalp a z) collect b))

> (aftershock '(a b a b a c a) 'a)
(B C)
```

Задача 8.66 grow.lisp

Определите функцию `(f w n)`, которая из списка создает новый список, прибавляя к каждому атому число `n`. Исходный список не предполагается одноуровневым.

Решение 8.66.1

```
(defun grow (w n)
  (mapcar #'
    (lambda (a)
      (if (atom a)
          (if (numberp a)
              (+ a n)
              a)
          (grow a n)))
    w))

> (grow '(1 a (2 b)) 10)
(11 A (12 B))
```

Задача 8.67 filter.lisp

Определите функцию `(filter p w)`, которая создает список из элементов списка `w`, удовлетворяющим предикату `p`.

Решение 8.67.1

```
(defun filter (p w &aux (a (car w)) (d (cdr w)))
  (cond ((null w) nil)
        ((funcall p a) (cons a (filter p d)))
        ((filter p d))))
```

```
> (filter #'evenp '(0 1 2 3))
(0 2)
```

Решение 8.67.2

```
(defun filter (p w &aux (a (car w)) (d (cdr w)))
  (when w (if (funcall p a) (cons a (filter p d)) (filter p d))))
```

```
> (filter #'evenp '(0 1 2 3))
(0 2)
```

Решение 8.67.3

```
(defun filter (p w &optional ac &aux (a (car w)) (d (cdr w)))
  (cond ((null w) (nreverse ac))
        ((funcall p a) (filter p d (cons a ac)))
        ((filter p d ac))))
```

```
> (filter #'evenp '(0 1 2 3))
(0 2)
```

Решение 8.67.4

```
(defun filter (p w &optional ac &aux (a (car w)) (d (cdr w)))
  (if w (filter p d (if (funcall p a) (cons a ac) ac))
      (nreverse ac)))
```

```
> (filter #'evenp '(0 1 2 3))
(0 2)
```

Решение 8.67.5

```
(defun filter (p w)
  (loop for a in w when (funcall p a) collect a))
```

```
> (filter #'evenp '(0 1 2 3))
(0 2)
```

Решение 8.67.6

```
(defun filter (p w)
  (remove-if-not p w))
```

```
> (filter #'evenp '(0 1 2 3))
(0 2)
```

Задача 8.68 del-n.lisp

Определите функцию удаления из исходного списка первых n элементов.

Решение 8.68.1

```
(defun del-n (n w)
  (cond ((zerop n) w)
        (t (del-n (1- n) (cdr w)))))
```

```
> (del-n 3 '(1 2 3 4 5))
(4 5)
```

Решение 8.68.2

```
(defun del-n (n w)
  (loop for a downfrom n for c on w
        when (zerop a) return c))
```

```
> (del-n 3 '(1 2 3 4 5))
(4 5)
```

Решение 8.68.3

```
(defun del-n (n w)
  (loop for a downfrom n for c on w
        when (zerop a) return c))
```

```
> (del-n 3 '(1 2 3 4 5))
(4 5)
```

Решение 8.68.4

```
(defun del-n (n w) (subseq w n))
```

```
> (del-n 3 '(1 2 3 4 5))
(4 5)
```

Решение 8.68.5

```
(defun del (n w) (nthcdr n w))
```

```
> (del-n 3 '(1 2 3 4 5))
(4 5)
```

Задача 8.69 symb-to-depth.lisp

Определите функцию заменяющую символ на число, равное глубине вложения этого символа в список.

Решение 8.69.1

```

(defun symb-to-depth (z w)
  (reverse-list (depth z w)))

(defun depth (z
              w
              &optional (n 0) ac
              &aux (a (car w)) (d (cdr w)))
  (cond ((null w) ac)
        ((listp a) (depth z
                           d
                           n
                           (cons (depth z a (1+ n) nil) ac)))
        ((eq a z) (depth z d n (cons n ac)))
        ((depth z d n (cons a ac)))))

(defun reverse-list (w)
  (cond ((atom w) w)
        ((rearrange w nil))))

(defun rearrange (v acc)
  (cond ((null v) acc)
        ((rearrange (cdr v)
                     (cons (reverse-list (car v))
                           acc)))))

> (symb-to-depth 'a '((a b) a (c (a (a d)))))
(1 B) 0 (C (2 (3 D)))

```

Решение 8.69.2

```

(defun sumb-to-depth (z
                      w
                      &optional (n 0) ac
                      &aux (a (car w)) (d (cdr w)))
  (cond ((null w) ac)
        ((listp a) (sumb-to-depth z
                                   d
                                   n
                                   (cons (sumb-to-depth z a (1+ n) nil)
                                         ac)))
        ((eq a z) (sumb-to-depth z d n (reverse (cons n ac))))
        ((sumb-to-depth z d n (reverse (cons a ac)))))

> (symb-to-depth 'a '((a b) a (c (a (a d)))))
(1 B) 0 (C (2 (3 D)))

```

Решение 8.69.3

```

(defun sumb-to-depth (z
                      w
                      &optional (n 0) ac
                      &aux (a (car w)) (d (cdr w)))
  (cond ((null w) ac)

```

```

((listp a) (sumb-to-depth z
                        d
                        n
                        (cons (sumb-to-depth z a (1+ n) nil)
                              ac)))
((sumb-to-depth z d n (reverse (cons (if (eq a z) n a)
                                       ac)))))

```

```

> (symb-to-depth 'a '((a b) a (c (a (a d)))))
((1 B) 0 (C (2 (3 D))))

```

Решение 8.69.4

```

(defun sumb-to-depth (z w &optional (n 0) ac &aux (a (car w)))
  (if w
      (sumb-to-depth z
                     (cdr w)
                     n
                     (if (listp a)
                         (cons (sumb-to-depth z a (1+ n) nil) ac)
                         (reverse (cons (if (eq a z) n a)
                                         ac)))))
      ac))

```

```

> (symb-to-depth 'a '((a b) a (c (a (a d)))))
((1 B) 0 (C (2 (3 D))))

```

Решение 8.69.5

```

(defun symb-to-depth (z
                     w
                     &optional (n 0)
                     &aux (a (car w)) (d (cdr w)))
  (cond ((null w) nil)
        ((listp a) (cons (symb-to-depth z a (1+ n))
                          (symb-to-depth z d n)))
        ((eq a z) (cons n (symb-to-depth z d n)))
        ((cons a (symb-to-depth z d n)))))

```

```

> (symb-to-depth 'a '((a b) a (c (a (a d)))))
((1 B) 0 (C (2 (3 D))))

```

Решение 8.69.6

```

(defun symb-to-depth (z w &optional (n 0) &aux (a (car w)))
  (when w (cons (cond ((listp a) (symb-to-depth z a (1+ n)))
                    ((eq a z) n)
                    (a))
                (symb-to-depth z (cdr w) n))))

```

```

> (symb-to-depth 'a '((a b) a (c (a (a d)))))
((1 B) 0 (C (2 (3 D))))

```


Задача 8.70 rem-num-sq-rest.lisp

Определите функцию, которая удаляет из числового списка все совпадающие с заданным числовым атомом элементы и возвращает в качестве значения список из квадратов оставшихся элементов.

Решение 8.70.1

```
(defun rem-num-sq-rest (n w)
  (cond ((null w) nil)
        ((= (car w) n) (rem-num-sq-rest n (cdr w)))
        ((cons (expt (car w) 2)
                 (rem-num-sq-rest n (cdr w))))))

> (rem-num-sq-rest 1 '(1 2))
(4)
```

Решение 8.70.2

```
(defun rem-num-sq-rest (n w)
  (when w (if (= (car w) n)
              (rem-num-sq-rest n (cdr w))
              (cons (expt (car w) 2)
                     (rem-num-sq-rest n (cdr w))))))

> (rem-num-sq-rest 1 '(1 2))
(4)
```

Решение 8.70.3

```
(defun rem-num-sq-rest (n w &aux (a (car w)) (d (cdr w)))
  (when w (if (= a n)
              (rem-num-sq-rest n d)
              (cons (expt a 2)
                     (rem-num-sq-rest n d)))))

> (rem-num-sq-rest 1 '(1 2))
(4)
```

Решение 8.70.4

```
(defun rem-num-sq-rest (n w &aux (a (car w)) (d (cdr w)))
  (when w (if (= a n)
              (rem-num-sq-rest n d)
              (cons (* a a) (rem-num-sq-rest n d)))))

> (rem-num-sq-rest 1 '(1 2))
(4)
```

Решение 8.70.5

```
(defun rem-num-sq-rest (n w)
```

```
(loop for a in w unless (= a n) collect (* a a)))
```

```
> (rem-num-sq-rest 1 '(1 2))
(4)
```

Решение 8.70.6

```
(defun rem-num-sq-rest (n w)
  (mapcar #'(lambda (a) (expt a 2)) (remove n w)))
```

```
> (rem-num-sq-rest 1 '(1 2))
(4)
```

Решение 8.70.7

```
(defun rem-num-sq-rest (n w)
  (mapcar #'(lambda (a) (* a a)) (remove n w)))
```

```
> (rem-num-sq-rest 1 '(1 2))
(4)
```

Решение 8.70.8

```
(defun rem-num-sq-rest (n w &aux (z (remove n w)))
  (mapcar #'* z z))
```

```
> (rem-num-sq-rest 1 '(1 2))
(4)
```

Задача 8.71 shelf-life>.lisp

На аптечном складе хранятся лекарства. Сведения о лекарствах содержатся в специальной ведомости: наименование препарата, количество, цена, срок хранения (в месяцах). Выяснить, какие препараты имеют срок хранения более 3 месяцев.

Решение 8.71.1

```
(defstruct pharmacy name quantity price shelf-life)
```

```
(defun res (n)
  (when (> n 0)
    (cons (make-pharmacy :name (read)
                          :quantity (read)
                          :price (read)
                          :shelf-life (read))
          (res (1- n)))))
```

```
(defun shelf-life> (months w)
  (mapcan #'(lambda (a)
    (when (> (pharmacy-shelf-life a) months)
      (list (pharmacy-name a))))))
```

```

      w))

> (setf w (res 2))
aspirin
2
10
6
analgin
3
20
1
(#S(PHARMACY :NAME ASPIRIN :QUANTITY 2 :PRICE 10 :SHELF-LIFE 6)
 #S(PHARMACY :NAME ANALGIN :QUANTITY 3 :PRICE 20 :SHELF-LIFE 1))
> (shelf-life> 3 w)
(ASPIRIN)

```

Решение 8.71.2

```

(defstruct pharmacy name quantity price shelf-life)

(defun res (n)
  (when (> n 0)
    (cons (make-pharmacy :name (read)
                          :quantity (read)
                          :price (read)
                          :shelf-life (read))
          (res (1- n)))))

(defun shelf-life> (months w)
  (mapcar
   #'(lambda (a) (pharmacy-name a))
   (remove-if-not
    #'(lambda (a) (> (pharmacy-shelf-life a) months))
    w)))

> (setf w (res 2))
aspirin
2
10
6
analgin
3
20
1
(#S(PHARMACY :NAME ASPIRIN :QUANTITY 2 :PRICE 10 :SHELF-LIFE 6)
 #S(PHARMACY :NAME ANALGIN :QUANTITY 3 :PRICE 20 :SHELF-LIFE 1))
> (shelf-life> 3 w)
(ASPIRIN)

```

Решение 8.71.3

```

(defstruct pharmacy name quantity price shelf-life)

```

```

(defun res (n)
  (when (> n 0)
    (cons (make-pharmacy :name (read)
                        :quantity (read)
                        :price (read)
                        :shelf-life (read))
          (res (1- n)))))

(defun shelf-life> (months w)
  (loop for a in w
        when (> (pharmacy-shelf-life a) months)
        collect (pharmacy-name a)))

> (setf w (res 2))
aspirin
2
10
6
analgin
3
20
1
(#S(PHARMACY :NAME ASPIRIN :QUANTITY 2 :PRICE 10 :SHELF-LIFE 6)
 #S(PHARMACY :NAME ANALGIN :QUANTITY 3 :PRICE 20 :SHELF-LIFE 1))
> (shelf-life> 3 w)
(ASPIRIN)

```

Задача 8.72 children-rhymes.lisp

Определить функцию, моделирующую "детскую считалочку". В кругу стоит n человек имеющих разные имена, в "считалочке" m слов. Функция должна возвращать последовательность выбывания игроков из круга.

Решение 8.72.1

```

(defun children-rhymes (w n &optional (m n) ac &aux (a (car w)))
  (cond ((null w) (when ac (children-rhymes (reverse ac) n m nil)))
        ((= n 1) (cons a (children-rhymes (cdr w) m m ac)))
        ((children-rhymes (cdr w) (1- n) m (cons a ac)))))

> (children-rhymes '(a b c d e f g) 3)
(C F B G E A D)
> (children-rhymes '("rob" "eva" "lee" "ann" "tom" "mik" "suz" "jes")
4)
("ann" "jes" "tom" "eva" "rob" "lee" "suz" "mik")
> (children-rhymes '("rob" "eva" "lee" "ann" "tom" "mik" "suz" "jes")
7)
("suz" "mik" "jes" "eva" "tom" "rob" "lee" "ann")

```

Решение 8.72.2

```

(defun children-rhymes (w n &optional (m n) ac)

```

```

(cond ((null w) (when ac (children-rhymes (reverse ac) n m nil)))
      ((= n 1) (cons (car w) (children-rhymes (cdr w) m m ac)))
      ((children-rhymes (cdr w) (1- n) m (cons (car w) ac)))))

> (children-rhymes ' (a b c d e f g) 3)
(C F B G E A D)
> (children-rhymes ' ("rob" "eva" "lee" "ann" "tom" "mik" "suz" "jes")
4)
("ann" "jes" "tom" "eva" "rob" "lee" "suz" "mik")
> (children-rhymes ' ("rob" "eva" "lee" "ann" "tom" "mik" "suz" "jes")
7)
("suz" "mik" "jes" "eva" "tom" "rob" "lee" "ann")

```

Задача 8.73 *bigger-than-this.lisp*

Определить функцию, возвращающую элементы списка большие заданного числа.

Решение 8.73.1

```

(defun bigger-than-this (w n)
  (cond ((null w) nil)
        ((> (car w) n)
         (cons (car w) (bigger-than-this (cdr w) n)))
        ((bigger-than-this (cdr w) n))))

> (bigger-than-this ' (1 2 3 1 2 3 4) 2)
(3 3 4)

```

Решение 8.73.2

```

(defun bigger-than-this (w n &optional ac)
  (cond ((null w) (reverse ac))
        ((> (car w) n)
         (bigger-than-this (cdr w) n (cons (car w) ac)))
        ((bigger-than-this (cdr w) n ac))))

> (bigger-than-this ' (1 2 3 1 2 3 4) 2)
(3 3 4)

```

Решение 8.73.3

```

(defun bigger-than-this (w n &optional ac &aux (a (car w)))
  (cond ((null w) (reverse ac))
        ((> a n) (bigger-than-this (cdr w) n (cons a ac)))
        ((bigger-than-this (cdr w) n ac))))

> (bigger-than-this ' (1 2 3 1 2 3 4) 2)
(3 3 4)

```

Решение 8.73.4

```
(defun bigger-than-this (w n &optional ac &aux (a (car w)))
  (if w
    (bigger-than-this (cdr w) n (if (> a n) (cons a ac) ac))
    (reverse ac)))

> (bigger-than-this '(1 2 3 1 2 3 4) 2)
(3 3 4)
```

Решение 8.73.5

```
(defun bigger-than-this (w n)
  (loop for a in w
        when (> a n) collect a))

> (bigger-than-this '(1 2 3 1 2 3 4) 2)
(3 3 4)
```

Решение 8.73.6

```
(defun bigger-than-this (w n)
  (remove-if-not #'(lambda (a) (> a n)) w))

> (bigger-than-this '(1 2 3 1 2 3 4) 2)
(3 3 4)
```

Задача 8.74 drop.lisp

Определить функцию, удаляющую элемент из списка по заданной позиции.

Решение 8.74.1

```
(defun drop (w p &optional ac)
  (cond ((null w) (nreverse ac))
        ((zerop p) (append (nreverse ac) (cdr w)))
        ((drop (cdr w) (1- p) (cons (car w) ac)))))

> (drop '(1 2 3 4 5 6 7) 0)
(2 3 4 5 6 7)
> (drop '(1 2 3 4 5 6 7) 3)
(1 2 3 5 6 7)
```

Решение 8.74.2

```
(defun drop (w p)
  (loop for a in w for b from 0
        unless (= b p) collect a))

> (drop '(1 2 3 4 5 6 7) 0)
(2 3 4 5 6 7)
> (drop '(1 2 3 4 5 6 7) 3)
(1 2 3 5 6 7)
```

Решение 8.74.3

```
(defun drop (w p)
  (if (and w (pluseq p))
      (cons (car w) (drop (cdr w) (1- p)))
      (cdr w)))

> (drop '(1 2 3 4 5 6 7) 0)
(2 3 4 5 6 7)
> (drop '(1 2 3 4 5 6 7) 3)
(1 2 3 5 6 7)
```

Решение 8.74.4

```
(defun drop (w p &optional ac)
  (if (and w (pluseq p))
      (drop (cdr w) (1- p) (cons (car w) ac))
      (append (nreverse ac) (cdr w))))

> (drop '(1 2 3 4 5 6 7) 0)
(2 3 4 5 6 7)
> (drop '(1 2 3 4 5 6 7) 3)
(1 2 3 5 6 7)
```

Задача 8.75 under.lisp

Из списка целых чисел сформировать список чисел, которые меньше заданного.

Решение 8.75.1

```
(defun under (w n &aux (a (car w)))
  (cond ((null w) nil)
        ((< a n) (cons a (under (cdr w) n)))
        ((under (cdr w) n))))

> (under '(10 2 30 4 50 6 70) 7)
(2 4 6)
```

Решение 8.75.2

```
(defun under (w n &aux (a (car w)))
  (when w (if (< a n)
              (cons a (under (cdr w) n))
              (under (cdr w) n))))

> (under '(10 2 30 4 50 6 70) 7)
(2 4 6)
```

Решение 8.75.3

```
(defun under (w n &optional ac &aux (a (car w)))
```

```
(cond ((null w) (nreverse ac))
      ((< a n) (under (cdr w) n (cons a ac)))
      ((under (cdr w) n ac))))
```

```
> (under '(10 2 30 4 50 6 70) 7)
(2 4 6)
```

Решение 8.75.4

```
(defun under (w n &optional ac &aux (a (car w)))
  (if w
      (under (cdr w) n (if (< a n)
                           (cons a ac)
                           ac))
      (nreverse ac)))
```

```
> (under '(10 2 30 4 50 6 70) 7)
(2 4 6)
```

Решение 8.75.5

```
(defun under (w n)
  (loop for a in w
        when (< a n) collect a))
```

```
> (under '(10 2 30 4 50 6 70) 7)
(2 4 6)
```

Решение 8.75.5

```
(defun under (w n)
  (remove-if-not #'(lambda (a) (< a n)) w))
```

```
> (under '(10 2 30 4 50 6 70) 7)
(2 4 6)
```

Задача 8.76 *mn.lisp*

Найти индекс n-го отрицательного вхождения.

Решение 8.76.1

```
(defun mn (w
           z
           &optional
           (f (and (>= (count-if #'minusp w) z)
                   (plussp z)
                   w)))
  (when f (let ((m (minusp (car w))))
    (if (and m (= z 1)) 0
        (1+ (mn (cdr w) (- z (if m 1 0)) t))))))
```


Решение 8.76.2

```
(defun mn (w z)
  (when (and (>= (count-if #'minusp w) z)
            (plusp z)
            w)
    (mn- w z)))

(defun mn- (w z &aux (m (minusp (car w))))
  (if (and m (= z 1))
      0
      (1+ (mn- (cdr w) (- z (if m 1 0))))))

> (mn '(3 -2 -4 -7 9) 2)
2
```

Задача 8.77 deep-drop.lisp

Most people who graduate with CS degrees don't understand the significance of Lisp. Lisp is the most important idea in computer science.

Alan Kay

Напишите функцию (f x w), которая возвращает список, получающийся из списка w после удаления всех вхождений атома x на всех уровнях, например, (f 1 '(1 (2 3 (1)) (3 1))) => ((2 3 nil) (3)).

Решение 8.77.1

```
(defun deep-drop (x w)
  (loop for a in w
        if (listp a) collect (deep-drop x a)
        else unless (equalp a x) collect a))

> (deep-drop 1 '(1 (2 3 (1)) (3 1)))
((2 3 NIL) (3))
> (deep-drop 1 '())
NIL
```

Решение 8.77.2

```
(defun deep-drop (e w &aux (a (car w)) (d (cdr w)))
  (when w (if (equal a e) (deep-drop e d)
              (cons (if (atom a) a (deep-drop e a))
                    (deep-drop e d)))))

> (deep-drop 1 '(1 (2 3 (1)) (3 1)))
((2 3 NIL) (3))
> (deep-drop 1 '())
NIL
```

Задача 8.78 split.lisp

Определить функцию, осуществляющую разделение исходного списка на два подсписка. В первый из них должно попасть указанное количество элементов с начала списка, во второй — оставшиеся элементы.

Решение 8.78.1

```
(defun split (w n)
  (when w (if (zerop n) (list nil w)
              (glue (car w) (split (cdr w) (1- n))))))

(defun glue (e r) (cons (cons e (car r)) (cdr r)))

> (f '(a b c d e f g) 3)
((A B C) (D E F G))
```

Задача 8.79 p-pos.lisp

Дан список чисел и условие, заданное функцией. Сформировать список, содержащий номера позиций элементов исходного списка, удовлетворяющих данному условию.

Решение 8.79.1

```
(defun p-pos (w p &optional (n 0))
  (cond ((null w) nil)
        ((funcall p (car w)) (cons n (p-pos (cdr w) p (1+ n))))
        ((p-pos (cdr w) p (1+ n)))))

> (p-pos '(-1 1 1 -1) #'plusp)
(1 2)
```

Решение 8.79.2

```
(defun p-pos (w p &optional (n 0))
  (when w (if (funcall p (car w)) (cons n (p-pos (cdr w) p (1+ n)))
              (p-pos (cdr w) p (1+ n)))))

> (p-pos '(-1 1 1 -1) #'plusp)
(1 2)
```

Решение 8.79.3

```
(defun p-pos (w p)
  (loop for a in w for n from 0
        when (funcall p a) collect n))

> (p-pos '(-1 1 1 -1) #'plusp)
(1 2)
```

Задача 8.80 unveil.lisp

Определить функцию поиска в ассоциированном списке *w* точечной пары, соответствующей ключу *a*.

Решение 8.80.1

```
(defun unveil (w a)
  (car (remove-if-not #'(lambda (e) (equal (car e) a)) w)))

> (unveil '((a . 1) (b . 2) (c . 3)) 'b)
(b . 2)
```

Решение 8.80.2

```
(defun unveil (w a)
  (find-if #'(lambda (e) (equal (car e) a)) w))

> (unveil '((a . 1) (b . 2) (c . 3)) 'b)
(b . 2)
```

Решение 8.80.3

```
(defun unveil (w a)
  (find a w :key #'car))

> (unveil '((a . 1) (b . 2) (c . 3)) 'b)
(b . 2)
```

Задача 8.81 drop-less.lisp

Удалить из списка все элементы, меньше заданного.

Решение 8.81.1

```
(defun drop-less (n w &optional ac)
  (cond ((null w) (nreverse ac))
        ((< (car w) n) (drop-less n (cdr w) ac))
        ((drop-less n (cdr w) (cons (car w) ac)))))

> (drop-less 10 '(7 8 9 10 11))
(10 11)
```

Решение 8.81.2

```
(defun drop-less (n w)
  (labels ((drop (n w ac)
            (cond ((null w) (nreverse ac))
                  ((< (car w) n) (drop n (cdr w) ac))
                  ((drop n (cdr w) (cons (car w) ac))))))
    (drop n w nil)))
```

1000

```
> (drop-less 10 '(7 8 9 10 11))  
(10 11)
```

Решение 8.81.3

```
(defun drop-less (n w)  
  (labels ((drop (n w ac)  
             (if w (drop n (cdr w)  
                          (if (< (car w) n) ac  
                              (cons (car w) ac)))  
             (nreverse ac))))  
    (drop n w nil)))
```

```
> (drop-less 10 '(7 8 9 10 11))  
(10 11)
```

Решение 8.81.4

```
(defun drop-less (n w)  
  (labels ((drop (n w ac &aux (a (car w)))  
             (if w (drop n (cdr w) (if (< a n) ac (cons a ac)))  
             (nreverse ac))))  
    (drop n w nil)))
```

```
> (drop-less 10 '(7 8 9 10 11))  
(10 11)
```

Решение 8.81.5

```
(defun drop-less (n w)  
  (loop for a in w unless (< a n) collect a))
```

```
> (drop-less 10 '(7 8 9 10 11))  
(10 11)
```

Решение 8.81.6

```
(defun drop-less (n w)  
  (remove-if #'(lambda (a) (< a n)) w))
```

```
> (drop-less 10 '(7 8 9 10 11))  
(10 11)
```

Задача 8.82 drop.lisp

Определить функцию удаления из списка первого встреченного элемента соответствующего указанному значению.

Решение 8.82.1

```
(defun drop (e w)  
  (cond ((null w) nil)
```

```
((equalp (car w) e) (cdr w))
(t (cons (car w) (drop e (cdr w))))))
```

```
> (drop 'b '(a b c))
(A C)
```

Решение 8.82.2

```
(defun drop (e w)
  (when w (if (equalp (car w) e) (cdr w)
              (cons (car w) (drop e (cdr w))))))
> (drop 'b '(a b c))
(A C)
```

Решение 8.82.3

```
(defun drop (e w) (remove e w :count 1))
> (drop 'b '(a b c))
(A C)
```

Задача 8.83 shell-sort.lisp

If I sound grumpy, it is only because I have come across too many idiots of the "it can't be done" persuasion lately, the kind of managers who have an aquarium in their office because fifteen brains think better than one.

Erik Naggum

Определить функцию сортировки Шелла методом Кнута.

Решение 8.83.1

```
(defparameter *gs* '(9841 3280 1093 364 121 40 13 4 1))

(defun gap-insertion-sort (a p g &aux (n (length a)))
  (if (< n 2) a
      (do ((i 1 (1+ i))) ((eql i n) a)
        (do ((x (aref a i))
              (j i (- j g)))
            ((or (minusp (- j g))
                 (not (funcall p x (aref a (1- j))))))
          (setf (aref a j) x)
          (setf (aref a j) (aref a (- j g)))))))

(defun shell-sort (a p &optional (gs *gs*))
  (dolist (g gs a)
    (gap-insertion-sort a p g)))

> (shell-sort #(5 3 8 1 9 3 4 7 5 7 6 2 4 0 3 1) #'<)
```

```
#(0 1 1 2 3 3 3 4 4 5 5 6 7 7 8 9)
```

Задача 8.84 merge-sort.lisp

Определить функцию сортировки слиянием.

Решение 8.84.1

```
(defun merge-sort (w p &aux (n (floor (length w) 2)))
  (if (zerop n) (copy-seq w)
      (merge 'list (merge-sort (subseq w 0 n) p)
              (merge-sort (subseq w n) p)
              p)))

> (merge-sort '(5 3 8 1 9 3 4 7 5 7 6 2 4 0 3 1) #'<)
(0 1 1 2 3 3 3 4 4 5 5 6 7 7 8 9)
```

Задача 8.85 n-list.lisp

Определить функцию, возвращающую элемент-список по заданому номеру из заданного списка.

Решение 8.85.1

```
(defun n-list (w n)
  (cond ((null w) nil)
        ((and (= n 1) (listp (car w))) (car w))
        ((listp (car w)) (n-list (cdr w) (1- n)))
        (t (n-list (cdr w) n))))

> (n-list '((12) (3 0) 8 5 a (e t) g u) 3)
(E T)
> (n-list '((12) (3 0) 8 5 a (e t) g u) 4)
NIL
```

Решение 8.85.2

```
(defun n-list (w n)
  (cond ((null w) nil)
        ((listp (car w)) (if (= n 1) (car w) (n-list (cdr w) (1- n))))
        (t (n-list (cdr w) n))))

> (n-list '((12) (3 0) 8 5 a (e t) g u) 3)
(E T)
> (n-list '((12) (3 0) 8 5 a (e t) g u) 4)
NIL
```

Решение 8.85.3

```
(defun n-list (w n)
  (cond ((null w) nil)
```

```
((atom (car w)) (n-list (cdr w) n))
(t (if (= n 1) (car w) (n-list (cdr w) (1- n))))))
```

```
> (n-list '((12) (3 0) 8 5 a (e t) g u) 3)
(E T)
> (n-list '((12) (3 0) 8 5 a (e t) g u) 4)
NIL
```

Решение 8.85.4

```
(defun n-list (w n)
  (when w (if (atom (car w)) (n-list (cdr w) n)
              (if (= n 1) (car w) (n-list (cdr w) (1- n))))))
```

```
> (n-list '((12) (3 0) 8 5 a (e t) g u) 3)
(E T)
> (n-list '((12) (3 0) 8 5 a (e t) g u) 4)
NIL
```

Решение 8.85.5

```
(defun n-list (w n)
  (when w (if (and (= n 1) (listp (car w)) (= n 1)) (car w)
              (n-list (cdr w) (if (atom (car w)) n (1- n))))))
```

```
> (n-list '((12) (3 0) 8 5 a (e t) g u) 3)
(E T)
> (n-list '((12) (3 0) 8 5 a (e t) g u) 4)
NIL
```

Решение 8.85.6

```
(defun n-list (w n &aux (a (car w)) (d (cdr w)))
  (when w (if (atom a) (n-list d n)
              (if (= n 1) a (n-list d (1- n))))))
```

```
> (n-list '((12) (3 0) 8 5 a (e t) g u) 3)
(E T)
> (n-list '((12) (3 0) 8 5 a (e t) g u) 4)
NIL
```

Решение 8.85.7

```
(defun n-list (w n)
  (nth (1- n) (remove-if-not #'listp w)))
```

```
> (n-list '((12) (3 0) 8 5 a (e t) g u) 3)
(E T)
> (n-list '((12) (3 0) 8 5 a (e t) g u) 4)
NIL
```

Решение 8.85.8

```
(defun n-list (w n)
  (when (plusp n) (nth (1- n) (remove-if-not #'listp w))))

> (n-list '((12) (3 0) 8 5 a (e t) g u) 3)
(E T)
> (n-list '((12) (3 0) 8 5 a (e t) g u) 4)
NIL
```

Задача 8.86 mp.lisp

Определить функционал, который применяет функцию *f* сначала к первым элементам списков *w1*, *w2*... *wk*, затем – ко вторым, и т.д. и формирующий список результатов.

Решение 8.86.1

```
(defun mp (f &rest w) (apply #'mapcar f w))

> (mp #'* '(2 3 4) '(2 3 4) '(2 3 4))
(8 27 64)
```

Решение 8.86.2 (автор – helter, www.cyberforum.ru)

```
(defun mp (f &rest w)
  (apply #'mapcar f w))

> (mp #'* '(2 3 4) '(2 3 4) '(2 3 4))
(8 27 64)
```

Задача 8.87 add.lisp

Определить функцию, которая прибавляет к элементам списка *w* число *n*.

Решение 8.87.1

```
(defun add (w n)
  (mapcar #'(lambda (a) (+ a n)) w))

> (add '(1 2 3) 100)
(101 102 103)
```

Решение 8.87.2

```
(defun add (w n)
  (when w (cons (+ (car w) n)
                 (add (cdr w) n))))

> (add '(1 2 3) 100)
(101 102 103)
```

Решение 8.87.3


```
(defun add (w n)
  (loop for a in w collect (+ a n)))

> (add '(1 2 3) 100)
(101 102 103)
```

Задача 8.88 *deep-abate.lisp*

Самым странным языком мира является пирахан, с помощью которого общается небольшой народ охотников-собирателей Амазонии. Кроме того, в пирахане нет числительных и обозначений цвета. Для определения количества носители языка используют слова «несколько», «еще больше» и «много», для цвета – названия предметов: так, «кровь» обозначает «красный», а «лист» – «зеленый». Также в пирахане самое маленькое количество звуков – 3 гласных и 8 согласных, а у глаголов отсутствуют временные формы. Наконец, этому языку неизвестна рекурсия, характерная для сложноподчиненных предложений.

~

Определить функцию, которая уменьшает все числовые атомы списка на заданное значение, учитывая все атомы подсписков.

Решение 8.88.1

```
(defun deep-abate (w n)
  (cond ((null w) nil)
        ((listp (car w)) (cons (deep-abate (car w) n)
                                (deep-abate (cdr w) n)))
        ((numberp (car w)) (cons (- (car w) n)
                                (deep-abate (cdr w) n)))
        (t (cons (car w) (deep-abate (cdr w) n)))))

> (deep-abate '(11 22 (a b 33)) 11)
(0 11 (A B 22))
```

Решение 8.88.2

```
(defun deep-abate (w n &aux (a (car w)))
  (when w (cons (cond ((listp a) (deep-abate a n))
                  ((numberp a) (- a n))
                  (t a))
                (deep-abate (cdr w) n))))

> (deep-abate '(11 22 (a b 33)) 11)
(0 11 (A B 22))
```

Решение 8.88.3

```
(defun deep-dec (w n)
  (loop for a in w
        if (listp a) collect (deep-dec a n)
        else if (numberp a) collect (- a n)
        else collect a))

> (deep-abate '(11 22 (a b 33)) 11)
(0 11 (A B 22))
```

Задача 8.89 butlast-elm-from-file.lisp

I was also a heavy vi user in UNIX System V days (pre-vim). When I finally switched to Emacs, I never switched back. Others are recommending Evil but I think you would do better with a clean break and just learn Emacs. You can start pretty small with just the standard editing commands and macros, but once you actually start doing Emacs Lisp, you turn a corner and really can't go back. There are modern editors that are also programmable with imperative languages, e.g. Atom dot IO is programmable in JavaScript, but there's something about Lisp that cannot be reproduced by any other language IMO.

Reese Currie

Прочитать из файла значение переменной *a*. Написать рекурсивную функцию, удаляющую последнее вхождение *a* из списка *w*.

Решение 8.89.1

```
d:/test.txt
a
a

(defun butlast-elm-from-file (path w &aux e)
  (with-open-file (s path :direction :input)
    (do ((line (read-line s nil :eof) (read-line s nil :eof)))
        ((eq line :eof))
        (setf e (concatenate 'string e line))))
  (butlast-elm e w))

(defun butlast-elm (e w &aux (a (car w)) (d (cdr w)))
  (when w (if (and (equalp a e) (not (member e d :test #'equalp))) d
              (cons a (butlast-elm e d)))))

> (butlast-elm-from-file "d:/test.txt" '("aa" "bb" "aa" "bb"))
("aa" "bb" "bb")
```

Решение 8.89.2

```
d:/test.txt
a
a

(defun butlast-elm-from-file (path w &aux e)
  (with-open-file (s path :direction :input)
    (do ((line (read-line s nil :eof) (read-line s nil :eof)))
        ((eql line :eof))
        (setf e (concatenate 'string e line))))
  (butlast-elm (read-from-string e) w))

(defun butlast-elm (e w &aux (a (car w)) (d (cdr w)))
  (when w (if (and (equalp a e) (not (member e d :test #'equalp))) d
              (cons a (butlast-elm e d)))))

> (butlast-elm-from-file "d:/test.txt" '(aa bb aa bb))
(AA BB BB)
```

Задача 8.90 del-every-k.lisp

Определить функцию, удаляющую из списка каждый k-ый элемент.

Решение 8.90.1

```
(defun del-every-k (w k &optional (n 1))
  (when w (if (= n k) (del-every-k (cdr w) k 1)
              (cons (car w) (del-every-k (cdr w) k (1+ n))))))

> (del-every-k '(1 2 3 4 5 6 7 8 9 10) 3)
(1 2 4 5 7 8 10)
```

Решение 8.90.2

```
(defun del-every-k (w k)
  (loop for a in w for b from 1
        unless (zerop (rem b k)) collect a))

> (del-every-k '(1 2 3 4 5 6 7 8 9 10) 3)
(1 2 4 5 7 8 10)
```

Задача 8.91 exact-sum-sequence.lisp

Дан числовой список и число. Найти непрерывную последовательность (последовательности), сумма элементов которой равна этому числу.

Решение 8.91.1

```
(defun accs (w &optional (acc (list (car w))) &aux (d (cdr w)))
```

```

(if d
  (cons (reverse acc)
        (accs d (cons (car d) acc))))
(list (reverse acc)))

(defun exact-sum-sequence (w
                           m
                           &aux
                           (v (loop for d on w
                                     nconc (accs d))))
  (z (mapcar
      #'(lambda (a)
           `(,a ,(reduce #' + a)))
      v)))
(mapcar #'car (remove-if-not #'(lambda (a) (= (cadr a) m)) z)))

> (exact-sum-sequence '(1 3 -16 5 7 8 2 2) 12)
((1 3 -16 5 7 8 2 2) (5 7) (8 2 2))

```

Задача 8.92 sub<=n.lisp

Emacs: I cut my teeth on it and was forced to work with it, so it is part of my blood. Once the first pain of learning it was over, there is a beautiful elegance to it. It is so easy to extend it, for short term or for the long haul, as you type, it is like an extension of my fingers.

Daniel Billotte

Определить функцию, которая формирует список, состоящий из не более чем n элементов исходного списка.

Решение 8.92.1

```

(defun sub<=n (n w)
  (cond ((null w) nil)
        ((zerop n) nil)
        ((cons (car w) (sub<=n (1- n) (cdr w))))))

> (sub<=n 4 '(1 2 3 4 5 6 7))
(1 2 3 4)

```

Решение 8.92.2

```

(defun sub<=n (n w)
  (when (and w (plussp n))
    (cons (car w) (sub<=n (1- n) (cdr w)))))

> (sub<=n 4 '(1 2 3 4 5 6 7))
(1 2 3 4)

```

Решение 8.92.3

```
(defun sub<=n (n w)
  (loop for a from 1 to n for b in w
        collect b))

> (sub<=n 4 '(1 2 3 4 5 6 7))
(1 2 3 4)
```

Задача 8.93 *drop.lisp*

Lisp exists to show how wrong every other language designer has been.

Jarrod Roberson

Определить функцию, которая из удаляет элементы e из всех уровней списка w.

Решение 8.93.1

```
(defun drop (e w)
  (cond ((null w) nil)
        ((equal (car w) e) (drop e (cdr w)))
        ((listp (car w)) (cons (drop e (car w)) (drop e (cdr w))))
        ((cons (car w) (drop e (cdr w)))))

> (drop 'a '(a b a (b a c x) x b))
(B (B C X) X B)
```

Решение 8.93.2

```
(defun drop (e w &aux (a (car w)) (d (cdr w)))
  (cond ((null w) nil)
        ((equal a e) (drop e d))
        ((listp a) (cons (drop e a) (drop e d)))
        ((cons a (drop e d)))))

> (drop 'a '(a b a (b a c x) x b))
(B (B C X) X B)
```

Решение 8.93.3

```
(defun drop (e w &aux (a (car w)) (d (cdr w)))
  (when w (if (equal a e) (drop e d)
              (cons (if (listp a) (drop e a) a)
                    (drop e d)))))

> (drop 'a '(a b a (b a c x) x b))
(B (B C X) X B)
```

Решение 8.93.4

```
(defun drop (e w)
  (loop for a in w unless (equal a e)
        collect (if (listp a) (drop e a) a)))
```

```
> (drop 'a '(a b a (b a c x) x b))
(B (B C X) X B)
```

Решение 8.93.5

```
(defun drop (e w)
  (loop for a in w unless (equal a e)
        collect (if (atom a) a (drop e a))))
```

```
> (drop 'a '(a b a (b a c x) x b))
(B (B C X) X B)
```

Задача 8.94 compliment.lisp

...the smartest companies use Common Lisp, but
lie about it so all their competitors think
Lisp is slow and C++ is fast.

Erik Naggum

Определить функцию, которая вставляет в конец списка w такой элемент, чтобы сумма нового списка была равна m.

Решение 8.94.1

```
(defun compliment(w m)
  (if w (cons (car w) (compliment (cdr w) (- m (car w))))
      (list m)))
```

```
> (compliment '(1 2 3 4 5) 25)
(1 2 3 4 5 10)
```

Решение 8.94.2

```
(defun compliment(w m &aux (a (car w)))
  (if w (cons a (compliment (cdr w) (- m a))) `((,m))))
```

```
> (compliment '(1 2 3 4 5) 25)
(1 2 3 4 5 10)
```

Решение 8.94.3

```
(defun compliment(w m) (append w `((,(- m (reduce #'+ w)))))
```

```
> (compliment '(1 2 3 4 5) 25)
(1 2 3 4 5 10)
```

Задача 8.95 *neci.lisp*

If you give someone Fortran, he has Fortran.
 If you give someone Lisp, he has any language he pleases.

Guy Steele

Дан список, состоящий из подсписков вида (2 Petrov 200 1) и целое число n , не равное нулю. Определить функцию, которая возвращает список результатов деления третьих элементов подсписков на n .

Решение 8.95.1

```
(defun neci (w n)
  (when w (cons (/ (caddr w) n) (neci (cdr w)))))

> (neci '((1 Ivanov 300 1) (2 Petrov 200 1) (3 Sidorov 100 2)) 10)
(30 20 10)
```

Решение 8.95.2

```
(defun neci (w n)
  (loop for a in w collect (/ (caddr a) n)))

> (neci '((1 Ivanov 300 1) (2 Petrov 200 1) (3 Sidorov 100 2)) 10)
(30 20 10)
```

Решение 8.95.3

```
(defun neci (w n)
  (mapcar #'(lambda (a) (/ (third a) n)) w))

> (neci '((1 Ivanov 300 1) (2 Petrov 200 1) (3 Sidorov 100 2)) 10)
(30 20 10)
```

Задача 8.96 *plus-minus.lisp*

С клавиатуры последовательно вводятся: в первой строке n (длина списка), во второй строке вводится список - числа, разделенные пробелами. Определить функцию, выводящую на экран построчно: десятичное представление доли положительных чисел, десятичное представление доли отрицательных чисел, десятичное представление доли нулей в списке.

Пример ввода:

```
6
-4 3 -9 0 4 1
```

Образец вывода:

```
0.500000
0.333333
0.166667
```

Решение 8.96.1

```

(defun plus-minus (&aux
                  (n (read))
                  (v (pmz (read-from-string
                          (concatenate
                           'string "(" (read-line) ")")))))
  (form n v))

(defun pmz (w)
  (loop for a in w
        if (plusp a) count a into p
        else if (minusp a) count a into m
        else count a into z
        finally (return (list p m z))))

(defun form (n w)
  (loop for a in w collect
        (format t "~,6f~%" (float (/ a n)))))

> (plus-minus)
6
-4 3 -9 0 4 1
0.500000
0.333333
0.166667
(NIL NIL NIL)

```


Структура 9 (функция (СПИСОК) (СПИСОК)) > АТОМ

Sometimes, the only way to learn something really well is to revert to the state of mind of a novice and reawaken to the raw observations that you have accumulated instead of relying on the conclusions you have reached from the exogenous premises absorbed through teaching and bookish learning.

Erik Naggum

The difference in time between Stegosaurus and T-rex is greater than the difference in time between T-rex and now.

~

Задача 9.1 *count-xpr.lisp*

Значение функции равно количеству вхождений заданного s-выражения в список. Если s-выражение не принадлежит списку или список пустой, то значение функции nil.

Решение 9.1.1

```
(defun count-xpr (e w)
  (when w (cond ((equal e (car w)) (1+ (or (count-xpr e (cdr w)) 0)))
                ((count-xpr e (cdr w))))))

> (cnt '(a 1) '((a 1) (a 1)))
2
```

Задача 9.2 *equalsetp.lisp*

Написать функцию, которая проверяет равны ли два множества, представленные двумя списками v и w, которые заданы в качестве параметров.

Решение 9.2.1

```
(defun equalsetp (v w)
  (and (subsetp v w) (subsetp w v)))

> (equalsetp '(1 1 2 3) '(1 2 2 3))
T
```

Решение 9.2.2

```
(defun equalsetp (v w) (and (subp v w) (subp w v)))
```

```
(defun subp (v w)
  (cond ((null v))
        ((member (car v) w) (subp (cdr v) w))
        (nil)))
```

```
> (equalsetp '(1 1 2 3) '(1 2 2 3))
T
```

Задача 9.3 *every>.lisp*

Решить задачу на параллельную рекурсию: определить предикат, проверяющий, что все элементы первого числового списка больше элементов второго списка.

Решение 9.3.1

```
(defun every> (w v)
  (cond ((null w) t)
        ((not (ev (car w) v)) nil)
        ((every> (cdr w) v))))
```

```
(defun ev (a v)
  (cond ((null v) t)
        ((< a (car v)) nil)
        ((ev a (cdr v)))))
```

```
> (every> '(4 5 6 7) '(1 2 3))
T
> (every> '(4 1 6 7) '(1 2 3))
NIL
```

Решение 9.3.2

```
(defun every> (w v)
  (cond ((null w) t)
        ((some #'(lambda (e) (< (car w) e)) v) nil)
        ((every> (cdr w) v))))
```

```
> (every> '(4 5 6 7) '(1 2 3))
T
> (every> '(4 1 6 7) '(1 2 3))
NIL
```

Решение 9.3.3

```
(defun every> (w v)
  (> (apply #'min w) (apply #'max v)))
```

```
> (every> '(4 5 6 7) '(1 2 3))
T
> (every> '(4 1 6 7) '(1 2 3))
```

NIL

Задача 9.4 same-elm&pos.lisp

Даны два списка, проверить стоят ли одинаковые элементы списков на одинаковых позициях. Выводить *t*, если все одинаковые элементы стоят на одинаковых позициях, и *nil*, если хотя бы одна пара одинаковых элементов стоит на разных позициях.

Решение 9.4.1

```
(defun same-elm&pos (v w)
  (cond ((null v) t)
        ((or (member (car v) (cdr w)) (member (car w) (cdr v))) nil)
        ((same-elm&pos (cdr v) (cdr w)))))
```

```
> (same-elm&pos '(1 2 3 4) '(1 7 8 5))
```

```
T
```

```
> (same-elm&pos '(1 2 3 4) '(7 1 8 5))
```

```
NIL
```

Комментарии (построчно):

1. Функция *same-elm&pos* (список список);
2. Условие (добрались до конца списка, т.е. список *v* - пуст) > возвращаем *t* - истина;
3. Условие (голова первого списка входит в хвост второго списка или наоборот) > возвращаем *nil* - ложь;
4. Если не выполнено ни одно из условий > вызываем функцию *same-elm&pos* с хвостами списков.

Задача 9.5 over-atom.lisp

Определить функцию с двумя списками в качестве аргументов. Расположение скобок в *s*-выражениях этих списков одинаковое, т.е. (3 2 (6) 1) и (4 5 (7) 2). Значение функции - *t*, если каждый атом из первого списка больше соответствующего атома из второго списка, и *nil* в противном случае.

Решение 9.5.1

```
(defun over-atom (w v)
  (cond ((null w) t)
        ((listp (car w)) (over-atom (car w) (car v)))
        ((> (car w) (car v)) (over-atom (cdr w) (cdr v)))))
```

```
> (over-atom '(5 6 (7) 8) '(1 2 (3) 4))
```

```
T
```

Решение 9.5.2

```
(defun over-atom (w v)
```

```
(every #'(lambda (m n)
          (cond ((listp m) (over-atom m n))
                ((> m n))))
      w v))
```

```
> (over-atom '(5 6 (7) 8) '(1 2 (3) 4))
T
```

Решение 9.5.3

```
(defun over-atom (w v)
  (every #'(lambda (m n) (if (listp m) (over-atom m n) (> m n))) w v))
```

```
> (over-atom '(5 6 (7) 8) '(1 2 (3) 4))
T
```

Задача 9.6 same-order.lisp

Реализовать функцию, возвращающую *t* при идентичности порядка расположения одинаковых атомов в исходных списках.

Решение 9.6.1

```
(defun same-order (w v)
  (cond ((null w) t)
        ((eq (car w) (car w)) (same-order (cdr w) (cdr v)))
        ((member (car w) v) nil)
        ((same-order (cdr w) (cdr v)))))
```

```
> (same-order '(a b) '(a c))
T
```

Задача 9.7 sublistp.lisp

Написать функцию аргументов *w v*, возвращающую *t*, если *v* является под-списком *w*. Элементами списков могут быть атомы и списки любой вложенности.

Решение 9.7.1

```
(defun sublistp (w v)
  (when w ((lambda (a d)
              (if (atom a)
                  (sublistp d v)
                  (or (equal a v) (sublistp a v) (sublistp d v))))
            (car w) (cdr w))))
```

```
> (sublistp '(((j k) c (r p)) (u (r (m t) (a z) d) z)) '(a z))
T
```

Решение 9.7.2

```
(defun sublistp (w v)
  (when w ((lambda (a d)
              (cond ((atom a) (sublistp d v))
                    ((or (equal a v) (sublistp a v) (sublistp d v))))
            (car w) (cdr w))))

> (sublistp '(((j k) c (r p)) (u (r (m t) (a z) d) z)) '(a z))
T
```

Решение 9.7.3

```
(defun sublistp (v w)
  (cond ((null w) nil)
        ((equal v (car w)) t)
        ((atom (car w)) (sublistp v (cdr w)))
        ((or (sublistp v (car w)) (sublistp v (cdr w))))))

> (sublistp '(z z) '(a (a (a a) (z z))a))
T
```

Задача 9.8 *more-elements.lisp*

Написать программу, которая сравнивает два списка по количеству элементов.

Решение 9.8.1

```
(defun more-elements (w v)
  (cond ((and w (null v)) 'first)
        ((and v (null w)) 'second)
        ((null w) 'equal)
        ((more-elements (cdr w) (cdr v)))))

> (more-elements '(1) '(1 2))
SECOND
```

Задача 9.9 *more-atoms.lisp*

Написать программу, которая сравнивает два списка по количеству атомов.

Решение 9.9.1

```
(defun more-elements (w v)
  (cond ((and w (null v)) 'first)
        ((and v (null w)) 'second)
        ((null w) 'equal)
        ((more-elements (cdr w) (cdr v)))))

(defun flat (w &optional acc)
```

```

(cond ((null w) acc)
      ((atom w) (cons w acc))
      ((flat (car w) (flat (cdr w) acc)))))

(defun more-atoms (w v)
  (more-elements (flat w) (flat v)))

> (more-atoms '(1 2) '((1 2) 3))
SECOND

```

Решение 9.9.2

```

(defun more-elements (w v)
  (cond ((and w (null v)) 'first)
        ((and v (null w)) 'second)
        ((null w) 'equal)
        ((more-elements (cdr w) (cdr v)))))

(defun flat (w)
  (mapcan #'(lambda (a) (if (listp a) (flat a) `(',a))) w))

(defun more-atoms (w v)
  (more-elements (flat w) (flat v)))

> (more-atoms '(1 2) '((1 2) 3))
SECOND

```

Решение 9.9.3

```

(defun more-elements (w v)
  (cond ((and w (null v)) 'first)
        ((and v (null w)) 'second)
        ((null w) 'equal)
        ((more-elements (cdr w) (cdr v)))))

(defun flat (w)
  (loop for a in w
        if (listp a) append (flat a)
        else collect a))

(defun more-atoms (w v)
  (more-elements (flat w) (flat v)))

> (more-atoms '(1 2) '((1 2) 3))
SECOND

```

Задача 9.10 subset-next.lisp

Даны два списка элементов типа `symbol`. Если `v` есть подсписок `w`, то определить следующий после `v` элемент списка `w`, иначе ответом является слово "no".

Решение 9.10.1

```
(defun subset-next (v w)
  (cond ((null w) 'no)
        ((equal v (car w)) (cadr w))
        ((subset-next v (cdr w)))))

> (subset-next '(b c) '(a (b c) d))
D
```

Задача 9.11 maxum.lisp

Даны два списка чисел произвольной длины. Определить функцию которая вычисляет сумму элементов каждого списка и выводит наибольшую из них.

Решение 9.11.1

```
(defun maxum (w v)
  (max (reduce #'+ w) (reduce #'+ v)))

> (maxum '(1 1) '(1))
2
```

Решение 9.11.1

```
(defun sum (w)
  (if (null w) 0 (+ (car w) (sum (cdr w)))))

(defun mx (a b)
  (if (>= a b) a b))

(defun maxum (w v)
  (mx (sum w) (sum v)))

> (maxum '(1 1) '(1))
2
```

Задача 9.12 miss.lisp

Определите функцию, возвращающую первый совпавший в двух списках атом, либо nil, если таких атомов нет.

Решение 9.12.1

```
(defun miss (w v)
  (cond ((null w) nil)
        ((eq (car w) (car v)) (car w))
        ((miss (cdr w) (cdr v)))))

> (miss '(1 2) '(0 2))
2
```

Задача 9.13 reversed.lisp

Определить функцию, проверяющую является ли один список результатом реверсирования другого списка.

Решение 9.13.1

```
(defun reversed (w v &optional z)
  (if w (reversed (cdr w) v (cons (car w) z)) (equal z v)))

> (reversed '(1 2 3) '(3 2 1))
T
> (reversed '(1 2 3) '(1 2 3))
NIL
```

Задача 9.14 max-sum-pair.lisp

Определить функцию, находящую наибольшее значение $x_i + y_i$ - суммы элементов произвольных массивов одинаковой размерности.

Решение 9.14.1

```
(defun max-pair (w v)
  (reduce #'max (map 'array #' + w v)))

> (max-pair #(1 2 3) #(1 1 1))
4
```

Решение 9.14.2 * для списков

```
(defun max-sum-pair (w v)
  (reduce #'max (mapcar #' + w v)))

> (max-sum-pair '(0 1) '(1 1))
2
```

Задача 9.15 birthday-earlier.lisp

Известны даты рождения двух человек. Определить, кто из них младше.

Решение 9.15.1

```
(defun encode-date (n d m y)
  (list n (encode-universal-time 0 0 1 d m y)))

(defun birthday-earlier (v w)
  (if (<= (cadr (apply #'encode-date v))
        (cadr (apply #'encode-date w)))
      (car v)
      (car w)))
```



```
> (birthday-earlier '(ann 1 1 1990) '(rob 1 1 1991))
ANN
```

Задача 9.16 euclidian-distance.lisp

Определить функцию, находящую расстояние между двумя точками по их координатам.

Решение 9.16.1

```
(defun euclidian-distance (w v &aux (z (mapcar #'- w v)))
  (sqrt (reduce #'+ (mapcar #'* z z))))

> (euclidian-distance '(1.0 2.0 0.5) '(3.0 4.0 0.5))
2.828427
```

Решение 9.16.2

```
(defun euclidian-distance (w v)
  (sqrt (loop for c in (loop for a in w
                             for b in v
                             collect (- a b))
              sum (* c c))))

> (euclidian-distance '(1.0 2.0 0.5) '(3.0 4.0 0.5))
2.828427
```

Решение 9.16.3

```
(defun euclidian-distance (w v)
  (sqrt (loop for a in w
              for b in v
              sum (expt (- a b) 2)))))

> (euclidian-distance '(1.0 2.0 0.5) '(3.0 4.0 0.5))
2.828427
```

Задача 9.17 longer-than-p.lisp

Определить функцию-предикат, аргументами которой являются два списка, возвращающую *t*, если первый список длиннее второго.

Решение 9.17.1

```
(defun longer-than-p (w v)
  (cond ((and w (null v))
        ((null w) nil)
        ((longer-than-p (cdr w) (cdr v)))))

> (longer-than-p '(a b) '(a))
T
```

Решение 9.17.2

```
(defun longer-than-p (w v)
  (when w (if (and w (null v))
              t
              (longer-than-p (cdr w) (cdr v)))))

> (longer-than-p '(a b) '(a))
T
```

Решение 9.17.3

```
(defun longer-than-p (w v)
  (when w (cond ((and w (null v))
                 ((longer-than-p (cdr w) (cdr v)))))

> (longer-than-p '(a b) '(a))
T
```

Решение 9.17.4

```
(defun longer-than-p (w v)
  (when w (or (and w (null v))
              (longer-than-p (cdr w) (cdr v)))))

> (longer-than-p '(a b) '(a))
T
```

Задача 9.18 sum-expt.lisp

Определить функцию, возвращающую сумму элементов списка *w*, возведенных в степень второго элемента списка *v*.

Решение 9.18.1

```
(defun sum-expt (w v)
  (cond ((null w) 0)
        (t (+ (expt (car w) (cadr v)) (sum-expt (cdr w) v)))))

> (sum-expt '(1 2 3) '(1 2))
14
```

Решение 9.18.2

```
(defun sum-expt (w v)
  (if w (+ (expt (car w) (cadr v)) (sum-expt (cdr w) v)) 0))

> (sum-expt '(1 2 3) '(1 2))
14
```

Задача 9.19 *first-and-second.lisp*

This is actually no different than any other language. To what extent do C programmers see its punctuation? Commas, semicolons, braces, parens, brackets, etc, all convey meaning immediately without been obsessed about as such. Getting them right can be a significant hurdle as you struggle with the syntax. Once you stop seeing the & and instead think "address", you have got the hang of it. Perl hackers have mastered this technique. The reason many people who have learned C prefer to continue on the C branch of evolution is that they found the process of learning all that syntax quite painful. This is especially true for parentheses in C. You only need them in expressions that cross a fairly high complexity threshold and you can get rid of them by simplifying the expressions. Lisp is chock full of parentheses, with no way to get rid of them - if you simplify your expressions, you end up with more parentheses. The "syntax = pain" equation in most C programmer's heads translates to a desire to reduce the cost of learning a new language by trying to adapt it to the pain they have already been through. However, once you grok the parentheses in Lisp, they are not painful. In fact, they are so much more liberating and enabling and downright friendly that you would just love to have similar tools available in every other language you use. This is why the "economy of syntax" in C is really a "poverty of syntax" and the perceived verbosity of Lisp translates to a wealth of opportunity. Like so many other things in life, you rarely get only what you optimize for.

Erik Naggum

Определить функцию сравнения двух списков: равны ли их первые два элемента?

Решение 9.19.1

```
(defun first-and-second (w v &aux (z (mismatch w v)))
  (when z (= z 2)))
```

```
> (first-and-second '(a b c) '(a b c))
```

```

NIL
> (first-and-second '(a b c) '(a a a))
NIL
> (first-and-second '(a b c) '(a b a))
T

```

Задача 9.20 *sum-product.lisp*

People who program in the main-stream languages because they are main-stream languages have yet to form the prerequisite concepts to say "I want to program in C". They have not yet developed an "I" who can actually want anything on its own.

Erik Naggum

Определить функцию, вычисляющую сумму произведения элементов двух списков.

Решение 9.20.1

```

(defun sum-product (w v)
  (cond ((null w) 0)
        ((+ (* (car w) (car v)) (sum-product (cdr w) (cdr v))))))

> (sum-product '(1 2 3) '(1 2 3))
14

```

Решение 9.20.2

```

(defun sum-product (w v)
  (if w (+ (* (car w) (car v)) (sum-product (cdr w) (cdr v))) 0))

> (sum-product '(1 2 3) '(1 2 3))
14

```

Решение 9.20.3

```

(defun sum-product (w v)
  (reduce #' + (mapcar #' * w v)))

> (sum-product '(1 2 3) '(1 2 3))
14

```

Решение 9.20.4

```

(defun sum-product (w v)
  (loop for a in w
        for b in v sum (* a b)))

> (sum-product '(1 2 3) '(1 2 3))

```

Задача 9.21 sorted.lisp

Python very much tries to guide you into writing your code a specific way. Usually burning you when you don't comply to the Python "ethos". Lisp doesn't do that, it's quite happy to take your imperative code, or functional, or turn it into macros, or add object orientation onto your code, or make your own derivative language to suit your problem better, etc.

Irné Barnard

Определить функцию, которая проверяла бы отсортирован ли список (по неубыванию или по невозрастанию). Если он отсортирован в каком-то из этих направлений - выводим Т, иначе - выводим NIL.

Решение 9.21.1

```
(defun ascend (w)
  (cond ((null (cdr w)) t)
        ((> (car w) (cadr w)) nil)
        ((ascend (cdr w)))))

(defun descend (w)
  (cond ((null (cdr w)) t)
        ((< (car w) (cadr w)) nil)
        ((descend (cdr w)))))

(defun sorted (w) (or (ascend w) (descend w)))

> (sorted '(1 3 2))
NIL
> (sorted '(1 3 3))
T
> (sorted '(3 2 2))
T
```

Решение 9.21.2

```
(defun ascend (w)
  (cond ((null (cdr w)) t)
        ((> (car w) (cadr w)) nil)
        ((ascend (cdr w)))))

(defun descend (w)
  (cond ((null (cdr w)) t)
        ((< (car w) (cadr w)) nil)
        ((descend (cdr w)))))
```

```
(defun sorted (w) (or (ascend w) (descend w)))
```

```
> (sorted '(1 3 2))
NIL
> (sorted '(1 3 3))
T
> (sorted '(3 2 2))
T
```

Решение 9.21.3

```
(defun ascend (w)
  (if (null (cdr w)) t (and (<= (car w) (cadr w)) (ascend (cdr w)))))

(defun descend (w)
  (if (null (cdr w)) t (and (>= (car w) (cadr w)) (descend (cdr w)))))

(defun sorted (w) (or (ascend w) (descend w)))
```

```
> (sorted '(1 3 2))
NIL
> (sorted '(1 3 3))
T
> (sorted '(3 2 2))
T
```

Решение 9.21.4

```
(defun ascend (w)
  (if (cdr w) (and (<= (car w) (cadr w)) (ascend (cdr w))) t))

(defun descend (w)
  (if (cdr w) (and (>= (car w) (cadr w)) (descend (cdr w))) t))

(defun sorted (w) (or (ascend w) (descend w)))
```

```
> (sorted '(1 3 2))
NIL
> (sorted '(1 3 3))
T
> (sorted '(3 2 2))
T
```

Решение 9.21.5

```
(defun sorted (w) (or (apply #'<= w) (apply #'>= w)))
```

```
> (sorted '(1 3 2))
NIL
> (sorted '(1 3 3))
T
> (sorted '(3 2 2))
```

T

Задача 9.22 *eq-exp.lisp*

From the professional standpoint, I would suggest Scala, as you will probably have trouble finding jobs for Haskell and Lisp. But Scala is wickedly complicated. It is often considered a "write-only" language. From the academic standpoint, Haskell is probably the best choice. Its functional purity and mathematical rigor will give you a solid grounding. But who likes math??? From the standpoint of enjoyment, Lisp is a lot more fun, in my opinion. Lisp feels very dynamic, fluid and flexible. Lisp is much easier to learn than either Haskell or Scala. Being homoiconic and offering a powerful macro facility, Lisp is compelling.

Richard Kenneth Eng

Напишите функцию предикатного типа, которая определяет эквивалентность двух s-выражений.

Решение 9.22.1

```
(defun eq-exp (w v)
  (if (atom w) (eq w v)
      (and (eq-exp (car w) (car v))
            (eq-exp (cdr w) (cdr v)))))

> (eq-exp '(1 (2 3)) '(1 (2 3)))
T
> (eq-exp '(1 (2 3)) '(1 (2 2)))
NIL
> (equalp '(1 (2 3)) '(1 (2 3)))
T
> (equalp '(1 (2 3)) '(1 (2 2)))
NIL
> (equal '(1 (2 3)) '(1 (2 3)))
T
> (equal '(1 (2 3)) '(1 (2 2)))
NIL
> (eq1 '(1 (2 3)) '(1 (2 3)))
NIL
> (eq '(1 (2 3)) '(1 (2 3)))
NIL
```

Структура 10 (функция (СПИСОК) (СПИСОК)) > (СПИСОК)

Ембеддшики, пишущие real time приложения на Форте и голом ассемблере ими же разработанных процессоров - те гораздо круче каких-то там всего лишь C++-ников. Те, кто пишет на Haskell и Erlang - тоже в основной своей массе на порядки умнее и грамотнее среднего C++-ника. Те, кто пишет на Common Lisp - вообще аццкая элита, и ++-ник рядом с ними вообще не заметен будет.

www.sql.ru/forum/466654-1/s

Задача 10.1 *priority-sort.lisp*

Даны 8 троек - 0 0 0, 0 0 1, 0 1 0, 0 1 1, 1 0 0, 1 0 1, 1 1 0, 1 1 1. Рассортировать их с разными приоритетами для столбцов.

Решение 10.1.1

```
(defun priority-sort (w prs &optional (m (reduce #'max prs)))
  (if (zerop m)
      w
      (priority-sort
       (stable-sort w #'<
                    :key #'(lambda (p)
                             (nth (position m prs) p))) prs (1- m)))))

> (priority-sort '((0 0 0) (0 0 1) (0 1 0) (0 1 1) (1 0 0) (1 0 1) (1 1 0) (1 1 1)) '(1 2 3))
((0 0 0) (0 0 1) (0 1 0) (0 1 1) (1 0 0) (1 0 1) (1 1 0) (1 1 1))
> (priority-sort '((0 0 0) (0 0 1) (0 1 0) (0 1 1) (1 0 0) (1 0 1) (1 1 0) (1 1 1)) '(3 2 1))
((0 0 0) (1 0 0) (0 1 0) (1 1 0) (0 0 1) (1 0 1) (0 1 1) (1 1 1))
```

Задача 10.2 *uni.lisp*

Написать рекурсивную функцию, реализующую объединение двух множеств w и v. Пример: (объединение '(1 2 3) '(1 2 4)) есть (1 2 3 4).

Решение 10.2.1

```
(defun uni (w v)
  (cond ((null w) v)
        ((member (car w) v) (uni (cdr w) v))
        ((cons (car w) (uni (cdr w) v)))))
```



```
> (uni '(1 2 3) '(1 2 4))
(3 1 2 4)
```

Решение 10.2.2

```
(defun uni (w v)
  (sort (union w v) #'<))

> (uni '(1 2 3) '(1 2 4))
(1 2 3 4)
```

Решение 10.2.3

```
(defun uni (w v)
  (delete-duplicates (merge 'list w v #'<)))

> (uni '(1 2 3) '(1 2 4))
(1 2 3 4)
```

Задача 10.3 *mix-elements.lisp*

Написать функцию, которая чередует элементы двух списков: (a b c...) (1 2 3...) > (a 1 b 2 c 3...) .

Решение 10.3.1

```
(defun mix-elms (w v)
  (cond ((null w) v) ((null v) w) ((mix w v))))

(defun mix (w v)
  (when w (cons (car w) (mix v (cdr w)))))

> (mix-elms '(1 2 3) '(a b c))
(1 A 2 B 3 C)
```

Решение 10.3.2 (автор - Catstail, www.cyberforum.ru)

```
(defun mix-elms (w v)
  (cond ((null w) v) ((null v) w)
        ((cons (car w) (cons (car v) (mix-elms (cdr w) (cdr v)))))))

> (mix-elms '(1 2 3) '(a b c))
(1 A 2 B 3 C)
> (mix-elms '(1 2 3) '(a b c d))
(1 A 2 B 3 C D)
> (mix-elms '(1 2 3 4 5) '(a b c d))
(1 A 2 B 3 C 4 D 5)
```

Решение 10.3.3 (автор - Nameless One, www.cyberforum.ru) *корректное лишь при равном числе элементов в списках

```
(defun mix-elms (w v)
```

```
(mapcan #'list w v))

> (mix-elms '(1 2 3) '(a b c))
(1 A 2 B 3 C)
> (mix-elms '(1 2 3) '(a b c d))
(1 A 2 B 3 C)
> (mix-elms '(1 2 3 4 5) '(a b c d))
(1 A 2 B 3 C 4 D)
```

Решение 10.3.1

```
(defun mix-elements (w v)
  (if w (cons (car w) (mix-elements v (cdr w))) v))

> (mix-elements '(a b c) '(1 2 3))
(A 1 B 2 C 3)
```

Задача 10.4 alter-mix.lisp

Используя рекурсию преобразовать два исходных списка ((a b) c d e (f) g h (i j k)...) и ((1) 2 3 (4 5 6 7) 8 (9 10)...) . Получить ((a b 2 3) (1 c d e) (f 8) (4 5 6 7 g h) (i j k...) (9 10...)...).

Решение 10.4.1

```
(defun alter-mix (w v)
  (glue (thick w) (thick v)))

(defun thick (w &optional ac)
  (cond ((null w) (if ac (cons (nreverse ac) nil) nil))
        ((listp (car w))
         (if ac
              (nconc (list (nreverse ac) (car w))
                      (thick (cdr w) nil))
              (cons (car w) (thick (cdr w) nil))))
        ((push (car w) ac) (thick (cdr w) ac))))

(defun glue (w v)
  (cond ((null w) nil)
        ((nconc
          (list
            (nconc (car w) (cadr v))
            (nconc (car v) (cadr w)))
          (glue (cddr w) (cddr v))))))

> (alter-mix '((a b) c d e (f) g h (i j k) l m) '((1) 2 3 (4 5 6 7) 8
(9 10) 11 12))
((A B 2 3) (1 C D E) (F 8) (4 5 6 7 G H) (I J K 11 12) (9 10 L M))
```

Решение 10.4.2

```
(defun alter-mix (w v)
```

```

(glue (thick w) (thick v)))

(defun thick (w &optional ac)
  (cond ((null w) (if ac (cons (nreverse ac) nil) nil))
        ((listp (car w))
         (if ac
              (list* (nreverse ac) (car w) (thick (cdr w) nil))
              (cons (car w) (thick (cdr w) nil))))
        ((push (car w) ac) (thick (cdr w) ac))))

(defun glue (w v)
  (cond ((null w) nil)
        ((list* (nconc (car w) (cadr v))
                 (nconc (car v) (cadr w))
                 (glue (cddr w) (cddr v))))))

> (alter-mix '((a b) c d e (f) g h (i j k) l m) '((1) 2 3 (4 5 6 7) 8
(9 10) 11 12))
((A B 2 3) (1 C D E) (F 8) (4 5 6 7 G H) (I J K 11 12) (9 10 L M))

```

Задача 10.5 cartesian-product.lisp

Написать на Лиспе программу смешивания двух списков. Например вызывать ее так: (mix '(a b c d) '(1 2 3)), и чтобы результат был такой: ((a 1) (a 2) (a 3) (b 1) (b 2) ... (d 2) (d 3)).

Решение 10.5.1

```

(defun cartesian-product (w v)
  (cond ((null w) nil)
        ((append (mi (car w) v)
                  (cartesian-product (cdr w) v)))))

(defun mi (a w)
  (cond ((null w) nil)
        ((cons (list a (car w)) (mi a (cdr w))))))

> (cartesian-product '(a b c) '(1 2 3))
((A 1) (A 2) (A 3) (B 1) (B 2) (B 3) (C 1) (C 2) (C 3))

```

Решение 10.5.2

```

(defun cartesian-product (w v)
  (when w (nconc (mi (car w) v)
                 (cartesian-product (cdr w) v))))

(defun mi (a v)
  (when v (cons (list a (car v)) (mi a (cdr v)))))

> (cartesian-product '(a b c) '(1 2 3))
((A 1) (A 2) (A 3) (B 1) (B 2) (B 3) (C 1) (C 2) (C 3))

```

Решение 10.5.3

```
(defun cartesian-product (w v)
  (when w
    (append
      (maplist
        #'(lambda (e) (list (car w) (car e)))) v)
      (cartesian-product (cdr w) v))))

> (cartesian-product '(a b c) '(1 2 3))
((A 1) (A 2) (A 3) (B 1) (B 2) (B 3) (C 1) (C 2) (C 3))
```

Решение 10.5.4

```
(defun cartesian-product (w v)
  (apply
    #'nconc
    (mapcar
      #'(lambda (e) (mapcar #'(lambda (a) (list e a)) v)) w)))

> (cartesian-product '(a b c) '(1 2 3))
((A 1) (A 2) (A 3) (B 1) (B 2) (B 3) (C 1) (C 2) (C 3))
```

Решение 10.5.5

```
(defun cartesian-product (w v)
  (loop for a in w nconc
        (loop for b in v collect (list a b))))

> (cartesian-product '(a b c) '(1 2 3))
((A 1) (A 2) (A 3) (B 1) (B 2) (B 3) (C 1) (C 2) (C 3))
```

Решение 10.5.6 (автор – Max Vasin, www.linux.org.ru)

```
(defun cartesian-product (w v)
  (mapcan (lambda (e) (mapcar (lambda (a) `(,a ,e)) w)) v))

> (cartesian-product '(a b c) '(1 2 3))
((A 1) (A 2) (A 3) (B 1) (B 2) (B 3) (C 1) (C 2) (C 3))
```

Задача 10.6 exhp.lisp

Функция возводит элемент списка *w* в степень равную элементу из списка *v*, записывая каждый результат в выходной список. При этом:
 Если элемент списка *v* не является целым числом, а элемент списка *w* – является, функция возвращает в результирующем списке запись «E1».
 Если элемент списка *w* не является целым числом, а элемент списка *v* – является, функция возвращает в результирующем списке запись «E2».
 Если оба элемента не являются целыми числами, функция возвращает в результирующем списке запись «E3».

Решение 10.6.1

```
(defun exxp (w v)
  (mapcar #' (lambda (aw av)
    (cond ((and (zerop (mod aw 1)) (/= (mod av 1) 0)) 'e1)
          ((and (/= (mod aw 1) 0) (zerop (mod av 1))) 'e2)
          ((and (/= (mod aw 1) 0) (/= (mod av 1) 0)) 'e3)
          ((expt aw av))))
    w v))

> (exxp '(2 2.1 2.1 2) '(2.1 2 2.1 2))
(E1 E2 E3 4)
```

Задача 10.7 num-v-carvcdrw.lisp

Есть два списка. Если первый элемент списка есть натуральное число, то вернуть второй список, иначе вернуть список из головы второго и хвоста первого.

Решение 10.7.1

```
(defun num-v-carvcdrw (w v)
  (cond ((and (integerp (car w)) (plusp (car w))) v)
        ((cons (car v) (cdr w)))))

> (num-v-carvcdrw '(1 2 3) '(a b c))
(A B C)
> (num-v-carvcdrw '(0 2 3) '(a b c))
(A 2 3)
```

Задача 10.8 compare.lisp

Определите функцию, которая из двух списков одинаковой длины строит список из t и nil в результате попарного сравнения атомов, например, в результате вызова (f '(5 3 9 7 4) '(5 2 9 7 8)) выдается ответ (T Nil T T Nil).

Решение 10.8.1

```
(defun compare (w v)
  (cond ((null w) nil)
        ((cons (eq (car w) (car v)) (compare (cdr w) (cdr v))))))

> (compare '(5 3 9 7 4) '(5 2 9 7 8))
(T NIL T T NIL)
```

Решение 10.8.2

```
(defun compare (w v)
  (mapcar #'eq w v))

> (compare '(5 3 9 7 4) '(5 2 9 7 8))
```

```
(T NIL T T NIL)
```

Задача 10.9 *mix-wwwv.lisp*

Есть два списка w (a b) и v (1 2). Определить функцию, чтобы на выходе получалось (a 1 b 2).

Решение 10.9.1

```
(defun mix-wwwv (w v)
  (cond ((null w) v) ((null v) w) ((mi w v))))
```

```
(defun mi (w v)
  (cond (w (cons (car w) (mi v (cdr w))))
        ((when v (nconc v nil)))))
```

```
> (mix-wwwv '(1 2 3 4) '(a b c d))
(1 A 2 B 3 C 4 D)
> (mix-wwwv '(1 2 3 4) '(a b c))
(1 A 2 B 3 C 4)
> (mix-wwwv '(1 2 3 4) '(a b))
(1 A 2 B 3 4)
> (mix-wwwv '(1 2 3 4) '(a))
(1 A 2 3 4)
> (mix-wwwv '(1 2 3 4) '())
(1 2 3 4)
> (mix-wwwv '(1 2 3) '(a b c d))
(1 A 2 B 3 C D)
> (mix-wwwv '(1 2) '(a b c d))
(1 A 2 B C D)
> (mix-wwwv '(1) '(a b c d))
(1 A B C D)
> (mix-wwwv '() '(a b c d))
(A B C D)
```

Решение 10.9.2

```
(defun mix-wwwv (w v)
  (if w (cons (car w) (mix-wwwv v (cdr w))) v))
```

```
> (mix-wwwv '(a b c) '(1 2 3))
(A 1 B 2 C 3)
```

Решение 10.9.3

```
(defun mix-wwwv (w v) (mapcan #'list w v))
```

```
> (mix-wwwv '(a b c) '(1 2 3))
(A 1 B 2 C 3)
```

Задача 10.10 mult-pairs.lisp

Перемножить поэлементно 2 числовых списка и вернуть результирующий список.

Решение 10.10.1

```
(defun mult-pairs (w v)
  (when (and w v)
    (cons (* (car w) (car v)) (mult-pairs (cdr w) (cdr v)))))
```

```
> (mult-pairs '(1 2 3) '(2 2 2))
(2 4 6)
```

Решение 10.10.2

```
(defun mult-pairs (w v) (mapcar #'* w v))
```

```
> (mult-pairs '(1 2 3) '(2 2 2))
(2 4 6)
```

Решение 10.10.3

```
(defun mult-pairs (w v) (loop for a in w for b in v collect (* a b)))
```

```
> (mult-pairs '(1 2 3) '(2 2 2))
(2 4 6)
```

Задача 10.11 our-set-difference.lisp

Разработать функцию, находящую теоретико-множественную разность двух списков.

Решение 10.11.1

```
(defun our-set-difference (w v)
  (cond ((null w) nil)
        ((member (car w) v) (our-set-difference (cdr w) v))
        ((cons (car w) (our-set-difference (cdr w) v)))))
```

```
> (our-set-difference '(1 2 3 4 5) '(4 5 6 7))
(1 2 3)
```

Решение 10.11.2

```
(defun our-set-difference (w v)
  (when w (if (member (car w) v)
              (our-set-difference (cdr w) v)
              (cons (car w) (our-set-difference (cdr w) v)))))
```

```
> (our-set-difference '(1 2 3 4 5) '(4 5 6 7))
(1 2 3)
```

Решение 10.11.3

```
(defun our-set-difference (w v &optional ac &aux (a (car w)))
  (cond ((null w) (nreverse ac))
        ((member a v) (our-set-difference (cdr w) v ac))
        ((our-set-difference (cdr w) v (cons a ac )))))

> (our-set-difference '(1 2 3 4 5) '(4 5 6 7))
(1 2 3)
```

Решение 10.11.4

```
(defun our-set-difference (w v &optional ac &aux (a (car w)))
  (if w
      (our-set-difference (cdr w) v (if (member a v)
                                         ac
                                         (cons a ac)))
      (nreverse ac)))

> (our-set-difference '(1 2 3 4 5) '(4 5 6 7))
(1 2 3)
```

Решение 10.11.5

```
(defun our-set-difference (w v)
  (loop for a in w unless (member a v) collect a))

> (our-set-difference '(1 2 3 4 5) '(4 5 6 7))
(1 2 3)
```

Решение 10.11.6

```
(defun our-set-difference (w v)
  (loop for a in w unless (find a v) collect a))

> (our-set-difference '(1 2 3 4 5) '(4 5 6 7))
(1 2 3)
```

Задача 10.12 string-set-difference.lisp

Определить функцию, находящую теоретико-множественную разность двух списков строк.

Решение 10.12.1

```
(defun string-set-difference (w v)
  (loop for a in w
        unless (member a v :test #'string-equal)
        collect a))

> (string-set-difference '("a" "b") '("b" "c"))
```



```
("a")
```

Задача 10.13 -union.lisp

Определить функцию объединения двух множеств.

Решение 10.13.1

```
(defparameter w (loop repeat 1000 collect (random 1000)))
(defparameter v (loop repeat 1000 collect (random 1000)))

(defun -union (w v)
  (cond ((null w) v)
        ((member (car w) v) (-union (cdr w) v))
        ((cons (car w) (-union (cdr w) v)))))

> (time (-union w v))
Real time: 0.0156229 sec.
Run time: 0.015625 sec.
Space: 2832 Bytes
(151 884 629 836 930 651 236 592 529 31 471 349 641 834 415 160 143
189 335 20 652 799 390 189 858 148 148 683 434 877 520 919 503 890 44
210 993 959 356 404 103 720 998 700 33 869 830 551 311 839 148 302 230
302 997 407 487 3 465 9 749 460 285 335 454 50 929 511 470 601 592 31
183 939 29 569 917 169 734 398 513 920 461 375 621 39 518 488 274 889
288 630 480 323 755 198 127 237 20 59 20 192 210 606 261 148 551 720
630 9 263 405 528 92 359 274 853 454 990 868 556 915 30 919 925 138
503 81 678 766 839 558 33 198 272 889 983 558 148 170 64 818 746 489
930 395 418 264 976 518 369 936 719 694 880 114 393 356 458 31 556 371
528 294 720 548 562 988 790 227 630 656 356 472 753 259 139 877 57 810
215 268 33 853 161 260 749 268 592 103 160 369 285 877 598 928 712 912
276 607 616 806 936 543 915 276 714 443 149 541 817 261 995 495 148
917 447 459 210 139 395 719 879 518 305 214 505 323 9 939 486 564 569
719 322 28 296 503 433 31 137 843 261 285 562 303 770 81 857 587 273
932 905 76 543 447 529 407 487 518 276 522 505 912 845 734 73 730 343
304 711 170 719 972 551 400 60 220 208 151 585 223 993 80 76 33 322
999 386 476 413 988 651 276 251 752 704 954 834 986 378 454 486 629
174 586 154 551 489 192 558 927 294 712 288 584 251 220 826 630 19 810
607 856 103 483 358 393 44 55 606 656 972 429 678 714 454 692 49 183
610 472 305 522 766 929 364 664 538 880 840 879 806 602 955 97 17 952
774 802 157 354 977 952 798 284 593 406 896 771 722 807 628 85 600 736
166 70 615 913 462 534 494 906 673 455 814 178 960 202 46 72 813 353
952 36 107 140 314 485 165 574 680 235 676 575 191 731 252 690 785 848
93 871 854 239 908 0 863 784 801 309 315 338 945 650 797 246 123 197
10 832 805 739 437 163 63 571 804 684 180 663 666 428 910 944 677 729
769 865 688 907 421 624 376 345 943 507 793 196 166 655 816 792 360
168 312 456 596 974 632 521 872 181 257 332 466 866 563 95 761 18 295
491 292 949 620 397 279 533 924 422 108 97 213 493 125 502 571 754 532
427 331 675 134 451 184 58 25 437 991 914 892 573 693 807 422 618 534
85 771 535 901 758 796 613 825 351 789 685 948 225 159 409 708 2 431
442 940 252 467 201 715 531 241 72 377 800 794 152 519 83 321 67 301
242 935 937 132 365 560 373 604 709 549 935 85 262 218 570 952 234 104
```

```

133 765 95 554 916 205 854 841 350 726 469 82 15 172 723 13 797 557
504 566 557 835 477 502 238 908 550 737 978 402 670 695 90 69 577 821
809 671 297 987 757 757 699 966 628 882 1 653 579 421 536 478 886 320
186 225 205 26 123 298 803 308 798 97 96 125 287 563 638 679 792 112
326 498 199 684 473 167 196 233 849 284 653 521 120 420 340 623 758
374 229 709 797 787 627 605 583 627 194 603 876 247 444 439 462 926
895 637 32 242 963 951 688 346 958 567 699 567 474 872 775 761 436 300
424 410 773 77 502 792 658 974 779 397 269 969 640 633 292 737 599 844
159 568 336 318 613 552 334 337 878 91 865 867 807 293 228 176 123 967
430 228 119 812 435 706 411 70 842 450 534 392 687 677 48 771 512 441
280 211 802 618 6 17 956 12 271 345 432 180 446 764 199 886 360 829
372 134 581 862 605 874 100 34 319 68 342 129 618 795 783 690 655 913
994 938 7 524 412 696 949 911 6 324 593 567 654 881 317 577 940 560
409 145 823 718 697 675 968 819 116 772 412 51 213 695 772 278 555 898
686 946 867 88 515 942 115 535 705 423 647 354 768 77 134 895 135 53
979 312 815 894 238 235 901 166 859 363 922 589 808 367 124 382 97 457
820 339 590 798 200 16 113 898 185 85 131 731 804 430 670 187 115 445
978 825 899 333 962 791 941 862 926 131 775 789 609 632 899 987 850
580 444 516 613 25 726 888 337 334 158 595 842 38 419 838 974 158 325
206 266 721 733 362 380 893 367 79 425 870 510 78 815 760 102 578 705
351 128 221 360 336 52 738 526 15 657 250 763 426 77 570 90 410 841 89
731 175 52 844 17 106 373 99 180 635 931 604 718 171 327 501 887 153
279 660 979 47 891 425 449 740 633 328 724 146 91 98 451 63 317 803
779 926 442 380 330 437 622 8 4 286 603 72 216 672 822 65 385 702 410
779 279 653 354 778 118 624 689 646 724 902 739 669 965 368 275 182
473 829 509 246 994 736 811 982 553 847 155 456 560 51 789 231 135 909
144 974 793 717 611 469 878 219 846 594 760 921 1 84 864 860 961 56
545 473 1 526 75 979 204 419 815 992 54 918 167 228 2 805 372 153 971
391 40 87 862 682 436 344 347 221 321 498 466 452 793 432 898 153 701
535 370 576 559 771 27 464 199 665 409 318 289 412 463 604 475 883 466
188 217 159 307 136 546 15 784 859 506 526 873 79 552 794 517 591 384
916 84 780 849 573 685 827 142 109 456 827 846 396 27 665 958 484 703
286 844 547 202 69 172 875 108 408 540 795 876 514 546 5 51 828 507
776 446 66 180 110 462 141 243 741 689 121 180 872 608 342 542 733 577
778 14 22 421 43 897 747 105 631 212 900 798 710 950 793 579 340 914
741 696 82 648 964 156 996 723 866 366 946 122 838 613 217 457 697 321
713 383 713 490 559 911 698 94 691 762 554 757 401 90 922 544 329 582
994 899 937 682 716 688 280 654 870 725 107 708 842 299 202 557 807
751 778 555 567 155 426 886 705 980 536 626 783 892 140 765 79 591 897
634 540 247 913 451 516 162 408 561 58 913 782 788 883 970 591 512 91
213 85 732 767 112 882 266 130 68 209 4 795 888 68 588 331 179 61 360
953 725 807 614 384 61 612 820 670 289 967 909 645 763 314 618 547 568
743 252 724 754 848 516 589 357 761 581 497 572 627 855 77 667 314 348
485 797 173 549 440 765 682 299)

```

Решение 10.13.2

```

(defparameter w (loop repeat 1000 collect (random 1000)))
(defparameter v (loop repeat 1000 collect (random 1000)))

(defun -union (x y)
  (cond ((null x) y)
        ((null y) x)

```

```
((member (car x) y) (-union (cdr x) y))
(t (-union (cdr x) (cons (car x) y))))
```

```
> (time (-union w v))
```

```
Real time: 0.015629 sec.
```

```
Run time: 0.015625 sec.
```

```
Space: 1856 Bytes
```

```
(151 884 629 836 930 651 236 592 529 31 471 349 641 834 415 160 143
189 335 20 652 799 390 189 858 148 148 683 434 877 520 919 503 890 44
210 993 959 356 404 103 720 998 700 33 869 830 551 311 839 148 302 230
302 997 407 487 3 465 9 749 460 285 335 454 50 929 511 470 601 592 31
183 939 29 569 917 169 734 398 513 920 461 375 621 39 518 488 274 889
288 630 480 323 755 198 127 237 20 59 20 192 210 606 261 148 551 720
630 9 263 405 528 92 359 274 853 454 990 868 556 915 30 919 925 138
503 81 678 766 839 558 33 198 272 889 983 558 148 170 64 818 746 489
930 395 418 264 976 518 369 936 719 694 880 114 393 356 458 31 556 371
528 294 720 548 562 988 790 227 630 656 356 472 753 259 139 877 57 810
215 268 33 853 161 260 749 268 592 103 160 369 285 877 598 928 712 912
276 607 616 806 936 543 915 276 714 443 149 541 817 261 995 495 148
917 447 459 210 139 395 719 879 518 305 214 505 323 9 939 486 564 569
719 322 28 296 503 433 31 137 843 261 285 562 303 770 81 857 587 273
932 905 76 543 447 529 407 487 518 276 522 505 912 845 734 73 730 343
304 711 170 719 972 551 400 60 220 208 151 585 223 993 80 76 33 322
999 386 476 413 988 651 276 251 752 704 954 834 986 378 454 486 629
174 586 154 551 489 192 558 927 294 712 288 584 251 220 826 630 19 810
607 856 103 483 358 393 44 55 606 656 972 429 678 714 454 692 49 183
610 472 305 522 766 929 364 664 538 880 840 879 806 602 955 97 17 952
774 802 157 354 977 952 798 284 593 406 896 771 722 807 628 85 600 736
166 70 615 913 462 534 494 906 673 455 814 178 960 202 46 72 813 353
952 36 107 140 314 485 165 574 680 235 676 575 191 731 252 690 785 848
93 871 854 239 908 0 863 784 801 309 315 338 945 650 797 246 123 197
10 832 805 739 437 163 63 571 804 684 180 663 666 428 910 944 677 729
769 865 688 907 421 624 376 345 943 507 793 196 166 655 816 792 360
168 312 456 596 974 632 521 872 181 257 332 466 866 563 95 761 18 295
491 292 949 620 397 279 533 924 422 108 97 213 493 125 502 571 754 532
427 331 675 134 451 184 58 25 437 991 914 892 573 693 807 422 618 534
85 771 535 901 758 796 613 825 351 789 685 948 225 159 409 708 2 431
442 940 252 467 201 715 531 241 72 377 800 794 152 519 83 321 67 301
242 935 937 132 365 560 373 604 709 549 935 85 262 218 570 952 234 104
133 765 95 554 916 205 854 841 350 726 469 82 15 172 723 13 797 557
504 566 557 835 477 502 238 908 550 737 978 402 670 695 90 69 577 821
809 671 297 987 757 757 699 966 628 882 1 653 579 421 536 478 886 320
186 225 205 26 123 298 803 308 798 97 96 125 287 563 638 679 792 112
326 498 199 684 473 167 196 233 849 284 653 521 120 420 340 623 758
374 229 709 797 787 627 605 583 627 194 603 876 247 444 439 462 926
895 637 32 242 963 951 688 346 958 567 699 567 474 872 775 761 436 300
424 410 773 77 502 792 658 974 779 397 269 969 640 633 292 737 599 844
159 568 336 318 613 552 334 337 878 91 865 867 807 293 228 176 123 967
430 228 119 812 435 706 411 70 842 450 534 392 687 677 48 771 512 441
280 211 802 618 6 17 956 12 271 345 432 180 446 764 199 886 360 829
372 134 581 862 605 874 100 34 319 68 342 129 618 795 783 690 655 913
994 938 7 524 412 696 949 911 6 324 593 567 654 881 317 577 940 560
409 145 823 718 697 675 968 819 116 772 412 51 213 695 772 278 555 898
```

```

686 946 867 88 515 942 115 535 705 423 647 354 768 77 134 895 135 53
979 312 815 894 238 235 901 166 859 363 922 589 808 367 124 382 97 457
820 339 590 798 200 16 113 898 185 85 131 731 804 430 670 187 115 445
978 825 899 333 962 791 941 862 926 131 775 789 609 632 899 987 850
580 444 516 613 25 726 888 337 334 158 595 842 38 419 838 974 158 325
206 266 721 733 362 380 893 367 79 425 870 510 78 815 760 102 578 705
351 128 221 360 336 52 738 526 15 657 250 763 426 77 570 90 410 841 89
731 175 52 844 17 106 373 99 180 635 931 604 718 171 327 501 887 153
279 660 979 47 891 425 449 740 633 328 724 146 91 98 451 63 317 803
779 926 442 380 330 437 622 8 4 286 603 72 216 672 822 65 385 702 410
779 279 653 354 778 118 624 689 646 724 902 739 669 965 368 275 182
473 829 509 246 994 736 811 982 553 847 155 456 560 51 789 231 135 909
144 974 793 717 611 469 878 219 846 594 760 921 1 84 864 860 961 56
545 473 1 526 75 979 204 419 815 992 54 918 167 228 2 805 372 153 971
391 40 87 862 682 436 344 347 221 321 498 466 452 793 432 898 153 701
535 370 576 559 771 27 464 199 665 409 318 289 412 463 604 475 883 466
188 217 159 307 136 546 15 784 859 506 526 873 79 552 794 517 591 384
916 84 780 849 573 685 827 142 109 456 827 846 396 27 665 958 484 703
286 844 547 202 69 172 875 108 408 540 795 876 514 546 5 51 828 507
776 446 66 180 110 462 141 243 741 689 121 180 872 608 342 542 733 577
778 14 22 421 43 897 747 105 631 212 900 798 710 950 793 579 340 914
741 696 82 648 964 156 996 723 866 366 946 122 838 613 217 457 697 321
713 383 713 490 559 911 698 94 691 762 554 757 401 90 922 544 329 582
994 899 937 682 716 688 280 654 870 725 107 708 842 299 202 557 807
751 778 555 567 155 426 886 705 980 536 626 783 892 140 765 79 591 897
634 540 247 913 451 516 162 408 561 58 913 782 788 883 970 591 512 91
213 85 732 767 112 882 266 130 68 209 4 795 888 68 588 331 179 61 360
953 725 807 614 384 61 612 820 670 289 967 909 645 763 314 618 547 568
743 252 724 754 848 516 589 357 761 581 497 572 627 855 77 667 314 348
485 797 173 549 440 765 682 299)

```

Задача 10.14 *apply-list.lisp*

Определить функцию, которая применяет каждую функцию первого списка к соответствующему элементу второго списка и возвращает список, сформированный из результатов.

Решение 10.14.1

```

(defun apply-list (v w) (mapcar #'(lambda (a b) (funcall a b)) v w))

> (apply-list '(1- 1+ 1+) '(1 1 1))
(0 2 2)

```

Задача 10.15 *inter-evenp.lisp*

Найти четные числа в результирующем множестве, полученном из пересечения двух множеств.

Решение 10.15.1

```
(defun inter-evenp (w v)
  (remove-if #'oddp (intersection w v)))

> (inter-evenp '(1 2 3 4 5 6) '(4 5 6 7 8))
(4 6)
```

Задача 10.16 *sort-exclusive.lisp*

Реализовать функцию, которая осуществляет поиск и сортировку по возрастанию всех положительных чисел в результирующем множестве, полученном из симметричной разности двух множеств. В качестве множеств выступают списки из чисел.

Решение 10.16.1

```
(defun sort-exclusive (w v)
  (sort (set-exclusive-or w v) #'<))

> (sort-exclusive '(5 4 3) '(4 3 2))
(2 5)
```

Задача 10.17 *unite-unique.lisp*

Определить функцию создания списка, который является объединением двух списков, за исключением элементов, встречающихся в обоих списках.

Решение 10.17.1

```
(defun unite-unique (w v)
  (set-exclusive-or w v))

> (unite-unique '(1 2 3 4 5) '( 3 4 5 6 7))
(1 2 6 7)
```

Задача 10.18 *cartesian-sort.lisp*

Реализовать функцию, которая сортирует по возрастанию все числа в множестве, которое мы получаем в результате Декартового произведения двух множеств. Множества в виде списков чисел.

Решение 10.18.1

```
(defun cartesian-sort (w v)
  (mapcar #'(lambda (u) (sort u #'<))
    (mapcan (lambda (e) (mapcar (lambda (a) `(,a ,e)) w)) v)))

> (cartesian-sort '(2 2 2) '(1 1 1))
((1 2) (1 2) (1 2) (1 2) (1 2) (1 2) (1 2) (1 2) (1 2) (1 2))
```

Решение 10.18.2

```
(defun cartesian-sort (w v)
  (mapcar #'(lambda (u) (sort u #'<))
    (when w (nconc (mi (car w) v) (cartesian-sort (cdr w) v)))))

(defun mi (a v)
  (when v (cons (list a (car v)) (mi a (cdr v)))))

> (cartesian-sort '(2 2 2) '(1 1 1))
((1 2) (1 2) (1 2) (1 2) (1 2) (1 2) (1 2) (1 2) (1 2))
```

Решение 10.18.3

```
(defun sort-cartesian (w v)
  (mapcar #'(lambda (u) (sort u #'<))
    (cond ((null w) nil)
          (t (append (mi (car w) v)
                      (sort-cartesian (cdr w) v)))))

(defun mi (a v)
  (cond ((null v) nil)
        (t (cons (list a (car v)) (mi a (cdr v)))))

> (sort-cartesian '(2 2 2) '(1 1 1))
((1 2) (1 2) (1 2) (1 2) (1 2) (1 2) (1 2) (1 2) (1 2))
```

Задача 10.19 *our-set-exclusive-or.lisp*

Определить функцию высокого порядка для создания списка, который является объединением двух списков, за исключением элементов, встречающихся в обоих списках, т. е. симметричную разность.

Решение 10.19.1

```
(defun our-set-exclusive-or (w v)
  (nconc (unique w v) (unique v w)))

(defun unique (w v &optional ac &aux (a (car w)))
  (cond ((null w) (nreverse ac))
        ((or (member a v) (member a ac)) (unique (cdr w) v ac))
        ((unique (cdr w) v (cons a ac)))))

> (our-set-exclusive-or '(a b) '(b c))
(A C)
> (our-set-exclusive-or '(a b) '(b))
(A)
> (our-set-exclusive-or '(a b) '())
(A B)
> (our-set-exclusive-or '(a b) '(c))
(A B C)
```

Решение 10.19.2

```
(defun our-set-exclusive-or (w v)
  (nconc (unique w v) (unique v w)))

(defun unique (w v &optional ac)
  (if w
      (if (or (member (car w) v) (member (car w) ac))
          (unique (cdr w) v ac)
          (unique (cdr w) v (cons (car w) ac)))
      (nreverse ac)))
```

```
> (our-set-exclusive-or '(a b) '(b c))
(A C)
> (our-set-exclusive-or '(a b) '(b))
(A)
> (our-set-exclusive-or '(a b) '())
(A B)
> (our-set-exclusive-or '(a b) '(c))
(A B C)
```

Решение 10.19.3

```
(defun our-set-exclusive-or (w v)
  (nconc (unique w v) (unique v w)))

(defun unique (w v &optional ac &aux (a (car w)) (d (cdr w)))
  (if w
      (if (or (member a v) (member a ac))
          (unique d v ac)
          (unique d v (cons a ac)))
      (nreverse ac)))
```

```
> (our-set-exclusive-or '(a b) '(b c))
(A C)
> (our-set-exclusive-or '(a b) '(b))
(A)
> (our-set-exclusive-or '(a b) '())
(A B)
> (our-set-exclusive-or '(a b) '(c))
(A B C)
```

Решение 10.19.4

```
(defun our-set-exclusive-or (w v)
  (nconc (unique w v) (unique v w)))

(defun unique (w v)
  (loop for a in (remove-duplicates w)
        unless (member a v) collect a))
```

```
> (our-set-exclusive-or '(a b) '(b c))
(A C)
> (our-set-exclusive-or '(a b) '(b))
(A)
> (our-set-exclusive-or '(a b) '(c))
(A)
```

```
> (our-set-exclusive-or '(a b) '())
(A B)
> (our-set-exclusive-or '(a b) '(c))
(A B C)
```

Задача 10.20 *_intersection.lisp*

Определить функцию высшего порядка для создания списка из элементов, которые встречаются в обоих списках.

Решение 10.20.1

```
(defun _intersection (w v)
  (cond ((null w) nil)
        ((member (car w) v) (cons (car w) (_intersection (cdr w) v)))
        ((_intersection (cdr w) v))))

> (_intersection '(a s d f g h) '(q a w s e d r))
(A S D)
```

Решение 10.20.2

```
(defun _intersection (w v &aux (a (car w)))
  (when w (if (member a v)
              (cons a (_intersection (cdr w) v))
              (_intersection (cdr w) v))))

> (_intersection '(a s d f g h) '(q a w s e d r))
(A S D)
```

Решение 10.20.3

```
(defun _intersection (w v)
  (loop for a in w when (member a v) collect a))

> (_intersection '(a s d f g h) '(q a w s e d r))
(A S D)
```

Решение 10.20.4

```
(defun _intersection (w v)
  (remove-if-not #'(lambda (a) (member a v)) w))

> (_intersection '(a s d f g h) '(q a w s e d r))
(A S D)
```

Решение 10.20.5

```
(defun _intersection (w v &optional acc &aux (a (car w)))
  (cond ((null w) (nreverse acc))
        ((member a v) (_intersection (cdr w) v (cons a acc)))
        ((_intersection (cdr w) v acc))))
```



```
> (_intersection '(a s d f g h) '(q a w s e d r))
(A S D)
```

Решение 10.20.6

```
(defun _intersection (w v &optional acc &aux (a (car w)))
  (if w
      (_intersection (cdr w) v (if (member a v)
                                   (cons a acc)
                                   acc))
      (nreverse acc)))
```

```
> (_intersection '(a s d f g h) '(q a w s e d r))
(A S D)
```

Задача 10.21 shuffle.lisp

Даны два списка, построить третий по условию: даны списки L1 и L2 элементов типа char, построить список L3 по следующему правилу:

- если L1 является префиксом L2, то получить L3, исключив этот префикс и добавив его в качестве суффикса;
- в противном случае получить L3, присоединив к L1 список L2.

Решение 10.21

```
(defun shuffle (v w &aux (n (mismatch v w)))
  (if (zerop n) (nconc v w) (nconc (subseq w n) v)))

> (shuffle '(#\a #\b #\c) '(#\a #\b #\c #\d))
(#\d #\a #\b #\c)

> (shuffle '(#\z #\z #\z) '(#\a #\b #\c #\d))
(#\z #\z #\z #\a #\b #\c #\d)
```

Задача 10.22 sum-elms.lisp

Даны два списка одинаковой длины, элементы которых - числа. Найти список с элементами - суммами соответствующих элементов исходных списков.

Решение 10.22.1

```
(defun sum-elms (w v)
  (cond ((null w) nil)
        ((cons (+ (car w) (car v)) (sum-elms (cdr w) (cdr v))))))

> (sum-elms '(1 2) '(1 2))
(2 4)
```

Решение 10.22.2

```
(defun sum-elms (w v)
  (mapcar #'(lambda (w) (+ w v))
    w))

> (sum-elms '(1 2) '(1 2))
(2 4)
```

Задача 10.23 drop-set.lisp

Даны списки *w* и *v*. Реализовать рекурсивную функцию, которая удаляет из *w* все элементы-списки, соответствующие тому же множеству, что и *v*. Пример: *w*: '(1 (2 3) 5 (2 3 3 2) 4 (2 2 3) 6) , *v*: '(3 2 3 2) результат: '(1 5 4 6).

Решение 10.23.1

```
(defun drop-set (w v)
  (when w (if (or (atom (car w)) (set-difference (car w) v))
    (cons (car w) (drop-set (cdr w) v))
    (drop-set (cdr w) v))))

> (drop-set '((2 3 2 3) 8 (2 3) (2 7) (2) (2 2 3) 5) '(3 2 3 2))
(8 (2 7) 5)
```

Решение 10.23.2

```
(defun drop-set (w v)
  (when w ((lambda (a d)
    (if (or (atom a) (set-difference a v))
      (cons a (drop-set d v))
      (drop-set d v)))
    (car w) (cdr w))))

> (drop-set '((2 3 2 3) 8 (2 3) (2 7) (2) (2 2 3) 5) '(3 2 3 2))
(8 (2 7) 5)
```

Решение 10.23.3

```
(defun drop-sub (w v)
  (remove-if #'(lambda (s)
    (when (listp s)
      (subsetp s v)))
    w))

> (drop-set '((2 3 2 3) 8 (2 3) (2 7) (2) (2 2 3) 5) '(3 2 3 2))
(8 (2 7) 5)
```

Решение 10.23.4

```
(defun drop-set (w v)
  (delete-if #'(lambda (s)
    (and (listp s) (subsetp s v)))
    w))
```

```
> (drop-set '((2 3 2 3) 8 (2 3) (2 7) (2) (2 2 3) 5) '(3 2 3 2))
(8 (2 7) 5)
```

Решение 10.23.5*

* Переделать решение задачи, чтобы использовалась только форма let.

```
(defun drop-set (w v)
  (when w (let ((a (car w)) (d (cdr w)))
    (if (or (atom a) (nset-difference a v))
        (cons a (drop-set d v))
        (drop-set d v))))))
```

```
> (drop-set '((2 3 2 3) 8 (2 3) (2 7) (2) (2 2 3) 5) '(3 2 3 2))
(8 (2 7) 5)
```

Задача 10.24 bowl.lisp

Some day we will build a thinking machine.
It will be a truly intelligent machine. One
that can see and hear and speak. A machine
that will be proud of us.

From a Thinking Machines brochure

Имеются два кувшина объемом 9 литров и 4 литра. Можно заполнять кувшины из источника, выливать и переливать воду из одного кувшина в другой. Найти последовательность действий, при которой в большем кувшине останется 6 литров воды.

Решение 10.24.1 (автор - Alexey Dejneka, www.progz.ru)

```
(defun mv (w s c &aux (n (copy-seq s)))
  (destructuring-bind (p f) c
    (case p
      ((m) (let* ((a (- 1 f))
                  (d (min (- (svref w a) (svref n a))
                          (svref n f))))
              (decf (svref n f) d) (incf (svref n a) d)))
      ((f) (setf (svref n f) (svref w f)))
      ((e) (setf (svref n f) 0)))
    n))

(defun gn (w s l)
  (loop for c in '((m 0) (m 1) (f 0) (f 1) (e 0) (e 1))
    for n = (mv w s c) unless (eq c l) collect (cons n c)))

(defun fg (w v s r l)
  (cond ((equalp s v) (list s))
        ((zerop r) nil)
        ((loop for (n . m) in (gn w s l) do
          (let ((z (fg w v n (1- r) m)))
```

```

      (when z (return (cons s z)))))))))

(defun bowl (w v &optional (s #(0 0)))
  (loop for r from 0 thereis (fg w v s r nil)))

> (bowl #(9 4) #(6 0))
(#(0 0) #(9 0) #(5 4) #(5 0) #(1 4) #(1 0) #(0 1) #(9 1) #(6 4) #(6 0))
> (bowl #(8 5) #(3 0))
(#(0 0) #(8 0) #(3 5) #(3 0))
> (bowl #(5 3) #(1 0))
(#(0 0) #(0 3) #(3 0) #(3 3) #(5 1) #(0 1) #(1 0))
> (bowl #(8 3) #(7 0))
(#(0 0) #(8 0) #(5 3) #(5 0) #(2 3) #(2 0) #(0 2) #(8 2) #(7 3) #(7 0))
> (bowl #(5 2) #(1 0))
(#(0 0) #(5 0) #(3 2) #(3 0) #(1 2) #(1 0))
> (bowl #(8 5) #(6 0))
(#(0 0) #(8 0) #(3 5) #(3 0) #(0 3) #(8 3) #(6 5) #(6 0))

```

Задача 10.25 *remove-same-els.lisp*

Даны списки *lst1* и *lst2*. Реализовать функцию, которая удаляет из *lst1* все элементы-списки, которые соответствуют тому же множеству, что и *lst2*. Пример: для списков: *lst1* = (1 (2 2 3) 4 (3 2 3) 5), *lst2* = (2 2 3) результатом будет (1 4 (3 2 3) 5).

Решение 10.25.1

```

(defun remove-same-els (w v)
  (cond ((null w) nil)
        ((equal (car w) v) (remove-same-els (cdr w) v))
        ((cons (car w) (remove-same-els (cdr w) v)))))

> (remove-same-els '(1 (2 2 3) 4 (3 2 3) 5) '(2 2 3))
(1 4 (3 2 3) 5)

```

Задача 10.26 *add-pairs.lisp*

Сложить попарно порядковые элементы двух списков, например (1 2 3) и (1 2 3), чтобы получился список (2 4 6).

Решение 10.26.1

```

(defun add-pairs (w v)
  (cond ((or (null w) (null v)) nil)
        ((cons (+ (car w) (car v)) (add-pairs (cdr w) (cdr v))))))

> (add-pairs '(1 2 3) '(1 2 3))
(2 4 6)

```

Решение 10.26.2

```
(defun add-pairs (w v)
  (mapcar #'(lambda (w) (+ w v))
    w))

> (add-pairs '(1 2 3) '(1 2 3))
(2 4 6)
```

Решение 10.26.3

```
(defun add-pairs (w v)
  (loop for a in w for b in v collect (+ a b)))

> (add-pairs '(1 2 3) '(1 2 3))
(2 4 6)
```

Задача 10.27 intersect.lisp

Определите функцию, зависящую от двух аргументов *w* и *v*, являющихся списками, которая вычисляет список всех элементов, содержащихся одновременно и в списке *w*, и в списке *v*.

Решение 10.27.1

```
(defun intersect (w v &aux (a (car w)) (d (cdr w)))
  (cond ((null w) nil)
        ((member a v) (cons a (intersect d v)))
        ((intersect d v))))

> (intersect '(a b c) '(b c d))
(B C)
```

Задача 10.28 product-elms.lisp

Определите функцию, зависящую от двух аргументов *w* и *v*, являющихся списками, которая вычисляет список всех элементов, содержащихся одновременно и в списке *w*, и в списке *v*.

Решение 10.28.1

```
(defun product-elms (w v)
  (loop for a in w
        for b in v collect (* a b)))

> (product-elms '(1 2 3) '(2 2 2))
(2 4 6)
```

Решение 10.28.2

```
(defun product-elms (w v) (mapcar #'(lambda (w) (* w v))
  w))
```

```
> (product-elms '(1 2 3) '(2 2 2))
(2 4 6)
```

Задача 10.29 insert-ordered.lisp

Определите функцию, которая вставляет элементы одного упорядоченного списка в другой упорядоченный список, не нарушая упорядоченности.

Решение 10.29.1

```
(defun insert-ordered (w v)
  (cond ((null w) v)
        ((null v) w)
        ((< (car w) (car v))
         (cons (car w) (insert-ordered (cdr w) v)))
        ((cons (car v) (insert-ordered w (cdr v))))))

> (insert-ordered '(1 2 4 5 7) '(0 1 2 3 4 5 6 7))
(0 1 1 2 2 3 4 4 5 5 6 7 7)
```

Решение 10.29.2

```
> (merge 'list '(1 2 4 5 7) '(0 1 2 3 4 5 6 7) #'<)
(0 1 1 2 2 3 4 4 5 5 6 7 7)
```

Задача 10.30 exclusive-atoms.lisp

Объединить два одноуровневых списка, из которых нужно выбрать только атомы, которые не повторяются.

Решение 10.30.1

```
(defun exclusive-atoms (w v)
  (set-exclusive-or w v))

> (exclusive-atoms '(a b c) '(c b d))
(A D)
```

Решение 10.30.2

```
(defun exclusive-atoms (w v)
  (our-set-exclusive-or w v))

(defun our-set-exclusive-or (w v)
  (nconc (unique w v) (unique v w)))

(defun unique (w v &optional ac &aux (a (car w)))
  (cond ((null w) (nreverse ac))
        ((or (member a v) (member a ac)) (unique (cdr w) v ac))
        ((unique (cdr w) v (cons a ac)))))
```

```
> (exclusive-atoms '(a b c) '(c b d))
(A D)
```

Задача 10.31 exclusive-atomsxx.lisp

Объединить два многоуровневых списка, из которых нужно выбрать только атомы, которые не повторяются.

Решение 10.31.1

```
(defun exclusive-atomsxx (w v)
  (our-set-exclusive-or (flat w) (flat v)))

(defun our-set-exclusive-or (w v)
  (nconc (unique w v) (unique v w)))

(defun flat (w &optional acc)
  (cond ((null w) acc)
        ((atom w) (cons w acc))
        ((flat (car w) (flat (cdr w) acc)))))

(defun unique (w v &optional ac &aux (a (car w)))
  (cond ((null w) (nreverse ac))
        ((or (member a v) (member a ac)) (unique (cdr w) v ac))
        ((unique (cdr w) v (cons a ac)))))

> (exclusive-atomsxx '(1 2 3 3 (4 5 6) 7) '(7 (10 11 12) 13 14))
(1 2 3 3 4 5 6 10 11 12 13 14)
```

Решение 10.31.2

```
(defun exclusive-atomsxx (w v)
  (our-set-exclusive-or (flat w) (flat v)))

(defun our-set-exclusive-or (w v)
  (nconc (unique w v) (unique v w)))

(defun flat (w)
  (mapcan #'(lambda (a) (if (listp a) (flat a) `(,a))) w))

(defun unique (w v &optional ac &aux (a (car w)) (d (cdr w)))
  (if w
      (if (or (member a v) (member a ac))
          (unique d v ac)
          (unique d v (cons a ac)))
      (nreverse ac)))

> (exclusive-atomsxx '(1 2 3 3 (4 5 6) 7) '(7 (10 11 12) 13 14))
(1 2 3 3 4 5 6 10 11 12 13 14)
```

Решение 10.31.3

```

(defun exclusive-atomsxx (w v)
  (our-set-exclusive-or (flat w) (flat v)))

(defun our-set-exclusive-or (w v)
  (nconc (unique w v) (unique v w)))

(defun flat (w)
  (loop for a in w
        if (listp a) append (flat a)
        else collect a))

(defun unique (w v)
  (loop for a in (remove-duplicates w)
        unless (member a v) collect a))

> (exclusive-atomsxx '(1 2 3 3 (4 5 6) 7) '(7 (10 11 12) 13 14))
(1 2 3 3 4 5 6 10 11 12 13 14)

```

Решение 10.31.4

```

(defun exclusive-atomsxx (w v)
  (our-set-exclusive-or (flat w) (flat v)))

(defun our-set-exclusive-or (w v)
  (nconc (unique w v) (unique v w)))

(defun flat (w)
  (typecase w
    (cons (mapcan #'flat w)
      (t `(,w))))

(defun unique (w v)
  (loop for a in (remove-duplicates w)
        unless (member a v) collect a))

> (exclusive-atomsxx '(1 2 3 3 (4 5 6) 7) '(7 (10 11 12) 13 14))
(1 2 3 3 4 5 6 10 11 12 13 14)

```

Решение 10.31.5

```

(defun exclusive-atomsxx (w v)
  (our-set-exclusive-or (flat w) (flat v)))

(defun our-set-exclusive-or (w v)
  (nconc (unique w v) (unique v w)))

(defun flat (w)
  (if (atom w) `(,w) (mapcan #'flat w)))

(defun unique (w v)
  (loop for a in (remove-duplicates w)
        unless (member a v) collect a))

```



```
> (exclusive-atomsxx '(1 2 3 3 (4 5 6) 7) '(7 (10 11 12) 13 14))
(1 2 3 3 4 5 6 10 11 12 13 14)
```

Задача 10.32 max-pair.lisp

Дано два списка: w , v . Определить функцию, создающую список z , элементы которого определяются как $z_n = \max(w_n, v_n)$.

Решение 10.32.1

```
(defun max-pair (w v)
  (cond ((null w) nil)
        ((>= (car w) (car v))
         (cons (car w) (max-pair (cdr w) (cdr v))))
        ((cons (car v) (max-pair (cdr w) (cdr v))))))
```

```
> (max-pair '(1 2 3 4 5) '(2 1 33 11 0))
(2 2 33 11 5)
```

Решение 10.32.2

```
(defun max-pair (w v)
  (cond ((null w) nil)
        ((cons (if (>= (car w) (car v)) (car w) (car v))
                 (max-pair (cdr w) (cdr v))))))
```

```
> (max-pair '(1 2 3 4 5) '(2 1 33 11 0))
(2 2 33 11 5)
```

Решение 10.32.3

```
(defun max-pair (w v)
  (when w (cons (if (>= (car w) (car v)) (car w) (car v))
                 (max-pair (cdr w) (cdr v)))))
```

```
> (max-pair '(1 2 3 4 5) '(2 1 33 11 0))
(2 2 33 11 5)
```

Решение 10.32.4

```
(defun max-pair (w v)
  (when w (cons (max (car w) (car v))
                 (max-pair (cdr w) (cdr v)))))
```

```
> (max-pair '(1 2 3 4 5) '(2 1 33 11 0))
(2 2 33 11 5)
```

Решение 10.32.5

```
(defun max-pair (w v)
  (loop for a in w for b in v
        collect (max a b)))
```

```
> (max-pair '(1 2 3 4 5) '(2 1 33 11 0))
(2 2 33 11 5)
```

Решение 10.32.6

```
(defun max-pair (w v)
  (mapcar #'max w v))

> (max-pair '(1 2 3 4 5) '(2 1 33 11 0))
(2 2 33 11 5)
```

Задача 10.33 *_set-difference.lisp*

Определите функцию, зависящую от двух аргументов *w* и *v*, являющихся списками, которая вычисляет список всех элементов *w*, не содержащихся в *v*.

Решение 10.33.1

```
(defun _set-difference (a b)
  (cond ((null a) nil)
        ((member (car a) b) (_set-difference (cdr a) b))
        ((cons (car a) (_set-difference (cdr a) b)))))

> (_set-difference '(1 2 3 4 5) '(3 4 5 6 7 8))
(1 2)
```

Решение 10.33.2

```
(defun _set-difference (w v)
  (loop for a in w unless (member a v) collect a))

> (_set-difference '(1 2 3 4 5) '(3 4 5 6 7 8))
(1 2)
```

Задача 10.34 *a-merge.lisp*

Объединить два упорядоченных числовых списка без нарушения порядка.

Решение 10.34.1

```
(defun a-merge (w v)
  (cond ((null w) v) ((null v) w)
        ((<= (car w) (car v)) (cons (car w)
                                       (a-merge (cdr w) v)))
        ((cons (car v) (a-merge w (cdr v))))))

> (a-merge '(1 4 5 6 12) '(2 3 4 5))
(1 2 3 4 4 5 5 6 12)
```

Решение 10.34.2

```
(defun a-merge (w v)
  (cond ((null w) v) ((null v) w)
        ((> (car w)) (cons (car v)
                             (a-merge w (cdr v))))
        ((cons (car w) (a-merge (cdr w) v)))))

> (a-merge '(1 4 5 6 12) '(2 3 4 5))
(1 2 3 4 4 5 5 6 12)
```

Решение 10.34.3

```
(defun a-merge (w v)
  (merge 'list w v #'<))

> (a-merge '(1 4 5 6 12) '(2 3 4 5))
(1 2 3 4 4 5 5 6 12)
```

Задача 10.35 zip.lisp

Разработать функцию, объединяющую два списка в результирующий список, в котором чередуются элементы исходных списков. Например:

Вход: (1 2 3 4 5), (a b c)

Выход: (1 a 2 b 3 c 4 5)

Решение 10.35.1

```
(defun zip (w v)
  (cond ((null w) v) ((null v) w)
        ((cons (car w) (zip v (cdr w))))))

> (zip '(1 2 3 4) '(a b c d))
(1 A 2 B 3 C 4 D)
> (zip '(1 2 3 4 5) '(a b c))
(1 A 2 B 3 C 4 5)
```

Задача 10.36 inter-mass.lisp

Напишите функцию (f x y), которая возвращает список z - "пересечение" списков x и y, т.е. список, содержащий их общие элементы, причем кратность (число повторений) каждого элемента в списке z равняется максимуму из его кратностей (числа повторений) в списках x и y.

Решение 10.36.1

```
(defun inter-mass (w v)
  (reduce #'nconc
    (loop for a in (remove-duplicates (intersection w v))
      collect (loop for e from 1 to
                    (max (count a w) (count a v))
```

```

collect a))))
> (inter-mass '(1 2 3 3 4 5) '(3 3 3 4 5 6 7))
(3 3 3 4 5)

```

Решение 10.36.2

```

(defun inter-mass (w v)
  (loop for a in (remove-duplicates (intersection w v))
        nconc (loop repeat (max (count a w) (count a v))
                     collect a)))

> (inter-mass '(1 2 3 3 4 5) '(3 3 3 4 5 6 7))
(3 3 3 4 5)

```

Задача 10.37 *mix-zip.lisp*

The key to understanding Emacs is that it's all about efficiency, which includes economy of motion. Any trained musician will tell you that economy of motion is critical to becoming a world-class virtuoso. Any unnecessary motion is wasted energy and yields sloppy results. Using the mouse is almost always the worst possible violation of economy of motion, because you have to pick your hand up and fumble around with it. The mouse is a clumsy instrument, and Emacs gurus consider it a cache miss when they have to resort to using it. Compared to Emacs Wizards, graphical-IDE users are the equivalent of amateur musicians, pawing at their instrument with a sort of desperation. An IDE has blinking lights and pretty dialogs that you can't interact with properly (see Item 6), and gives newbies a nice comfortable sense of control. But that control is extremely crude, and all serious programmers prefer something that gives them more power.

Steve Yegge

W и *v* – списки, элементами которых являются атомы. Построить список, четные элементы которого содержат соответствующие четные элементы списка *w*, а нечетные – из *v*. Пример: (f '(1 2 3 4) '(4 3 2 1)) Результат: (1 3 3 1)

Решение 10.37.1

```

(defun mix-zip (w v)
  (when w (cons (car w) (mix-zip (cdr v) (cdr w)))))

```

```
> (mix-zip '(1 2 3 4) '(5 6 7 8))
(1 6 3 8)
```

Задача 10.38 max-atoms.lisp

Даны списки списков чисел *w* и *v*. Определить, какой из списков меньше по длинам списков.

Решение 10.38.1

```
(defun len (w)
  (cond ((null w) 0)
        ((atom (car w)) (1+ (len (cdr w))))
        ((+ (len (car w)) (len (cdr w))))))

(defun max-atoms (w v)
  (if (>= (len w) (len v)) w v))

> (max-atoms '((2 (3) (4 ((5)))))) '(1 (2 (3) 4) 5))
(1 (2 (3) 4) 5)
```

Задача 10.39 max-sum.lisp

Даны списки списков чисел *w* и *v*. Определить, какой из списков меньше по значениям элементов.

Решение 10.39.1

```
(defun sum (w)
  (cond ((null w) 0)
        ((atom (car w)) (+ (car w) (lenn (cdr w))))
        ((+ (lenn (car w)) (lenn (cdr w))))))

(defun max-sum (w v)
  (if (>= (sum w) (sum v)) w v))

> (max-sum '((2 (3) (4 ((5)))))) '(1 (2 (3) 4) 5))
(1 (2 (3) 4) 5)
```

Задача 10.40 extract-nths.lisp

Определить функцию, возвращающую элементы первого списка, заданные номерами во втором списке.

Решение 10.40.1

```
(defun extract-nths (w v &optional (n 1))
  (cond ((null w) nil)
        ((member n v) (cons (car w) (extract-nths (cdr w) v (1+ n))))
        ((extract-nths (cdr w) v (1+ n)))))
```

```
> (extract-nths '(a b c d e) '(2 4))
(B D)
```

Задача 10.41 *united.lisp*

Определить функцию, результатом которой является список, представляющий объединение многоуровневых списков в смысле объединения множеств.

Решение 10.41.1

```
(defun flat (w &optional acc)
  (cond ((null w) acc)
        ((atom w) (cons w acc))
        ((flat (car w) (flat (cdr w) acc)))))

(defun uni (w v &aux (a (car w)))
  (cond ((null w) (remove-duplicates v))
        ((or (member a (cdr w)) (member a v)) (uni (cdr w) v))
        ((cons a (uni (cdr w) v)))))

(defun united (w v) (uni (flat w) (flat v)))

> (united '((1) 3 4 (3)) '((3 4) 5 3))
(1 4 5 3)
```

Решение 10.41.2

```
(defun flat (w)
  (mapcan #'(lambda (a) (if (listp a) (flat a) `(",a"))) w))

(defun uni (w v &aux (a (car w)))
  (cond ((null w) (remove-duplicates v))
        ((or (member a (cdr w)) (member a v)) (uni (cdr w) v))
        ((cons a (uni (cdr w) v)))))

(defun united (w v)
  (uni (flat w) (flat v)))

> (united '((1) 3 4 (3)) '((3 4) 5 3))
(1 4 5 3)
```

Решение 10.41.3

```
(defun flat (w)
  (loop for a in w
        if (listp a) append (flat a)
        else collect a))

(defun uni (w v &aux (a (car w)))
  (cond ((null w) (remove-duplicates v))
        ((or (member a (cdr w)) (member a v)) (uni (cdr w) v))
        ((cons a (uni (cdr w) v)))))
```

```
(defun united (w v) (uni (flat w) (flat v)))

> (united '((1) 3 4 (3)) '((3 4) 5 3))
(1 4 5 3)
```

Решение 10.41.3

```
(defun flat (w)
  (typecase w
    (cons (mapcan #'flat w)
          (t `(:,w))))

(defun uni (w v &aux (a (car w)))
  (cond ((null w) (remove-duplicates v))
        ((or (member a (cdr w)) (member a v)) (uni (cdr w) v))
        ((cons a (uni (cdr w) v)))))

(defun united (w v) (uni (flat w) (flat v)))

> (united '((1) 3 4 (3)) '((3 4) 5 3))
(1 4 5 3)
```

Решение 10.41.4

```
(defun flat (w)
  (if (atom w) `(:,w) (mapcan #'flat w)))

(defun uni (w v &aux (a (car w)))
  (cond ((null w) (remove-duplicates v))
        ((or (member a (cdr w)) (member a v)) (uni (cdr w) v))
        ((cons a (uni (cdr w) v)))))

(defun united (w v) (uni (flat w) (flat v)))

> (united '((1) 3 4 (3)) '((3 4) 5 3))
(1 4 5 3)
```

Задача 10.42 *sum-pairs.lisp*

Определить функцию сложения элементов двух списков. Возвращает список, составленный из сумм элементов списков-параметров. Учесть, что переданные списки могут быть разной длины.

Решение 10.42.1

```
(defun sum-pairs (w v)
  (cond ((null w) v)
        ((null v) w)
        ((cons (+ (car w) (car v)) (sum-pairs (cdr w) (cdr v))))))

> (sum-pairs '(1 1) '(1 1 1))
```

(2 2 1)

Задача 10.43 *construct.lisp*

Вот два петуха,
 Которые будят того пастуха,
 Который бранится с коровницей строгою,
 Которая доит корову безрогую,
 Лягнувшую старого пса без хвоста,
 Который за шиворот треплет кота,
 Который пугает и ловит синицу,
 Которая часто ворует пшеницу,
 Которая в темном чулане хранится
 В доме,
 Который построил Джек.

Самуил Маршак «Дом, который построил Джек»

Определить функцию, формирующую список из второго списка по порядку номеров, представленных в первом списке.

Решение 10.43.1

```
(defun construct (w v)
  (cond ((null w) nil)
        ((cons (nth (car w) v) (construct (cdr w) v)))))

> (construct '(3 2 1 0) '(2 4 3 1))
(1 3 4 2)
```

Решение 10.43.2

```
(defun construct (w v)
  (when w (cons (nth (car w) v) (construct (cdr w) v))))

> (construct '(3 2 1 0) '(2 4 3 1))
(1 3 4 2)
```

Задача 10.44 *max-inter-seq.lisp*

Для двух списков найти общую последовательность элементов наибольшей длины (одну или несколько).

Решение 10.44.1

```
(defun max-inter-seq (w v &aux (z (arrange (inter-seq w v))))
  (most-wanted z (length (car z))))

(defun most-wanted (w n &aux (a (car w)))
  (when (and w (= (length a) n)) (cons a (most-wanted (cdr w) n))))
```



```

(defun inter-seq (w v)
  (intersection (all-seq w) (all-seq v) :test #'equalp))

(defun arrange (w)
  (sort w #'> :key #'length))

(defun all-seq (w)
  (reduce #'nconc (mapcar #'cage (chariot w))))

(defun chariot (w)
  (when w (cons w (chariot (cdr w)))))

(defun cage (w)
  (when w (cons w (cage (butlast w)))))

> (max-inter-seq '(1 b c d e 2 3 4 e f g) '(0 c d e f g h))
((C D E) (E F G))

```

Решение 10.44.2

```

(defun max-inter-seq
  (w v &aux (z (sort (inter-seq w v) #'> :key #'length)))
  (most-wanted z (length (car z))))

(defun most-wanted (w n &aux (a (car w)))
  (when (and w (= (length a) n)) (cons a (most-wanted (cdr w) n))))

(defun inter-seq (w v)
  (intersection (all-seq w) (all-seq v) :test #'equalp))

(defun all-seq (w)
  (reduce #'nconc (mapcar #'cage (loop for a on w collect a))))

(defun cage (w)
  (when w (cons w (cage (butlast w)))))

> (max-inter-seq '(1 b c d e 2 3 4 e f g) '(0 c d e f g h))
((C D E) (E F G))

```

Решение 10.44.3

```

(defun max-inter-seq (w v)
  (wanted (sort (inter-seq w v) #'> :key #'length)))

(defun wanted (w &optional (n (length (car w))) &aux (a (car w)))
  (when (and w (= (length a) n)) (cons a (wanted (cdr w) n))))

(defun inter-seq (w v)
  (intersection (all-seq w) (all-seq v) :test #'equalp))

(defun all-seq (w)
  (reduce #'nconc (mapcar #'cage (loop for a on w collect a))))

```

```
(defun cage (w)
  (when w (cons w (cage (butlast w)))))

> (max-inter-seq '(1 b c d e 2 3 4 e f g) '(0 c d e f g h))
((C D E) (E F G))
```

Задача 10.45 maxx-inter-seq.lisp

Для двух и более список найти общую последовательность элементов наибольшей длины (одну или несколько).

Решение 10.45.1

```
(defun maxx-inter-seq (&rest w)
  (wanted (sort (inter-seq w) #'> :key #'length)))

(defun wanted (w &optional (n (length (car w))) &aux (a (car w)))
  (when (and w (= (length a) n)) (cons a (wanted (cdr w) n))))

(defun inter-seq (w)
  (reduce #'(lambda (a b) (intersection a b :test #'equalp))
    (mapcar #'all-seq w)))

(defun all-seq (w)
  (reduce #'nconc (mapcar #'cage (loop for a on w collect a))))

(defun cage (w)
  (when w (cons w (cage (butlast w)))))

> (maxx-inter-seq '(b c d e f 2 e) '(0 c d e f g) '(b c d e))
((C D E))
```

Задача 10.46 deep-member.lisp

Определить функцию, проверяющую является ли список элементом другого списка.

Решение 10.46.1

```
(defun deep-member (v w)
  (cond ((null w) nil)
        ((equalp (car w) v) t)
        ((atom (car w)) (deep-member v (cdr w)))
        (t (or (deep-member v (car w)) (deep-member v (cdr w))))))

> (deep-member '(1 2 (7)) '(1 2 2 (1 2(7)) 1))
T
> (deep-member '(7) '(1 2 2 (1 2(7)) 1))
T
```

Решение 10.46.2

```
(defun deep-member (v w)
  (cond ((null w) nil)
        ((equalp (car w) v))
        ((atom (car w)) (deep-member v (cdr w)))
        ((or (deep-member v (car w)) (deep-member v (cdr w))))))

> (deep-member '(7) '(1 2 2 (1 2(7)) 1))
T
```

Задача 10.47 cartesian-product-sorted.lisp

Определить функцию, образующую упорядоченное декартово произведение двух заданных множеств '(b c a) и '(3 1 2).

Решение 10.47.1

```
(defun cartesian-product-sorted (w v)
  (cartesian-product (sort-lets w) (sort v #'<)))

(defun sort-lets (w)
  (loop for a across
        (sort (apply #'concatenate
                     'string
                     (mapcar #'string w))
              #'char<)
        collect (intern (string a))))

(defun cartesian-product (w v)
  (when w
    (append
     (maplist
      #'(lambda (e) (list (car w) (car e))) v)
     (cartesian-product (cdr w) v))))

> (cartesian-product-sorted '(b c a) '(3 1 2))
((A 1) (A 2) (A 3) (B 1) (B 2) (B 3) (C 1) (C 2) (C 3))
```

Решение 10.47.2

```
(defun cartesian-product-sorted (w v)
  (cartesian-product (sort-lets w) (sort v #'<)))

(defun sort-lets (w)
  (loop for a across
        (sort (apply #'concatenate
                     'string
                     (mapcar #'string w))
              #'char<)
        collect (intern (string a))))

(defun cartesian-product (w v)
```

```

    (apply
      #'nconc
      (mapcar
        #'(lambda (e) (mapcar #'(lambda (a) (list e a)) v)) w)))
> (cartesian-product-sorted '(b c a) '(3 1 2))
((A 1) (A 2) (A 3) (B 1) (B 2) (B 3) (C 1) (C 2) (C 3))

```

Решение 10.47.3

```

(defun cartesian-product-sorted (w v)
  (cartesian-product (sort-lets w) (sort v #'<)))

(defun sort-lets (w)
  (loop for a across
    (sort (apply #'concatenate
      'string
      (mapcar #'string w))
      #'char<)
    collect (intern (string a))))

(defun cartesian-product (w v)
  (mapcan (lambda (e) (mapcar (lambda (a) `(,a ,e)) w)) v))
> (cartesian-product-sorted '(b c a) '(3 1 2))
((A 1) (A 2) (A 3) (B 1) (B 2) (B 3) (C 1) (C 2) (C 3))

```

Решение 10.47.4

```

(defun cartesian-product-sorted (w v)
  (cartesian-product (sort-lets w) (sort v #'<)))

(defun sort-lets (w)
  (loop for a in
    (sort (mapcar #'string w) #'string<)
    collect (intern a)))

(defun cartesian-product (w v)
  (mapcan (lambda (e) (mapcar (lambda (a) `(,a ,e)) w)) v))
> (cartesian-product-sorted '(b c a) '(3 1 2))
((A 1) (A 2) (A 3) (B 1) (B 2) (B 3) (C 1) (C 2) (C 3))

```

Решение 10.47.5

```

(defun cartesian-product-sorted (w v)
  (mapcan #'(lambda (e)
    (mapcar #'(lambda (a) `(,a ,e))
      (loop for a in
        (sort (mapcar #'string w)
          #'string<)
        collect (intern a))))
    (sort v #'<)))

```

```
> (cartesian-product-sorted '(b c a) '(3 1 2))
((A 1) (A 2) (A 3) (B 1) (B 2) (B 3) (C 1) (C 2) (C 3))
```

Решение 10.47.6

```
(defun cartesian-product-sorted (w v)
  (cartesian-product (sort-lets w) (sort v #'<)))

(defun sort-lets (w)
  (loop for a in
        (sort (mapcar #'string w) #'string<)
        collect (intern a)))

(defun cartesian-product (w v)
  (loop for e in w nconc (loop for a in v collect (list e a))))

> (cartesian-product-sorted '(b c a) '(3 1 2))
((A 1) (A 2) (A 3) (B 1) (B 2) (B 3) (C 1) (C 2) (C 3))
```

Решение 10.47.7

```
(defun cartesian-product-sorted (w v)
  (cartesian-product
   (loop for a in
         (sort (mapcar #'string w) #'string<)
         collect (intern a))
   (sort v #'<)))

(defun cartesian-product (w v)
  (loop for e in w nconc (loop for a in v collect (list e a))))

> (cartesian-product-sorted '(b c a) '(3 1 2))
((A 1) (A 2) (A 3) (B 1) (B 2) (B 3) (C 1) (C 2) (C 3))
```

Решение 10.47.8

```
(defun cartesian-product-sorted (w v)
  (loop for e in (loop for a in (sort (mapcar #'string w) #'string<)
                        collect (intern a))
            nconc (loop for a in (sort v #'<)
                        collect (list e a))))

> (cartesian-product-sorted '(b c a) '(3 1 2))
((A 1) (A 2) (A 3) (B 1) (B 2) (B 3) (C 1) (C 2) (C 3))
```

Решение 10.47.9

```
(defun cartesian-product-sorted (w v)
  (loop for e in
        (loop for a in (sort (loop for c in w
                                collect (string c))
                              #'string<))
```

```

      collect (intern a))
    nconc (loop for a in (sort v #'<)
      collect (list e a))))

> (cartesian-product-sorted '(b c a) '(3 1 2))
((A 1) (A 2) (A 3) (B 1) (B 2) (B 3) (C 1) (C 2) (C 3))

```

Задача 10.48 bowl-sorted.lisp

Определить функцию, соединяющую два упорядоченных по неубыванию списка в упорядоченный по неубыванию список.

Решение 10.48.1

```

(defun bowl-sorted (w v &optional acc)
  (cond ((and (null w) (null v)) (reverse acc))
        ((null w) (nconc (reverse acc) v))
        ((null v) (nconc (reverse acc) w))
        ((<= (car w) (car v))
         (bowl-sorted (cdr w) v (cons (car w) acc)))
        ((bowl-sorted w (cdr v) (cons (car v) acc)))))

> (bowl-sorted '(1 1 2 5) '(0 3 4))
(0 1 1 2 3 4 5)

```

Решение 10.48.2

```

(defun bowl-sorted (w v)
  (cond ((null w) v)
        ((null v) w)
        ((< (car w) (car v))
         (cons (car w) (bowl-sorted (cdr w) v)))
        ((cons (car v) (bowl-sorted w (cdr v))))))

> (bowl-sorted '(1 1 2 5) '(0 3 4))
(0 1 1 2 3 4 5)

```

Решение 10.48.3

```

(defun bowl-sorted (w v)
  (cond ((null w) v)
        ((null v) w)
        ((< (car w) (car v)) (glue w v))
        (t (glue v w))))

(defun glue (w v)
  (cons (car w) (bowl-sorted (cdr w) v)))

> (bowl-sorted '(1 1 2 5) '(0 3 4))
(0 1 1 2 3 4 5)

```

Решение 10.48.4

```
(defun bowl-sorted (w v)
  (cond ((null w) v) ((null v) w)
        ((< (car w) (car v)) (glue w v) ((glue v w))))

(defun glue (w v) (cons (car w) (bowl-sorted (cdr w) v)))

> (bowl-sorted '(1 1 2 5) '(0 3 4))
(0 1 1 2 3 4 5)
```

Задача 10.49 sum-/lisp

Даны два списка: (1 2 3 4) и (4 5 6 7). Получить новый список $(1/4 + 2/5 + 3/6 + 4/7)$.

Решение 10.49.1

```
(defun sum-/ (w v) (reduce #'(lambda (a b) (+ (/ a b))) (mapcar #'(lambda (a b) (/ a b)) w v)))

> (sum-/ '(1 2 3 4) '(4 5 6 7))
241/140
> (reduce #'(lambda (a b) (+ (/ a b))) (mapcar #'(lambda (a b) (/ a b)) w v))
241/140
```

Решение 10.49.2

```
(defun sum-/ (w v)
  (butlast (mapcan #'(lambda (a b) (list (/ a b) '+)) w v)))

> (sum-/ '(1 2 3 4) '(4 5 6 7))
(1/4 + 2/5 + 1/2 + 4/7)
```

Решение 10.49.3

```
(defun sum-/ (w v)
  (butlast (mapcan #'(lambda (a b) `(/ a b) '+)) w v)))

> (sum-/ '(1 2 3 4) '(4 5 6 7))
(1/4 + 2/5 + 1/2 + 4/7)
```

Решение 10.49.4

```
(defun sum-/ (w v)
  (butlast (loop for a in w
                 for b in v
                 nconc (list (/ a b) '+)))))

> (sum-/ '(1 2 3 4) '(4 5 6 7))
(1/4 + 2/5 + 1/2 + 4/7)
```

Задача 10.50 matrices-tensor-product.lisp

Определить функцию, возвращающую тензорное произведение двух матриц.

Решение 10.50.1

```
(defun matrices-tensor-product (w v)
  (tensor (reduce #'nconc w) (reduce #'nconc v)))

(defun tensor (w v)
  (when w (cons (product v (car w)) (tensor (cdr w) v))))

(defun product (w n)
  (when w (cons (* (car w) n)
                 (product (cdr w) n))))

> (matrices-tensor-product '((1 2) (3 4)) '((11 22) (33 44)))
((11 22 33 44) (22 44 66 88) (33 66 99 132) (44 88 132 176))
```

Решение 10.50.2

```
(defun matrices-tensor-product (ww vv
                                &aux
                                (w (reduce #'nconc ww))
                                (v (reduce #'nconc vv)))
  (mapcar #'(lambda (a) (product v a)) w))

(defun product (w n)
  (when w (cons (* (car w) n) (product (cdr w) n))))

> (matrices-tensor-product '((1 2) (3 4)) '((11 22) (33 44)))
((11 22 33 44) (22 44 66 88) (33 66 99 132) (44 88 132 176))
```

Решение 10.50.3

```
(defun matrices-tensor-product (ww
                                Vv
                                &aux
                                (w (reduce #'nconc ww))
                                (v (reduce #'nconc vv)))
  (loop for a in w collect (product v a)))

(defun product (w n)
  (when w (cons (* (car w) n)
                 (product (cdr w) n))))

> (matrices-tensor-product '((1 2) (3 4)) '((11 22) (33 44)))
((11 22 33 44) (22 44 66 88) (33 66 99 132) (44 88 132 176))
```

Решение 10.50.4

```
(defun matrices-tensor-product (ww
```



```

                                Vv
                                &aux
                                (w (reduce #'nconc ww))
                                (v (reduce #'nconc vv)))
(mapcar
 #'(lambda (a)
   (mapcar
    #'(lambda (b) (* a b))
    v))
 w))

> (matrices-tensor-product '((1 2) (3 4)) '((11 22) (33 44)))
((11 22 33 44) (22 44 66 88) (33 66 99 132) (44 88 132 176))

```

Решение 10.50.5

```

(defun matrices-tensor-product (ww
                                vv
                                &aux
                                (w (reduce #'nconc ww))
                                (v (reduce #'nconc vv)))

  (loop for a in w collect
    (loop for b in v
      collect (* a b))))

> (matrices-tensor-product '((1 2) (3 4)) '((11 22) (33 44)))
((11 22 33 44) (22 44 66 88) (33 66 99 132) (44 88 132 176))

```

Задача 10.51 ordered.lisp

Дано два простых списка. Определить функцию, которая упорядочивает элементы первого списка так, чтобы атомы в нем шли в том же порядке, в котором эти атомы содержатся во втором списке. Элементы первого списка, не входящие во второй, не меняя взаимного расположения, перенести в конец результата.

Решение 10.51.1

```

(defun ordered (w v &aux (a (car v)) (c (count a w)))
  (cond ((null v) w)
        ((append (loop repeat c collect a)
                  (ordered (remove a w) (cdr v))))))

> (ordered '(1 2 c a b b a c) '(a b c))
(A A B B C C 1 2)
> (ordered '(a 1 c a b 7 8 b a c) '(c a b))
(C C A A B B 1 7 8)

```

Решение 10.51.2

```

(defun ordered (w v &aux (a (car v)) (c (count a w)))
  (if v

```

```

      (append (loop repeat c collect a)
              (ordered (remove a w) (cdr v)))
    w))

> (ordered '(1 2 c a b b a c) '(a b c))
(A A B B C C 1 2)
> (ordered '(a 1 c a b 7 8 b a c) '(c a b))
(C C A A A B B 1 7 8)

```

Решение 10.51.3

```

(defun ordered (w v &aux (a (car v)) (c (count a w)))
  (if v
      (nconc (make-list c :initial-element a)
              (ordered (delete a w) (cdr v)))
      w))

> (ordered '(1 2 c a b b a c) '(a b c))
(A A B B C C 1 2)
> (ordered '(a 1 c a b 7 8 b a c) '(c a b))
(C C A A A B B 1 7 8)

```

Задача 10.52 pair.lisp

Сформировать из двух списков один. Допустим, даны (a b c d e) и (1 2 3 4 5), тогда результат должен быть ((a 1) (b 2) (c 3) (d 4) (e 5)).

Решение 10.52.1

```

(defun pair (w v)
  (and w v (cons (list (car w) (car v)) (pair (cdr w) (cdr v)))))

> (pair '(1 2 3 4 5) '(a b c d e))
((1 A) (2 B) (3 C) (4 D) (5 E))
> (pair '(1 2 3 4 5) '(a b c d))
((1 A) (2 B) (3 C) (4 D))

```

Решение 10.52.2

```

(defun pair (w v) (loop for a in w for b in v collect (list a b)))

> (pair '(1 2 3 4 5) '(a b c d e))
((1 A) (2 B) (3 C) (4 D) (5 E))
> (pair '(1 2 3 4 5) '(a b c d))
((1 A) (2 B) (3 C) (4 D))

```

Решение 10.52.3

```

(defun pair (w v) (mapcar #'list w v))

> (pair '(1 2 3 4 5) '(a b c d e))
((1 A) (2 B) (3 C) (4 D) (5 E))

```

```
> (pair '(1 2 3 4 5) '(a b c d))
((1 A) (2 B) (3 C) (4 D))
```

Задача 10.53 drop-suffixes.lisp

Написать программу, исключающую из слов их окончания по словарю. Словарь окончаний представлять списком строк.

Решение 10.53.1

```
(defun string-right (s n)
  (subseq s (- (length s) n)))

(defun drop-suffix (s v &aux (m (length s)) (n (length (car v))))
  (cond ((null v) s)
        ((string= (string-right s n) (car v)) (subseq s 0 (- m n)))
        ((drop-suffix s (cdr v)))))

(defun drop-suffixes (w v &aux (v> (sort v #'string>)))
  (mapcar #'(lambda (s) (drop-suffix s v>)) w))

> (drop-suffixes '("nokia" "nokia") '("kia" "okia"))
("n" "n")
```

Задача 10.54 prefix-infix.lisp

Определить функцию, преобразующую польскую запись, заданную в форме списка, в арифметическое выражение. Например: вход: $(- * 3 2 5)$, выход: $(3 * 2 - 5)$.

Решение 10.54.1

```
(defun prefix-infix (w &optional ac &aux (a (car w)) (d (cdr w)))
  (if d
      (if (numberp a)
          (cons a (cons (car ac) (prefix-infix d (cdr ac))))
          (prefix-infix d (cons a ac)))
      w))

> (prefix-infix '(- * 3 2 5))
(3 * 2 - 5)
```

Решение 10.54.2

```
(defun prefix-infix (w &optional ac &aux (a (car w)) (d (cdr w)))
  (cond ((null d) w)
        ((numberp a)
         (cons a (cons (car ac) (prefix-infix d (cdr ac)))))
        ((prefix-infix d (cons a ac)))))

> (prefix-infix '(- * 3 2 5))
```

```
(3 * 2 - 5)
```

Спецификатор `&optional` - указывает на то, что далее следует необязательный (опциональный) параметр `ас`, который накапливает (аккумулятор) в виде списка арифметические знаки; спецификатор `&aux` - в списке параметров предваряет дополнительные (вспомогательные) переменные `а` ("голова" списка) и `д` ("хвост" списка).

Решение 10.54.3

```
(defun prefix-infix (w &aux (p (position-if #'numberp w)))
  (mix-elements (nthcdr p w) (reverse (subseq w 0 p))))

(defun mix-elements (w v)
  (if w (cons (car w) (mix-elements v (cdr w)))
    v))

> (prefix-infix '(- * 3 2 5))
(3 * 2 - 5)
```

Решение 10.54.4

```
(defun prefix-infix (w &aux (p (position-if #'numberp w)))
  (nconc (mapcan #'list (nthcdr p w) (reverse (subseq w 0 p)))
    (last w)))

> (prefix-infix '(- * 3 2 5))
(3 * 2 - 5)
```

Задача 10.55 *drop-index.lisp*

Writing DSL in Lisp - putting parentheses around the specification and make it run.

Rainer Joswig

Определить функцию, удаляющую элементы списка по номерам второго списка.

Решение 10.55.1

```
(defun drop-index (w v &optional (n 0))
  (cond ((null w) nil)
        ((null v) w)
        ((= (car v) n) (drop-index (cdr w) (cdr v) (1+ n)))
        ((cons (car w) (drop-index (cdr w) v (1+ n))))))

> (drop-index '(a b c d e) '(1 2 4))
(A D)
```

Решение 10.55.2

```
(defun drop-index (w v &optional (n 0))
  (cond ((or (null w) (null v)) w)
        ((= (car v) n) (drop-index (cdr w) (cdr v) (1+ n)))
        ((cons (car w) (drop-index (cdr w) v (1+ n))))))

> (drop-index '(a b c d e) '(1 2 4))
(A D)
```

Решение 10.55.3

```
(defun drop-index (w v) (drop w (sort (copy-list v) #'<)))

(defun drop (w v &optional (n 0))
  (cond ((or (null w) (null v)) w)
        ((= (car v) n) (drop (cdr w) (cdr v) (1+ n)))
        ((cons (car w) (drop (cdr w) v (1+ n))))))

> (drop-index '(a b c d e) '(1 2 4))
(A D)
> (drop-index '(a b c d e) '(4 1 2))
(A D)
```

Решение 10.55.4

```
(defun drop-index (w v) (drop w (sort (copy-list v) #'<)))

(defun drop (w v &optional (n 0))
  (cond ((or (null w) (null v)) w)
        ((= (car v) n) (drop (cdr w) (cdr v) (1+ n)))
        ((cons (car w) (drop (cdr w) v (1+ n))))))

> (drop-index '(a b c d e) '(1 2 4))
(A D)
> (drop-index '(a b c d e) '(4 1 2))
(A D)
```

Решение 10.55.4

```
(defun drop-index (w v) (drop w (sort (copy-list v) #'<)))

(defun drop (w v &optional ac (n 0))
  (if (or (null w) (null v))
      (append (nreverse ac) w)
      (drop (cdr w)
            (if (= (car v) n) (cdr v) v)
            (if (= (car v) n) ac (cons (car w) ac))
            (1+ n))))

> (drop-index '(a b c d e) '(1 2 4))
(A D)
> (drop-index '(a b c d e) '(4 1 2))
(A D)
```

Задача 10.56 subtract.lisp

Определить функцию вычитания десятичных чисел, представленных в виде списков.

Решение 10.56.1

```
(defun subtract (w v &aux (n (- (int w) (int v))))
  (unless (and (null w) (null v))
    (if (minusp n)
        (cons '- (int-digits (- n)))
        (int-digits n))))

(defun int-digits (n)
  (loop with r do (setf (values n r) (truncate n 10))
    collect r into ac until (zerop n)
    finally (return (nreverse ac))))

(defun int (w &optional (n (1- (length w))))
  (if (minusp n)
      0
      (+ (* (car w)
             (expt 10 n))
         (int (cdr w) (1- n)))))

> (subtract '(5 6 7) '(7 4 5))
(- 1 7 8)
> (subtract '(5 6 7) '(1 4 5))
(4 2 2)
> (subtract '(5 6 7) '())
(5 6 7)
> (subtract '() '(1 4 5))
(- 1 4 5)
> (subtract '() '())
NIL
```

Решение 10.56.2

```
(defun subtract (w v &aux (n (- (int w) (int v))))
  (unless (and (null w) (null v))
    (if (minusp n)
        (cons '- (int-digits (- n)))
        (int-digits n))))

(defun int-digits (n &optional ac)
  (if (zerop n)
      ac
      (int-digits (truncate n 10)
                   (cons (rem n 10) ac))))

(defun int (w &optional (n (1- (length w))))
  (if (minusp n)
```

```

0
(+ (* (car w)
      (expt 10 n))
  (int (cdr w) (1- n))))

> (subtract '(5 6 7) '(7 4 5))
(- 1 7 8)
> (subtract '(5 6 7) '(1 4 5))
(4 2 2)
> (subtract '(5 6 7) '())
(5 6 7)
> (subtract '() '(1 4 5))
(- 1 4 5)
> (subtract '() '())
NIL

```

Задача 10.57 remove-elms.lisp

Определить функцию удаления из многоуровневого списка *w* всех элементов списка *v* на всех уровнях.

Решение 10.57.1

```

(defun remove-elms (w v)
  (loop for a in w
        if (listp a) collect (remove-elms a v)
        else unless (member a v) collect a))

> (remove-elms '(1 2 (3 (4 5) 6) 7 8) '(3 6 8))
(1 2 ((4 5)) 7)

```

Задача 10.58 cs.lisp

Даны два списка одинаковой структуры. Построить список такой же структуры, состоящий из пар элементов исходных списков.

Решение 10.58.1

```

(defun cs (w v)
  (cond ((null w) nil)
        ((atom (car w)) (cons (list (car w) (car v))
                               (cs (cdr w) (cdr v))))
        (t (cons (cs (car w) (car v)) (cs (cdr w) (cdr v))))))

> (cs '(a (b) ((c) d)) '(1 (2) ((3) 4)))
((A 1) ((B 2)) (((C 3) (D 4))))

```

Решение 10.58.2

```

(defun cs (w v)
  (when w (cons (if (atom (car w)) (list (car w) (car v))

```

```

      (cs (car w) (car v)))
    (cs (cdr w) (cdr v))))))

```

```

> (cs '(a (b) ((c) d)) '(1 (2) ((3) 4)))
((A 1) ((B 2)) (((C 3)) (D 4)))

```

Решение 10.58.3

```

(defun cs (w v &aux (a (car w)) (b (car v)))
  (when w (cons (if (atom a) `,(a ,b) (cs a b))
    (cs (cdr w) (cdr v)))))

```

```

> (cs '(a (b) ((c) d)) '(1 (2) ((3) 4)))
((A 1) ((B 2)) (((C 3)) (D 4)))

```

Решение 10.58.4

```

(defun cs (w v)
  (loop for a in w for b in v
    if (atom a) collect `,(a ,b)
    else collect (cs a b)))

```

```

> (cs '(a (b) ((c) d)) '(1 (2) ((3) 4)))
((A 1) ((B 2)) (((C 3)) (D 4)))

```

Решение 10.58.5

```

(defun cs (w v)
  (loop for a in w for b in v collect
    (if (atom a) `,(a ,b) (cs a b))))

```

```

> (cs '(a (b) ((c) d)) '(1 (2) ((3) 4)))
((A 1) ((B 2)) (((C 3)) (D 4)))

```

Решение 10.58.6

```

(defun cs (w v)
  (mapcar #'(lambda (a b) (if (atom a) `,(a ,b) (cs a b))) w v))

```

```

> (cs '(a (b) ((c) d)) '(1 (2) ((3) 4)))
((A 1) ((B 2)) (((C 3)) (D 4)))

```

Решение 10.58.7

```

(defun cs (w v) (mapcar #'cs- w v))

(defun cs- (a b) (if (atom a) `,(a ,b) (cs a b)))

```

```

> (cs '(a (b) ((c) d)) '(1 (2) ((3) 4)))
((A 1) ((B 2)) (((C 3)) (D 4)))

```

Решение 10.58.8


```
(defun cs (w v) (loop for a in w for b in v collect (cs- a b)))

(defun cs- (a b) (if (atom a) `(,a ,b) (cs a b)))

> (cs '(a (b) ((c) d)) '(1 (2) ((3) 4)))
((A 1) ((B 2)) (((C 3)) (D 4)))
```

Решение 10.58.9

```
(defun cs (w v &aux (a (car w)) (b (car v)))
  (when w (cons (cs- a b) (cs (cdr w) (cdr v)))))

(defun cs- (a b) (if (atom a) `(,a ,b) (cs a b)))

> (cs '(a (b) ((c) d)) '(1 (2) ((3) 4)))
((A 1) ((B 2)) (((C 3)) (D 4)))
```

Задача 10.59 srtr.lisp

Отсортировать в порядке убывания отрицательные элементы списка, полученного в результате пересечения множеств двух других списков, оставив на месте неотрицательные элементы.

Решение 10.59.1

```
(defun srtr (w v &aux (z (intersection w v)))
  (sr z (sort (remove-if-not #'minusp z) #>)))

(defun sr (w v)
  (cond ((null v) w)
        ((minusp (car w)) (cons (car v) (sr (cdr w) (cdr v))))
        ((cons (car w) (sr (cdr w) v)))))

> (intersection '(-1 32 -57 0 -4 -2 41 -41 -5 -7 12 -12 64 128 -256 -
14 -42) '(-1 -12 -2 4 6 8 -4 -256 -57 10 0 41 12))
(-1 -57 0 -4 -2 41 12 -12 -256)
> (srtr '(-1 32 -57 0 -4 -2 41 -41 -5 -7 12 -12 64 128 -256 -14 -42)
'(-1 -12 -2 4 6 8 -4 -256 -57 10 0 41 12))
(-1 -2 0 -4 -12 41 12 -57 -256)
```

Решение 10.59.2

```
(defun srtr (w v &aux (z (intersection w v)))
  (sr z (sort (remove-if-not #'minusp z) #>)))

(defun sr (w v)
  (when w (if (minusp (car w))
              (cons (car v) (sr (cdr w) (cdr v)))
              (cons (car w) (sr (cdr w) v)))))

> (intersection '(-1 32 -57 0 -4 -2 41 -41 -5 -7 12 -12 64 128 -256 -
14 -42) '(-1 -12 -2 4 6 8 -4 -256 -57 10 0 41 12))
```

```
(-1 -57 0 -4 -2 41 12 -12 -256)
> (srrt '(-1 32 -57 0 -4 -2 41 -41 -5 -7 12 -12 64 128 -256 -14 -42)
'(-1 -12 -2 4 6 8 -4 -256 -57 10 0 41 12))
(-1 -2 0 -4 -12 41 12 -57 -256)
```

Решение 10.59.3

```
(defun srrt (w v &aux (z (intersection w v)))
  (sr z (sort (remove-if-not #'minusp z) #'>)))

(defun sr (w v &aux (a (car w)) (d (cdr w)))
  (when w (if (minusp a)
              (cons (car v) (sr d (cdr v)))
              (cons a (sr d v)))))

> (intersection '(-1 32 -57 0 -4 -2 41 -41 -5 -7 12 -12 64 128 -256 -
14 -42) '(-1 -12 -2 4 6 8 -4 -256 -57 10 0 41 12))
(-1 -57 0 -4 -2 41 12 -12 -256)
> (srrt '(-1 32 -57 0 -4 -2 41 -41 -5 -7 12 -12 64 128 -256 -14 -42)
'(-1 -12 -2 4 6 8 -4 -256 -57 10 0 41 12))
(-1 -2 0 -4 -12 41 12 -57 -256)
```

Задача 10.60 *deep-set-difference.lisp*

Опишите функцию, аргументами которой являются два списка, а результатом — множество, содержащее атомы, принадлежащие только первому списку, учитывая все атомы всех подсписков обоих списков.

Решение 10.60.1

```
(defun deep-set-difference (w v)
  (our-set-difference (flatten w) (flatten v)))

(defun flatten (w &optional ac)
  (cond ((null w) ac)
        ((atom w) (cons w ac))
        ((flatten (car w) (flatten (cdr w) ac)))))

(defun our-set-difference (w v)
  (cond ((null w) nil)
        ((member (car w) v) (our-set-difference (cdr w) v))
        ((cons (car w) (our-set-difference (cdr w) v)))))

> (deep-set-difference '(a e ((b) c)) '(a (b (((c)))) d))
(E)
```

Решение 10.60.2

```
(defun deep-set-difference (w v)
  (our-set-difference (flatten w) (flatten v)))

(defun flatten (w)
```

```

(loop for a in w
      if (listp a) append (flatten a)
      else collect a))

(defun our-set-difference (w v)
  (loop for a in w unless (member a v) collect a))

> (deep-set-difference '(a e ((b)) c)) '(a (b (((c)))) d))
(E)

```

Решение 10.60.3

```

(defun deep-set-difference (w v)
  (set-difference (flatten w) (flatten v)))

(defun flatten (w)
  (typecase w
    (cons (mapcan #'flatten w))
    (t `(,w))))

> (deep-set-difference '(a e ((b)) c)) '(a (b (((c)))) d))
(E)

```

Решение 10.60.4

```

(defun deep-set-difference (w v)
  (set-difference (flatten w) (flatten v)))

(defun flatten (w)
  (if (atom w) `(,w) (mapcan #'flatten w)))

> (deep-set-difference '(a e ((b)) c)) '(a (b (((c)))) d))
(E)

```

Задача 10.61 *deep-intersection.lisp*

What's special with Lisp is the line between data and code is very thin, and one can be transformed into the other very easily, without using special syntax.

Mark Miller

Реализовать функцию пересечения двух списков, учитывая все атомы всех подсписков обоих списков.

Решение 10.61.1

```

(defun deep-intersection (w v)
  (-intersection (flatten w) (flatten v)))

(defun flatten (w &optional ac)

```

```

(cond ((null w) ac)
      ((atom w) (cons w ac))
      ((flatten (car w) (flatten (cdr w) ac)))))

(defun -intersection (w v)
  (cond ((null w) nil)
        ((member (car w) v) (cons (car w) (-intersection (cdr w) v)))
        (t (-intersection (cdr w) v))))

> (deep-intersection '(a (b) ((c)) d) '(b (c) ((d)) e))
(B C D)

```

Решение 10.61.2

```

(defun deep-intersection (w v)
  (-intersection (flatten w) (flatten v)))

(defun flatten (w)
  (loop for a in w
        if (listp a) append (flatten a)
        else collect a))

(defun -intersection (w v)
  (cond ((null w) nil)
        ((member (car w) v) (cons (car w) (-intersection (cdr w) v)))
        (t (-intersection (cdr w) v))))

> (deep-intersection '(a (b) ((c)) d) '(b (c) ((d)) e))
(B C D)

```

Решение 10.61.3

```

(defun deep-intersection (w v)
  (-intersection (flatten w) (flatten v)))

(defun flatten (w)
  (typecase w
    (cons (mapcan #'flatten w))
    (t `(,w))))

(defun -intersection (w v)
  (cond ((null w) nil)
        ((member (car w) v) (cons (car w) (-intersection (cdr w) v)))
        (t (-intersection (cdr w) v))))

> (deep-intersection '(a (b) ((c)) d) '(b (c) ((d)) e))
(B C D)

```

Решение 10.61.4

```

(defun deep-intersection (w v)
  (-intersection (flatten w) (flatten v)))

```

```
(defun flatten (w)
  (if (atom w) `(,w) (mapcan #'flatten w)))

(defun -intersection (w v)
  (cond ((null w) nil)
        ((member (car w) v) (cons (car w) (-intersection (cdr w) v)))
        (t (-intersection (cdr w) v))))

> (deep-intersection '(a (b) ((c)) d) '(b (c) ((d)) e))
(B C D)
```

Задача 10.62 *our-nconc.lisp*

Определить спискоразрушающую функцию, соединяющую списки.

Решение 10.62.1

```
(defun our-nconc (w v)
  (if w (progn (rplacd (last w) v) w) v))

> (our-nconc '(1 2 3) '(4 5 6))
(1 2 3 4 5 6)
```

Задача 10.63 *bind.lisp*

I have changed multiple editors from real crappy ones like notepad, editplus to modern ones like sublime. Sublime is great but I wanted to try emacs or vim. Main motivation for me to think about changing editor came from my colleague who was blazing fast in editing code with vim. So, I first started trying vim. I failed sticking to vim as the learning was steep and I could not dedicate enough time to learn vi as there are frustrating moments initially if your work takes longer just because you changed editor. So, I gave up using vim and started using sublime again. But, after using sublime again for few months, I wanted to try emacs. I have been using emacs for an year now.

~

Определить функцию, объединяющую атомы двух одноуровневых списков в один ассоциативный список ((a1.b1) (a2.b2) (a3.b3)).

Решение 10.63.1

```
(defun bind (w v)
  (cond ((or (null w) (null v)) nil)
        (t (cons (cons (car w) (car v)) (bind (cdr w) (cdr v))))))
```

```
> (bind '(a b c) '(1 2 3))
((A . 1) (B . 2) (C . 3))
```

Решение 10.63.2

```
(defun bind (w v)
  (when w (cons (cons (car w) (car v)) (bind (cdr w) (cdr v)))))

> (bind '(a b c) '(1 2 3))
((A . 1) (B . 2) (C . 3))
```

Решение 10.63.3

```
(defun bind (w v)
  (cond ((null w) nil)
        ((null w) nil)
        (t (cons (cons (car w) (car v)) (bind (cdr w) (cdr v))))))

> (bind '(a b c) '(1 2 3 4))
((A . 1) (B . 2) (C . 3))
```

Решение 10.63.4

```
(defun bind (w v)
  (when (and w v)
    (cons (cons (car w) (car v)) (bind (cdr w) (cdr v)))))

> (bind '(a b c) '(1 2 3 4))
((A . 1) (B . 2) (C . 3))
```

Задача 10.64 *deep-bind.lisp*

Определить функцию, объединяющую атомы двух многоуровневых списков в один ассоциативный список ((a1.b1) (a2.b2) (a3.b3)).

Решение 10.64.1

```
(defun deep-bind (w v) (bind (flatten w) (flatten v)))

(defun bind (w v)
  (cond ((null w) nil)
        ((null w) nil)
        (t (cons (cons (car w) (car v)) (bind (cdr w) (cdr v)))))

(defun flatten (w &optional acc)
  (cond ((null w) acc)
        ((atom w) (cons w acc))
        ((flatten (car w) (flatten (cdr w) acc)))))

> (deep-bind '(a ((b) c)) '(1 (2 ((3)) 4)))
((A . 1) (B . 2) (C . 3))
```

Решение 10.64.2

```

(defun deep-bind (w v) (bind (flatten w) (flatten v)))

(defun bind (w v)
  (when (and w v) (cons (cons (car w) (car v))
                        (bind (cdr w) (cdr v)))))

(defun flatten (w &optional acc)
  (cond ((null w) acc)
        ((atom w) (cons w acc))
        ((flatten (car w) (flatten (cdr w) acc)))))

> (deep-bind '(a ((b) c)) '(1 (2 (((3)) 4))))
((A . 1) (B . 2) (C . 3))

```

Решение 10.64.3

```

(defun deep-bind (w v) (bind (flatten w) (flatten v)))

(defun bind (w v)
  (unless (or (null w) (null v))
    (cons (cons (car w) (car v)) (bind (cdr w) (cdr v)))))

(defun flatten (w &optional acc)
  (cond ((null w) acc)
        ((atom w) (cons w acc))
        ((flatten (car w) (flatten (cdr w) acc)))))

> (deep-bind '(a ((b) c)) '(1 (2 (((3)) 4))))
((A . 1) (B . 2) (C . 3))

```

Решение 10.64.4

```

(defun deep-bind (w v) (bind (flatten w) (flatten v)))

(defun bind (w v)
  (loop for a in w for b in v collect (cons a b)))

(defun flatten (w)
  (loop for a in w
    if (listp a) append (flatten a)
    else collect a))

> (deep-bind '(a ((b) c)) '(1 (2 (((3)) 4))))
((A . 1) (B . 2) (C . 3))

```

Решение 10.64.5

```

(defun deep-bind (w v)
  (bind (flatten w) (flatten v)))

```

```
(defun bind (w v) (mapcar #'cons w v))

(defun flatten (w)
  (loop for a in w
        if (listp a) append (flatten a)
        else collect a))

> (deep-bind '(a ((b) c)) '(1 (2 ((3)) 4))))
((A . 1) (B . 2) (C . 3))
```

Решение 10.64.6

```
(defun deep-bind (w v)
  (mapcar #'cons (flatten w) (flatten v)))

(defun flatten (w)
  (loop for a in w
        if (listp a) append (flatten a)
        else collect a))

> (deep-bind '(a ((b) c)) '(1 (2 ((3)) 4))))
((A . 1) (B . 2) (C . 3))
```

Задача 10.65 *compose.lisp*

Lisp is a marvel of simplicity. Everything is a list. Such a brilliant idea, with important consequences. Data and code are equivalent. This means parsing is very easy, and it allows powerful macros or making domain specific languages. Everything is a list operation. Operations compose to form more complex operations. Composability allows reuse. The base language is ridiculously simple, which makes it hugely powerful.

Alexandru Bolboaca

Определить функцию, которая для заданных *w* и *v* возвращает список, содержащий первые и последние элементы *w* и *v*. Порядок перечисления определяется вторым элементом *v*: если это число, то в голове списка должны находиться элементы *w*, иначе элементы *v*.

Решение 10.65.1

```
(defun compose (w
               v
               &aux
               (a (cons (car w) (last w)))
               (b (cons (car v) (last v))))
  (when (and w v)
    (if (numberp (cadr v)) (append a b)
```



```

(append b a))))

> (compose '(a b c) '(d e f))
(D F A C)
> (compose '(a b c) '(d 0 f))
(A C D F)
> (compose '() '())
NIL

```

Задача 10.66 *set-distinction.lisp*

Определить функцию, аргументами которой являются два списка, а результатом — список, содержащий элементы первого списка (в том числе входящие в подсписки), не принадлежащие второму списку (в том числе входящие в подсписки).

Решение 10.66.1

```

(defun set-distinction (w v) (drop w (flatten v)))

(defun drop (w v &aux (a (car w)) (d (cdr w)))
  (cond ((null w) nil)
        ((listp a) (cons (drop a v) (drop d v)))
        ((member a v) (drop d v))
        ((cons a (drop d v)))))

(defun flatten (w &optional ac)
  (cond ((null w) ac)
        ((atom w) (cons w ac))
        ((flatten (car w) (flatten (cdr w) ac)))))

> (set-distinction '(((0) ((1) 2)) (3)) '(((0) (3)) (7)))
(1 2)

```

Решение 10.66.2

```

(defun set-distinction (w v) (drop w (flatten v)))

(defun drop (w v &aux (a (car w)) (d (cdr w)))
  (cond ((null w) nil)
        ((listp a) (cons (drop a v) (drop d v)))
        ((member a v) (drop d v))
        ((cons a (drop d v)))))

(defun flatten (w)
  (loop for a in w
        if (listp a) append (flatten a)
        else collect a))

> (set-distinction '(((0) ((1) 2)) (3)) '(((0) (3)) (7)))
(1 2)

```

Решение 10.66.3

```

(defun drop-elm (w v) (drop w (flatten v)))

(defun drop (w v &aux (a (car w)) (d (cdr w)))
  (cond ((null w) nil)
        ((listp a) (cons (drop a v) (drop d v)))
        ((member a v) (drop d v))
        ((cons a (drop d v)))))

(defun flatten (w)
  (mapcan #'(lambda (a) (if (listp a) (flatten a) `(,a))) w))

> (set-distinction '(((0) ((1) 2)) (3)) '(((0)) (3)) (7)))
(1 2)

```

Решение 10.66.4

```

(defun set-distinction (w v)
  (set-difference (flatten w) (flatten v)))

(defun flatten (w)
  (mapcan #'(lambda (a) (if (atom a) `(,a) (flatten a))) w))

> (set-distinction '(((0) ((1) 2)) (3)) '(((0)) (3)) (7)))
(1 2)

```

Решение 10.66.5

```

(defun set-distinction (w v)
  (set-difference (flatten w) (flatten v)))

(defun flatten (w)
  (typecase w
    (cons (mapcan #'flatten w))
    (t `(,w))))

> (set-distinction '(((0) ((1) 2)) (3)) '(((0)) (3)) (7)))
(1 2)

```

Решение 10.66.5

```

(defun set-distinction (w v)
  (set-difference (flatten w) (flatten v)))

(defun flatten (w)
  (if (atom w) `(,w)
      (mapcan #'flatten w)))

> (set-distinction '(((0) ((1) 2)) (3)) '(((0)) (3)) (7)))
(1 2)

```

Задача 10.67 *mx-product.lisp*

...neurosurgery is another field that requires an actual investment and lots of dedication to get into, is really rewarding to those who get good at it, but whose jobs are not advertised in regular newspapers. there is a shortage of neurosurgeons, but very little advertising in the media that the patients read. programming is both similar and different. whether you are a user or a programmer these days is often hard to tell (this has good qualities to it, too), but some programming tasks are still reserved to highly skilled people who are not afraid to take huge risks. ignoring for a moment the power of the American Medical Association, we still wouldn't see a huge amount of books on neurosurgery for dummies in 21 days or whatever. it's just plain inappropriate, and it's intentionally out of people's reach. Lisp is somewhat like that. people can get lots of medicines at the drugstore, but they can't be trusted to carve out a malignant tumor in their child's brain. all sorts of users can do lots of customization and cool stuff in their "apps", but they really can't be trusted to run actual flight control systems, configure the telephone network, write software for video-synchronized magnetic-resonance imaging for brain surgery, or write automated stock-trading systems. at some point, the risk of letting unskilled people do the task becomes too high. that's when you can't trust more than 1% of the programmers out there, and a surprisingly large number of them know and use Lisp and tools that are can be trusted. (consider an ATM that gets one of those frequent Windows crashes, or a naval warfare vessel that has to cold-boot because a certain display suddenly goes all blue, or any other story in comp.risks that would have been hilarious if it had been a joke.)

Erik Naggum

Определить функцию, возвращающую произведение двух матриц, заданных в виде списков. Предусмотреть проверку корректности ввода исходных данных.

Решение 10.67.1

```
(defun mx-product (w v)
  (when (= (length (car w)) (length v))
    (mapcar #'(lambda (r)
      (apply #'mapcar #'(lambda (&rest c)
        (reduce #'+ (mapcar #'* r c)))
        v))
      w)))

> (mx-product '((1 2) (3 4)) '((1 2 3) (4 5 6)))
((9 12 15) (19 26 33))
> (mx-product '((1 2) (3 4) (5 6)) '((1 2 3) (4 5 6)))
((9 12 15) (19 26 33) (29 40 51))
> (mx-product '((1 2) (3 4) (5 6)) '((1 2 3) (4 5 6) (7 8 9)))
NIL
```

Задача 10.68 mxx-product.lisp

Emacs is the 100-year editor... Even non-technical people love Emacs. I'm typing in Emacs right now. I'd never voluntarily type anywhere else. It's more than just a productivity boost from having great typing shortcuts and text-editing features found nowhere else on the planet. I type 130 to 140 WPM, error-free, in Emacs, when I'm doing free-form text. I've timed it, with a typing-test Emacs application I wrote. But it's more than that... Emacs has the Quality Without a Name.

Steve Yegge

Определить функцию, возвращающую произведение двух матриц, заданных в виде массивов. Предусмотреть проверку корректности ввода исходных данных.

Решение 10.68.1

```
(defun mxx-product (w v
  &aux
    (a (array-dimension w 0))
    (b (array-dimension w 1))
    (c (array-dimension v 1))
    (z (make-array `(,a ,c) :initial-element 0)))
  (when (= b (array-dimension v 0))
    (loop for i below a do
      (loop for k below c do
        (setf (aref z i k)
          (loop for j below b
            sum (* (aref w i j)
              (aref v j k)))))))
```

```

        finally (return z))))
> (mxx-product #2a((1 2) (3 4)) #2a((1 2 3) (4 5 6)))
#2A((9 12 15) (19 26 33))
> (mxx-product #2a((1 2) (3 4) (5 6)) #2a((1 2 3) (4 5 6) (7 8 9)))
NIL

```

Задача 10.69 *msort.lisp*

So, how should we fix Lisp in order to make it more attractive to performance-oriented people? I personally don't think we should try to alter one single thing of Lisp in order to attract these people. Why should all that counts in a programming language be popularity? I often see the very revealing parallel drawn between Lisp and a fine violin. Should we alter violins in order to attract people who are used to playing the accordion, and who are not willing to learn the violin because it is too hard and too different from the accordion? Of course not! The violin has its place, and is a great instrument when played by someone who really masters it. Lisp is a great programming language when used by someone who knows how to use it. Dumbing it down to attract mediocre performance-oriented programmers would be a grave mistake.

Robert Strandh

Определить функцию, возвращающую отсортированный по возрастанию результат слияния двух числовых списков.

Решение 10.69.1

```

(defun msort (w v) (merge 'list (sort w #'<) (sort v #'<) #'<))
> (msort '(3 1 7) '(9 5 4))
(1 3 4 5 7 9)

```

Задача 10.70 *summ.lisp*

Определить функцию, возвращающую список, элементами которого являются суммы соответствующих элементов двух заданных списков.

Решение 10.70.1

```

(defun summ (w v)
  (cond ((null w) nil)

```

```

      (t (cons (+ (car w) (car v)) (summ (cdr w) (cdr v))))))
> (summ '(3 2 3) '(4 5 6))
(7 7 9)

```

Решение 10.70.2

```

(defun summ (w v)
  (if w (cons (+ (car w) (car v)) (summ (cdr w) (cdr v))) nil))
> (summ '(3 2 3) '(4 5 6))
(7 7 9)

```

Решение 10.70.3

```

(defun summ (w v)
  (when w (cons (+ (car w) (car v)) (summ (cdr w) (cdr v)))))
> (summ '(3 2 3) '(4 5 6))
(7 7 9)

```

Решение 10.70.4

```

(defun summ (w v &optional ac)
  (cond ((null w) (reverse ac))
        (t (summ (cdr w) (cdr v) (cons (+ (car w) (car v)) ac)))))
> (summ '(3 2 3) '(4 5 6))
(7 7 9)

```

Решение 10.70.5

```

(defun summ (w v &optional ac)
  (if w (summ (cdr w) (cdr v) (cons (+ (car w) (car v)) ac))
      (reverse ac)))
> (summ '(3 2 3) '(4 5 6))
(7 7 9)

```

Решение 10.70.6

```

(defun summ (w v)
  (loop for a in w for b in v collect (+ a b)))
> (summ '(3 2 3) '(4 5 6))
(7 7 9)

```

Решение 10.70.7

```

(defun summ (w v) (mapcar #'+ w v))
> (summ '(3 2 3) '(4 5 6))
(7 7 9)

```

Задача 10.71 m.lisp

Я тоже веду войну, только не за пространство, а за время. Я сижу в окопе и отымаю у прошлого клочок времени.

Велимир Хлебников

Даны два числовых списка, упорядоченных по возрастанию. Образовать из элементов этих списков новый список, также упорядоченный.

Решение 10.71.1

```
(defun m (w v)
  (cond ((null w) v)
        ((null v) w)
        ((<= (car w) (car v)) (cons (car w) (m (cdr w) v)))
        (t (cons (car v) (m w (cdr v))))))

> (m '(1 2 7 9) '(3 5 8))
(1 2 3 5 7 8 9)
```

Решение 10.71.2

```
(defun m (w v)
  (cond ((null w) v) ((null v) w)
        ((> (car w) (car v)) (cons (car v) (m w (cdr v))))
        ((cons (car w) (m (cdr w) v)))))

> (m '(1 2 7 9) '(3 5 8))
(1 2 3 5 7 8 9)
```

Решение 10.71.3

```
(defun m (w v)
  (merge 'list w v #'<))

> (m '(1 2 7 9) '(3 5 8))
(1 2 3 5 7 8 9)
```

Задача 10.72 common-maximum-length-words.lisp

Два списка строк. Определить функцию, возвращающую список общих строк максимальной длины.

Решение 10.72.1

```
(defun common-maximum-length-words
  (w
   v
   &aux
    (z (intersection w v :test #'string=)))
```

```

(m (loop for a in z maximize (length a)))
(remove-if #'(lambda (a) (< (length a) m)) z))

> (common-maximum-length-words '("a" "bb" "ccc") '("bb" "ccc" "d"))
("ccc")

```

Задача 10.73 common-max-length-fragments.lisp

Я не желаю быть капитаном дураков.

Роберт Льюис Стивенсон «Остров сокровищ»

Два списка. Определить функцию, возвращающую список общих фрагментов максимальной длины.

Решение 10.73.1

```

(defun common-max-length-fragments
  (w v &aux
    (z (intersection (tails w) (tails v) :test #'equalp))
    (m (loop for a in z maximize (length a))))
  (remove-if #'(lambda (a) (< (length a) m)) z))

(defun tails (w)
  (loop for d on w append (cut d)))

(defun cut (w)
  (cons w (loop for n from 1 to (1- (length w))
    collect (butlast w n))))

> (common-max-length-fragments '(1 2 3 4 5 6 7 8) '(0 2 3 4 0 6 7 8))
((2 3 4) (6 7 8))

```

Решение 10.73.2

```

(defun common-max-length-fragments
  (w v &aux
    (z (intersection (tails w) (tails v) :test #'equalp))
    (m (loop for a in z maximize (length a))))
  (remove-if #'(lambda (a) (< (length a) m)) z))

(defun tails (w) (mapcon #'cut w))

(defun cut (w) (cons w (loop for n from 1 to (1- (length w))
  collect (butlast w n))))

> (common-max-length-fragments '(1 2 3 4 5 6 7 8) '(0 2 3 4 0 6 7 8))
((2 3 4) (6 7 8))

```

Решение 10.73.3

```

(defun common-max-length-fragments

```



```

(w v &aux
  (z (intersection (mapcon #'cut w) (mapcon #'cut v)
    :test #'equalp))
  (m (loop for a in z maximize (length a))))
(remove-if #'(lambda (a) (< (length a) m)) z))

(defun cut (w) (cons w (loop for n from 1 to (1- (length w))
  collect (butlast w n))))

> (common-max-length-fragments '(1 2 3 4 5 6 7 8) '(0 2 3 4 0 6 7 8))
((2 3 4) (6 7 8))

```

Решение 10.73.4

```

(defun common-max-length-fragments
  (w v &aux
    (z (intersection (mapcon #'cut w) (mapcon #'cut v)
      :test #'equalp))
    (m (reduce #'max z :key #'length))))
(remove-if #'(lambda (a) (< (length a) m)) z))

(defun cut (w) (cons w (loop for n from 1 to (1- (length w))
  collect (butlast w n))))

> (common-max-length-fragments '(1 2 3 4 5 6 7 8) '(0 2 3 4 0 6 7 8))
((2 3 4) (6 7 8))

```

Решение 10.73.5

```

(defun common-max-length-fragments
  (w v &aux (z (intersection (mapcon #'cut w) (mapcon #'cut v)
    :test #'equalp)))
  (loop for a in z when (= (length a) (reduce #'max z :key #'length))
    collect a))

(defun cut (w) (cons w (loop for n from 1 below (length w)
  collect (butlast w n))))

> (common-max-length-fragments '(1 2 3 4 5 6 7 8) '(0 2 3 4 0 6 7 8))
((2 3 4) (6 7 8))

```

Структура 11 (функция ATOM ATOM ATOM) > ATOM

We act as though comfort and luxury were the chief requirements of life, when all that we need to make us happy is something to be enthusiastic about.

Einstein

The currency in the developer community is enthusiasm.

Erik Naggum

Задача 11.1 *sum-fibonacci.lisp*

Each new term in the Fibonacci sequence is generated by adding the previous two terms. By starting with 1 and 2, the first 10 terms will be: 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ... Find the sum of all the even-valued terms in the sequence which do not exceed four million.

Каждый следующий элемент ряда Фибоначчи получается при сложении двух предыдущих. Начиная с 1 и 2, первые 10 элементов будут: 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ... Найдите сумму всех четных элементов ряда Фибоначчи, которые не превышают четыре миллиона.

Решение 11.1.1

```
(defun sum-fibonacci (a b z &aux (c (+ a b)))  
  (if (< b z)  
      (if (evenp b)  
          (+ b (sum-fibonacci b c z))  
          (sum-fibonacci b c z))  
      0))  
  
> (sum-fibonacci 1 2 4000000)  
4613732
```

Задача 11.2 *common-elm.lisp*

Написать функцию, которая проверяет – состоят ли два заданных списка из одних и тех же элементов (независимо от порядка их расположения).

Решение 11.2.1

```
(defun common-elm (w v)  
  (null (set-difference w v)))
```

```
> (common-elmisp '(a b c) '(c b a))
T
> (common-elmisp '(a b c) '(c b b))
NIL
> (odd3+ 2 3 5)
10
```

Задача 11.3 in-range.lisp

Написать функцию, которая проверяет принадлежит ли число какому-либо диапазону чисел.

Решение 11.3.1

```
(defun in-range (a b c)
  (<= b a c))

> (in-range 1 1 3)
T
```

Задача 11.4 n-fibonacci.lisp

Последовательность чисел Фибоначчи 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946... строится по следующему правилу: первые два числа - единицы; любое следующее число есть сумма двух предыдущих $f(n) = f(n - 1) + f(n - 2)$. Напишите функцию (f n f1 f2) с накапливающимися параметрами f1 и f2, которая вычисляет n-ое число Фибоначчи.

Решение 11.4.1 * без накапливающих параметров, простая рекурсия

```
(defun n-fibonacci (n)
  (if (< n 2)
      n
      (+ (n-fibonacci (- n 1)) (n-fibonacci (- n 2)))))

> (n-fibonacci 7)
13
> (time (n-fibonacci 35))
Real time: 11.477045 sec.
Run time: 11.434874 sec.
Space: 0 Bytes
9227465
```

Решение 11.4.2 * без накапливающих, хвостовая рекурсия

```
(defun n-fibonacci (n) *без накапливающих, хвостовая рекурсия
  (labels ((fibonacci (n a b)
    (if (= n 0)
        a
        (fibonacci (- n 1) b (+ a b)))))
```

```
(fibonacci n 0 1)))
```

```
> (n-fibonacci 7)
13
> (time (n-fibonacci 35))
Real time: 0.0 sec.
Run time: 0.0 sec.
Space: 0 Bytes
9227465
```

Решение 11.4.3 * с накапливающими, хвостовая рекурсия

```
(defun n-fibonacci (a b n)
  (cond ((= n 2) b)
        ((n-fibonacci b (+ a b) (1- n)))))

> (n-fibonacci 1 1 7)
13
> (time (n-fibonacci 1 1 35))
Real time: 0.0 sec.
Run time: 0.0 sec.
Space: 0 Bytes
9227465
```

Задача 11.5 *replace-lets.lisp*

Дан текст. В каждом слове текста заменить заданную литеру заданным сочетанием литер. Пример: заменяемая литера *b*, заменяющее сочетание литер: *"ku"*, слово: *"abrakadabra"*, результат: *"akurakadakura"*.

Решение 11.5.1

```
(defun replace-lets (s p r &aux (e (search p s)))
  (cond ((null e) s)
        ((concatenate
          'string
          (subseq s 0 e)
          r
          (replace-lets (subseq s (1+ e)) p r)))))

> (replace-lets "abrakadabra" "b" "ku")
"akurakadakura"
```

Решение 11.5.2

```
(defun replace-lets (s p r &aux (e (search p s)))
  (if e
      (concatenate
        'string
        (subseq s 0 e)
        r
        (replace-lets (subseq s (1+ e)) p r))
      s))
```

```

s))

> (replace-lets "abrakadabra" "b" "ku")
"akurakadakura"

```

Задача 11.6 a-delete-b.lisp

Дан текст. В каждом слове удалить литеру, стоящую между двумя заданными.

Решение 11.6.1

```

(defun sw (s)
  (map 'list #'string s))

(defun az (w a b)
  (cond ((null w) nil)
        ((and (string= (car w) a) (string= (caddr w) b))
         (cons (car w) (az (cddr w) a b)))
        ((cons (car w) (az (cdr w) a b)))))

(defun cn (w)
  (cond ((null w) nil)
        ((concatenate 'string (car w) (cn (cdr w))))))

(defun a-delete-b (s a b)
  (cn (az (sw s) a b)))

> (a-delete-b "abc abc" "a" "c")
"ac ac"

```

Решение 11.6.2

```

(defun sw (s)
  (map 'list #'string s))

(defun az (w a b)
  (cond ((null w) nil)
        ((and (equal (car w) a) (equal (caddr w) b))
         (cons (car w) (az (cddr w) a b)))
        ((cons (car w) (az (cdr w) a b)))))

(defun cn (w) (format nil "~{~A~^~}" w))

(defun a-delete-b (s a b)
  (cn (az (sw s) a b)))

> (a-delete-b "abc abc" "a" "c")
"ac ac"

```

Решение 11.6.3

```

(defun sw (s)
  (map 'list #'string s))

(defun az (w a b)
  (cond ((null w) nil)
        ((and (string-equal (car w) a) (string-equal (caddr w) b))
         (cons (car w) (az (cddr w) a b)))
        ((cons (car w) (az (cdr w) a b)))))

(defun cn (w) (format nil "~{~A~^~}" w))

(defun a-delete-b (s a b)
  (cn (az (sw s) a b)))

> (a-delete-b "abc abC" "a" "c")
"ac aC"

```

Решение 11.6.4

```

(defun sw (s)
  (map 'list #'string s))

(defun az (w a b)
  (when w (cons (car w)
                (az (if (and (string-equal (car w) a)
                              (string-equal (caddr w) b))
                        (cddr w)
                        (cdr w))
                  a b))))

(defun cn (w) (format nil "~{~A~^~}" w))

(defun a-delete-b (s a b)
  (cn (az (sw s) a b)))

> (a-delete-b "abc abC" "a" "c")
"ac aC"

```

Решение 11.6.5

```

(defun az (w a b)
  (when w (cons (car w)
                (az (if (and (string-equal (car w) a)
                              (string-equal (caddr w) b))
                        (cddr w)
                        (cdr w))
                  a b))))

(defun a-delete-b (s a b)
  (format nil "~{~A~^~}" (az (map 'list #'string s) a b)))

> (a-delete-b "abc abC" "a" "c")
"ac aC"

```

Решение 11.6.6

```
(defun a-delete-b (s a b)
  (format nil "~{~A~^~}"
    (labels ((az (w a b)
              (when w
                (cons (car w)
                      (az (if (and (string-equal (car w) a)
                                   (string-equal (caddr w) b))
                          (cddr w)
                          (cdr w))
                        a b))))))
    (az (map 'list #'string s) a b))))

> (a-delete-b "abc abC" "a" "c")
"ac aC"
```

Задача 11.7 *sum-n-fibonacci.lisp*

Вычислить сумму n чисел Фибоначчи.

Решение 11.7.1

```
(defun sum-n-fibonacci (a b z &aux (c (+ a b)))
  (if (= z 1) 1 (+ b (sum-n-fibonacci b c (1- z)))))

> (sum-n-fibonacci 1 1 100)
927372692193078999175
```

Задача 11.8 *sum-fibonacci<=n.lisp*

Последовательность чисел Фибоначчи 1,1,2,3,5,8,13... строится по следующему правилу: первые два числа - единицы; любое следующее число есть сумма двух предыдущих $f(n) = f(n - 1) + f(n - 2)$. Напишите функцию (f f1 f2 n), которая вычисляет сумму чисел Фибоначчи которые не превышают n .

Решение 11.8.1

```
(defun sum-fibonacci<=n (&optional (a 0) (b 1) &aux (c (+ a b)))
  (if (< b 100) (+ b (sum-fibonacci<=n b c)) 0))

> (sum-fibonacci<=n)
232
```

Решение 11.8.2

```
(defun sum-fibonacci<=n (&optional (a 0) (b 1))
  (if (< b 100) (+ b (sum-fibonacci<=n b (+ a b))) 0))
```

```
> (sum-fibonacci<=n)
232
```

Решение 11.8.3

```
(defun sum-fibonacci<=n (a b z &aux (c (+ a b)))
  (if (< b z)
      (+ b (sum-fibonacci<=n b c z))
      0))

> (sum-fibonacci<=n 0 1 100)
232
```

Решение 11.8.4

```
(defun sum-fibonacci<=n (a b z)
  (if (< b z)
      (+ b (sum-fibonacci<=n b (+ a b) z))
      0))

> (sum-fibonacci<=n 0 1 100)
232
```

Задача 11.9 balloon-eq.lisp

Никто не может грустить, когда у него есть воздушный шарик!

Алан Милн «Винни-Пух и все-все-все»

Воздушный шар летит в одном направлении со скоростью 20 км/час в течение 1 часа и 45 минут. Затем направление движения меняется на заданный угол, и воздушный шар летит еще 1 час и 45 минут с той же скоростью. Найти расстояние от точки старта до точки приземления.

Решение 11.9.1 * равные «плечи»

```
(defun balloon-eq (velocity time angle)
  (* 2 (sin (* pi (/ angle 360))) velocity time))

> (balloon-eq 20 1.75 60)
35.00000000000000000000003L0
> (balloon-eq 20 1.75 120)
60.621778264910705276L0
> (balloon-eq 20 1.75 300)
35.0L0
> (balloon-eq 20 1.75 0)
0
> (balloon-eq 20 1.75 180)
70.0L0
> (balloon-eq 20 7/4 180)
70.0L0
```


Задача 11.10 arithmetic-n.lisp

Найти n -й член арифметической прогрессии (нумерация начинается с нуля).

Решение 11.10.1

```
(defun arithmetic-n (zero step n) (+ zero (* step n)))

> (arithmetic-n 0 2 100)
200
```

Задача 11.11 check-expt.lisp

Определить функцию, проверяющую является ли третий элемент списка результатом возведения в степень первого элемента с показателем, равным второму элементу. Если является, то вернуть `t`, иначе - `nil`.

Решение 11.11.1

```
(defun check (a b c) (eql (expt a b) c))

> (check 2 3 8)
T
> (check 2 3.123 8.711975)
T
```

Задача 11.12 check-random.lisp

Определить функцию, которая будет проверять, попало ли случайно выбранное из отрезка (5 155) целое число в интервал (25 100).

Решение 11.12.1

```
(defun check-random (n m a b &aux (k (+ n (random (1+ (- m n))))))
  (when (< a k b) k))

> (check-random 5 155 25 100)
37
> (check-random 5 155 25 100)
34
> (check-random 5 155 25 100)
NIL
```

Задача 11.13 triangle.lisp

Определить функцию (`f a b c`), которая равна истине тогда и только тогда, когда из отрезков с длинами a , b и c можно построить треугольник.

Решение 11.13.1

```
(defun triangle (a b c) (< (abs (- a b)) c (+ a b)))

> (triangle 2 2 3)
T
> (triangle 1 2 3)
NIL
```

Задача 11.14 multiples-product.lisp

Среди всех натуральных двузначных чисел найти кратные четырем и вычислить их произведение.

Решение 11.14.1

```
(defun multiples-product (a b n)
  (reduce #'* (loop for e from a to b
                    when (zerop (rem e n))
                    collect e)))

> (four-multiples-product 12 96 4)
5457521857125853405570201949306880000
```

Задача 11.15 sum-multiples.lisp

If we list all the natural numbers below 10 that are multiples of 3 or 5, we get 3, 5, 6 and 9. The sum of these multiples is 23. Find the sum of all the multiples of 3 or 5 below 1000.

Если выписать все натуральные числа меньше 10, кратные 3 или 5, то получим 3, 5, 6 и 9. Сумма этих чисел - 23. Найдите сумму всех чисел меньше 1000, кратных 3 или 5.

Решение 11.15.1

```
(defun sum-multiples (n a b)
  (cond ((zerop n) 0)
        ((or (zerop (rem n a)) (zerop (rem n b)))
         (+ n (sum-multiples (1- n) a b)))
        ((sum-multiples (1- n) a b))))

> (sum-multiples 9 3 5)
23
> (sum-multiples 999 3 5)
233168
```

Решение 11.15.2

```
(defun sum-multiples (n a b)
  (loop for e from 1 to n
        when (or (zerop (rem e a)) (zerop (rem e b)))
```

```
sum e))
```

```
> (sum-multiples 9 3 5)
23
> (sum-multiples 999 3 5)
233168
```

Решение 11.15.3

```
(defun sum-multiples (n a b)
  (loop for e upto n
        when (or (zerop (rem e a))
                  (zerop (rem e b)))
        sum e))
```

```
> (sum-multiples 9 3 5)
23
> (sum-multiples 999 3 5)
233168
```

Задача 11.16 random-number-sower.lisp

Написать программу, которая создает файл, содержащий действительные числа.

Решение 11.16.1

```
(defun random-number-sower (z n path)
  (with-open-file
    (r path :direction :output :if-exists :supersede)
    (format r "~{~s ~}" (rand z n))))
```

```
(defun rand (r n)
  (loop for i from 1 to n
        collect (random r)))
```

```
> (random-number-sower 100 20 "d:/test.txt")
NIL
```

```
d:/test.txt
75 93 74 74 46 51 29 93 10 38 25 8 39 44 80 40 71 42 41 56
```

Решение 11.16.2

```
(defun random-number-sower (z n path)
  (with-open-file
    (r path :direction :output :if-exists :supersede)
    (format r "~{~s ~}" (loop for i from 1 to n
                              collect (random z)))))
```

```
> (random-number-sower 100 20 "d:/test.txt")
NIL
```

```
d:/test.txt
```

```
56 30 25 10 12 18 29 98 75 17 40 0 82 7 71 92 35 15 39 47
```

Задача 11.17 trianglep.lisp

Определить функцию, которая определяет - существует ли треугольник с заданными длинами сторон a , b и c .

Решение 11.17.1

```
(defun trianglep (a b c)
  (and (> (+ a b) c) (> (+ a c) b) (> (+ b c) a)))

> (trianglep 1 2 3)
NIL
> (trianglep 2 3 4)
T
```

Задача 11.18 product-xmultiples.lisp

Lisp will teach you how to do things with very little. The syntax is so minimal it changes your view of programming languages, and certain constructs in particular. Then there macros (no, these are not comparable to the C/C++ style macros) which you just have to experience to understand.

Jakub Arnold

Вычислить произведение натуральных двузначных чисел кратных четырем.

Решение 11.18.1

```
(defun product-xmultiples (n m x)
  (cond ((> n m) 1)
        ((zerop (rem n x)) (* n (product-xmultiples (1+ n) m x)))
        ((product-xmultiples (1+ n) m x))))

> (product-xmultiples 10 99 4)
5457521857125853405570201949306880000
```

Решение 11.18.2

```
(defun product-xmultiples (n m x)
  (if (> n m) 1 (* (if (zerop (rem n x)) n 1)
                  (product-xmultiples (1+ n) m x))))

> (product-xmultiples 10 99 4)
5457521857125853405570201949306880000
```

Решение 11.18.3

```
(defun product-xmultiples (n m x)
  (reduce #'* (loop for a from n to m
                    when (zerop (rem a x))
                    collect a)))
```

```
> (product-xmultiples 10 99 4)
5457521857125853405570201949306880000
```

Задача 11.19 *drop-xmultiples.lisp*

The fact that in effect Lisp isn't "actually" a language. Rather you're programming using s-expressions to represent an AST (Abstract Syntax Tree) directly ... which most other languages have to be translated (lexed & parsed) into before they can be interpreted / compiled.

Irné Barnard

Реализовать функцию, которая удаляет из списка натуральных чисел от n до m все числа, кратные x .

Решение 11.19.1

```
(defun drop-xmultiples (n m x)
  (cond ((> n m) nil)
        ((zerop (rem n x)) (drop-xmultiples (1+ n) m x))
        ((cons n (drop-xmultiples (1+ n) m x)))))
```

```
> (drop-xmultiples 1 30 4)
(1 2 3 5 6 7 9 10 11 13 14 15 17 18 19 21 22 23 25 26 27 29 30)
```

Решение 11.19.1

```
(defun drop-xmultiples (n m x)
  (loop for a from n to m
        unless (zerop (rem a x))
        collect a))
```

```
> (drop-xmultiples 1 30 4)
(1 2 3 5 6 7 9 10 11 13 14 15 17 18 19 21 22 23 25 26 27 29 30)
```

Задача 11.20 *substitute-word.lisp*

Заменить в строке все вхождения слова на новое слово.

Решение 11.20.1

```
(defun substitute-word (z a s)
  (format nil "~{~a~^ ~}"
    (substitute
      (read-from-string z)
      (read-from-string a)
      (read-from-string
        (concatenate 'string "(" s ")")))))

> (substitute-word "zz" "aa" "aa bb aa")
"ZZ BB ZZ"
```

Задача 11.21 *marathon-ms.lisp*

Скорость бега кенийского легкоатлета Денниса Киметто, победителя Берлинского марафона 2014 года с новым мировым рекордом — 2:02.57.

Решение 11.21.1

```
(defun marathon-ms (h m s)
  (float (/ 42195 (+ (* 3600 h) (* 60 m) s))))

> (marathon-ms 2 2 57)
5.719805
```

Задача 11.22 *sum-seven.lisp*

Напишите функцию семи аргументов, значением которой служит сумма всех семи аргументов.

Решение 11.22.1

```
(defun sum-seven (n1 n2 n3 n4 n5 n6 n7)
  (+ n1 n2 n3 n4 n5 n6 n7))

> (sum-seven 1 2 3 4 5 6 700)
721
```

Решение 11.22.2

```
(defun sum-seven (&rest w) (reduce #'+ w))

> (sum-seven 1 2 3 4 5 6 700)
721
```

Задача 11.23 *rectangular-cuboid-volume.lisp*

Даны длина, ширина и высота прямоугольного параллелепипеда. Определить функцию, возвращающую его объем.

Решение 11.23.1

```
(defun rectangular-cuboid-volume (a b c) (* a b c))

> (rectangular-cuboid-volume 10 5 3)
150
```

Решение 11.23.2

```
(defun rectangular-cuboid-volume (a b c)
  (* (footprint a b) c))

(defun footprint (a b)
  (* a b))

> (rectangular-cuboid-volume 10 5 3)
150
```

Задача 11.24 *rectangular-cuboid-lateral-surface-area.lisp*

What makes a Lisp a Lisp? IMO, it has to be homoiconic and it has to support macros. The funny thing is, Douglas Crockford (and many others) insist that JavaScript is “Lisp in C’s clothing.” But JavaScript is missing the two most important characteristics of Lisp. Just having lambdas does NOT make it a Lisp. I mean, Smalltalk has always had lambdas and closures, and nobody calls it a Lisp.

Richard Eng

Даны длина, ширина и высота прямоугольного параллелепипеда. Определить функцию, возвращающую площадь его боковой поверхности.

Решение 11.24.1

```
(defun rectangular-cuboid-lateral-surface-area (a b c)
  (* (+ a b) c 2))

> (rectangular-cuboid-lateral-surface-area 10 5 3)
90
```

Решение 11.24.2

```
(defun rectangular-cuboid-lateral-surface-area (a b c)
  (* (base-perimeter a b) c))

(defun base-perimeter (a b)
  (* (+ a b) 2))

> (rectangular-cuboid-lateral-surface-area 10 5 3)
90
```

Задача 11.25 rectangular-cuboid-surface-area.lisp

Даны длина, ширина и высота прямоугольного параллелепипеда. Определить функцию, возвращающую площадь его поверхности.

Решение 11.25.1

```
(defun rectangular-cuboid-surface-area (a b c)
  (+ (* a b 2) (* a c 2) (* b c 2)))

> (rectangular-cuboid-surface-area 10 5 3)
190
```

Решение 11.25.2

```
(defun rectangular-cuboid-surface-area (a b c)
  (* (+ (* a b) (* a c) (* b c)) 2))

> (rectangular-cuboid-surface-area 10 5 3)
190
```

Задача 11.26 rectangular-cuboid-space-diagonal-length.lisp

I think homoiconicity is absolutely important, but I think the “killer feature” of Lisp is that it’s interactive. I mean way more interactive than, say, Python. While it may seem a little hyperbolic, your Lisp image is like an extension of your mind, in the sense that everything is so accessible for your perusal, without much getting in your way. You can disassemble functions, examine data objects, traverse graphs of objects, reason about the safety and speed of your code, examine types, redefine functions while they’re running, update GUIs on the fly, etc. The language (and associated environment) feel so close to you. Compare this to the distance you feel every printf-compile-run debug cycle you have with C++ or so.

Robert Smith

Даны длина, ширина и высота прямоугольного параллелепипеда. Определить функцию, возвращающую длину его диагонали.

Решение 11.26.1

```
(defun rectangular-cuboid-space-diagonal-length (a b c)
  (pythagoras (pythagoras a b) c))

(defun pythagoras (a b)
```



```
(expt (+ (expt a 2) (expt b 2)) 1/2))

> (rectangular-cuboid-space-diagonal-length 10 5 3)
11.575837
```

Решение 11.26.2

```
(defun rectangular-cuboid-space-diagonal-length (a b c)
  (pythagoras (pythagoras a b) c))

(defun pythagoras (a b)
  (sqrt (+ (* a a) (* b b))))

> (rectangular-cuboid-space-diagonal-length 10 5 3)
11.575837
```

Решение 11.26.3

```
(defun rectangular-cuboid-space-diagonal-length (a b c)
  (expt (+ (expt a 2) (expt b 2) (expt c 2)) 1/2))

> (rectangular-cuboid-space-diagonal-length 10 5 3)
11.575837
```

Решение 11.26.4

```
(defun rectangular-cuboid-space-diagonal-length (a b c)
  (sqrt (+ (* a a) (* b b) (* c c))))

> (rectangular-cuboid-space-diagonal-length 10 5 3)
11.575837
```

Задача 11.27 *insert-word.lisp*

Написать программу, которая вставляет в текст слово после указанного слова.

Решение 11.27.1

```
(defun insert-word (s p r &aux (e (search p s)))
  (cond ((null e) s)
        ((concatenate 'string
                        (subseq s 0 e)
                        p
                        " "
                        r
                        (insert-word (subseq s
                                          (+ e (length p)))
                                    p
                                    r))))))

> (insert-word "To be or not to be" "be" "beer")
```

```
"To be beer or not to be beer"
> (insert-word "To bbbe" "be" "beer")
"To bbbe beer"
```

Задача 11.28 roofing.lisp

The single most important feature of Lisp for a 10-year-old is its simplicity: you always write it as (function arguments ...). Even an 7-year-old can learn it. But honestly, if I wasn't interested in meta-programming so much, I don't think I would be so much focused on Lisp. And it probably takes time to appreciate all that... Let us not drag 10 year olds in our discussions about the superiority of one programming language over others. I'm sure they're smarter than this, and won't be interested anyway. At this age, it is far more exciting to try out many different things, than to listen to some old farts whining about the rationality of choices. But if you're really afraid that your 10yo will choose the wrong technology stack, you can buy them Conrad Barski's «Land of Lisp» – it's got some comics in it! (and also make sure to cut them off from the Internet and remove the browser from their computer. That should work.) You can also try saying something like "when you get older, you'll understand", and occasionally laugh at JavaScript, saying that it's good for whimps from the kindergarden. Or you can forbid Common Lisp completely, saying that under no circumstances are they allowed to touch it – this should whet their curiosity.

Panicz Godek

Заданы следующие параметры здания: высота h , длина l , ширина w , фронтальный и боковой свес крыши d . Рассчитать количество кровельного материала для двускатной крыши, если высота конька $2/3 * h$.

Решение 11.28.1

```
(defun roofing (h l w d)
  (* (+ (sqrt (+ (expt (* 2/3 h) 2)
                  (expt (* 0.5 w) 2)))
      d)
     (+ 1 d d)
     2))
```

```
> (roofing 3 10 5 0.4)
77.79374
```

Задача 11.29 tourist-fatigue.lisp

They do certainly give very strange, and new-fangled, names to diseases.

Plato «The Republic»

Турист в первый день прошел 30 км, на каждый следующий день проходил на 3 км меньше, чем в предыдущий. Какое расстояние он прошел за 11 дней?

Решение 11.29.1

```
(defun tourist-fatigue (m n i)
  (cond ((or (zerop i) (< m n)) 0)
        (t (+ m (tourist-fatigue (- m n) n (1- i))))))

> (tourist-fatigue 30 3 1)
30
> (tourist-fatigue 30 3 9)
162
> (tourist-fatigue 30 3 10)
165
> (tourist-fatigue 30 3 11)
165
```

Решение 11.29.2

```
(defun tourist-fatigue (m n i)
  (if (or (zerop i) (< m n)) 0
      (+ m (tourist-fatigue (- m n) n (1- i)))))

> (tourist-fatigue 30 3 1)
30
> (tourist-fatigue 30 3 9)
162
> (tourist-fatigue 30 3 10)
165
> (tourist-fatigue 30 3 11)
165
```

Решение 11.29.3

```
(defun tourist-fatigue (m n i)
  (loop repeat i when (plusp m) sum m
        do (setf m (- m n))))

> (tourist-fatigue 30 3 1)
30
```

```
> (tourist-fatigue 30 3 9)
162
> (tourist-fatigue 30 3 10)
165
> (tourist-fatigue 30 3 11)
165
```

Решение 11.29.4

```
(defun tourist-fatigue (m n i)
  (loop repeat i when (plusp m) collect m into w
    do (setf m (- m n))
    finally (return (values w (reduce #'+ w))))))
```

```
> (tourist-fatigue 30 3 9)
(30 27 24 21 18 15 12 9 6)
162
> (tourist-fatigue 30 3 11)
(30 27 24 21 18 15 12 9 6 3)
165
```

Решение 11.29.5

```
(defun tourist-fatigue (m n i)
  (loop repeat i collect (if (plusp m) m 0) into w
    do (setf m (- m n))
    finally (return (values w (reduce #'+ w))))))
```

```
> (tourist-fatigue 30 3 11)
(30 27 24 21 18 15 12 9 6 3 0)
165
```

Решение 11.29.6

```
(defun tourist-fatigue (m n i)
  (loop repeat i collect (if (plusp m) m 0) into w
    when (plusp m) sum m into s
    do (setf m (- m n))
    finally (return (values w s))))
```

```
> (tourist-fatigue 30 3 11)
(30 27 24 21 18 15 12 9 6 3 0)
165
```

Решение 11.29.7

```
(defun tourist-fatigue (m n i)
  (loop repeat i
    for a from m downto 0 by n
    collect a into w
    sum a into s
    finally (return (values w s))))
```

```
> (tourist-fatigue 30 3 11)
(30 27 24 21 18 15 12 9 6 3 0)
165
```

Решение 11.29.8

```
(defun tourist-fatigue (m n i)
  (loop repeat i for a from m downto 0 by n sum a))

> (tourist-fatigue 30 3 11)
165
```

Задача 11.30 *replace-all.lisp*

Incompetence, like misery, seeks company.

Erik Naggum

Определить функцию, заменяющую все вхождения подстроки в строке.

Решение 11.30.1 (источник: cl-cookbook.sourceforge.net/strings.html)

```
(defun replace-all (string part replacement &key (test #'char=))
  "Returns a new string in which all the occurrences of the part
  is replaced with replacement."
  (with-output-to-string (out)
    (loop with part-length = (length part)
      for old-pos = 0 then (+ pos part-length)
      for pos = (search part string
                        :start2 old-pos
                        :test test)
      do (write-string string out
                       :start old-pos
                       :end (or pos (length string)))
      when pos do (write-string replacement out)
      while pos)))

> (replace-all "Marx Groucho" "Groucho" "ReplacementForGroucho")
"Marx ReplacementForGroucho"
```

Задача 11.31 *sum-third-even.lisp*

Найти сумму каждых третьих четных натуральных чисел от 1 до 100.

Решение 11.31.1

```
(defun sum-third-even (n m a b &aux (c (* a b)))
  (loop for e from n to m when (zerop (rem e c)) sum e))

> (sum-third-even 1 100 2 3)
```

816

Решение 11.31.2

```
(defun sum-third-even (n a b)
  (loop for e from 0 to n by (* a b) sum e))

> (sum-third-even 100 2 3)
816
```

Задача 11.32 gap-multip-345.lisp

In a way, learning a typical programming language is similar to learning a foreign language as an adult. Suppose you go out tomorrow and decide you're going to learn French. You may take every course on French that you can find, read materials that are only in French, and even move to France. But no matter what you do, your understanding of French will always remain a little imperfect. And no matter how good of a French speaker you eventually become, in your dreams you probably will still be speaking in your native language. Lisp is different. It's not just like learning any foreign language. Once you've learned Lisp, you'll even dream in Lisp. Lisp is such a powerful idea that it will crowd out your previous programming experience and become your new mother tongue! Whenever you encounter a new programming idea in any language, you'll always say to yourself, "That's kind of how I'd do it in Lisp, except" That's the kind of power only Lisp will give you.

Conrad Barski, M.D.

Даны числа a и n . Верно ли, что число a принадлежит промежутку от n до n и кратно 3, 4 и 5.

Решение 11.32.1

```
(defun gap-multip-345 (a n m)
  (and (<= n a m) (zerop (rem a (* 3 4 5)))))

> (gap-multip-345 600 0 1000)
T
> (gap-multip-345 600 0 500)
NIL
> (gap-multip-345 700 0 1000)
NIL
```

Задача 11.33 day-of-week-1753+.lisp

The other day I had to hand-modify around 1000 files to rid them of SVN keywords. The task wasn't fully programmable because I had to see the code and decide whether to replace the tag based on the context (e.g. was it a variable that happened to have the same name as the SVN keyword?). I threw together a few functions for loading all of the files at once, a function for scanning for all of the SVN keywords in a file, and prompting me to ask whether I should replace this instance of the keyword for each result it found. I noticed some file types where it was always safe to just replace them all, and made a non-interactive version of the same function for instantly dealing with those. So a process that would be very time intensive and error prone if done totally manually was a snap, just because I took around 15 minutes to write functions for dealing with the problem first. The only actual typing I had to do once we were in motion was to run the function on each file and answer y or n. As you gain successes with it, you love it more and more, and it sort of pulls you along to try more things that no other editor can be easily made to do. In the beginning you think it's a text editor. In the end you realize it's a Lisp-based text processing framework, and more.

Reese Currie

Определите функцию, вычисляющую день недели по григорианскому календарю, начиная с 1753 года по настоящее время (0 — понедельник, 1 — вторник, ... 6 — воскресенье).

Впервые григорианский календарь был введён папой римским Григорием XIII в католических странах 4 октября 1582 года взамен прежнего юлианского: следующим днём после четверга 4 октября стала пятница 15 октября. В перешедшей по решению короля Георга II на григорианский календарь 2 сентября 1752 года Британии пришлось сдвигать дату вперёд уже не на 10, а на 11 дней, поскольку с момента вступления в силу нового календаря в континентальной Европе миновал уже целый век и накопился ещё один лишний день. После 2-го сразу наступило 14 сентября.

Григорианский календарь — система исчисления времени, основанная на циклическом обращении Земли вокруг Солнца; продолжительность года принята равной 365,2425 суток. Длительность невисокосного года — 365 суток, високосного — 366. В новом календаре стало действовать новое, бо-

лее точное, правило о високосном годе — год високосный, если: номер года кратен 400 (1600, 2000, 2400); остальные годы — номер года кратен 4 и не кратен 100 (... 1892, 1896, 1904, 1908 ...). Григорианский календарь используется в большинстве государств и стран мира. В литературе также применяется название — Новый стиль.

Решение 11.33.1

```
(defun day-of-week-1753+ (d m y)
  (when (< m 3) (decf y 1))
  (mod (truncate (+ y (/ y 4) (/ (- y) 100) (/ y 400))
    (nth (1- m) '(0 3 2 5 0 3 5 1 4 6 2 4))
    (1- d)))
  7))

> (day-of-week-1753+ 7 1 2015)
2
```

Задача 11.34 *day-of-week-1900+.lisp*

Lisp can give you the power that programming
constantly promises you.

Max Thompson

Определите функцию, вычисляющую день недели по григорианскому календарю, начиная с 1900 года по настоящее время (0 — понедельник, 1 — вторник, ... 6 — воскресенье).

Решение 11.34.1

```
(defun day-of-week-1900+ (d m y)
  (nth-value 6 (decode-universal-time
    (encode-universal-time 0 0 0 d m y))))

> (day-of-week-1900+ 7 1 2015)
2
```


Структура 12 (функция АТОМ АТОМ АТОМ) > (СПИСОК)

Historically, languages designed for other people to use have been bad: Cobol, PL/I, Pascal, Ada, C++. The good languages have been those that were designed for their own creators: C, Perl, Smalltalk, Lisp.

Paul Graham

The best programmers are not marginally better than merely good ones. They are an order-of-magnitude better, measured by whatever standard: conceptual creativity, speed, ingenuity of design, or problem-solving ability.

Randall E. Stross

Задача 12.1 *our-list.lisp*

Определите функцию (list w1 w2 w3) с помощью базовых функций. Используйте имя *our-list*, чтобы не переопределять одноименную встроенную функцию Лиспа.

Решение 12.1.1

```
(defun our-list (a b c) (cons a (cons b (cons c nil))))
```

```
> (our-list 'z 'zz 'zzz)
(Z ZZ ZZZ)
```

Решение 12.1.2

```
(defun our-list (&rest w) w)
```

```
> (our-list 'z 'zz 'zzz)
(Z ZZ ZZZ)
```

Задача 12.2 *odd3+.lisp*

Даны три числа. Построить список из кубов этих чисел, если все три числа - нечетные, вернуть сумму чисел - иначе.

Решение 12.2.1

```
(defun odd3+ (a b c)
  (if (and (oddp a) (oddp b) (oddp c))
      (list (expt a 3) (expt b 3) (expt c 3))
      (+ a b c)))

> (odd3+ 1 3 5)
(1 27 125)
```

Задача 12.3 wagon.lisp

Надо построить 180 км железной дороги. Вес одного метра рельса 36 кг. Сколько потребуется вагонов грузоподъемности 60 тонн, что бы перевезти все рельсы к месту стройки одноколейной дороги.

Решение 12.3.1

```
(defun wagon (r s w) (/ (* r 2 s 1000) w 1000))

> (wagon 36 180 60)
216
```

Решение 12.3.2

```
(defun wagon (r s w) (/ (* r 2 s) w))

> (wagon 36 180 60)
216
```

Задача 12.4 square2nd-or-remove3rd.lisp

Написать функцию с тремя аргументами, которая проверяет, является ли второй элемент списка вещественным или рациональным числом. Если является, то поменять его на квадрат этого числа. Если не является, то вернуть исходный список без последнего элемента.

Решение 12.4.1

```
(defun square2nd-or-remove3rd (a b c)
  (if (realp b) (list a (expt b 2) c) (list a b)))

> (square2nd-or-remove3rd 1 2.2 3)
(1 4.84 3)
```

Решение 12.4.2

```
(defun square2nd-or-remove3rd (a b c)
  (if (realp b) `(,a ,(expt b 2) ,c) `(,a ,b)))

> (square2nd-or-remove3rd 1 2.2 3)
(1 4.84 3)
```

Задача 12.5 trapezoid.lisp

Определить функцию, которая печатает числа в виде трапеции.

Решение 12.5.1

```
(defun trap (w m &aux (v (butlast w)))
  (cond ((zerop m) nil)
        (t (print w) (trap v (1- m)))))

(defun trapezoid (k n m)
  (trap (loop for a from k to n collect a) m))

> (trapezoid 1 6 3)
(1 2 3 4 5 6)
(1 2 3 4 5)
(1 2 3 4)
NIL
```

Задача 12.6 div-rem.lisp

Напишите программу поиска пятизначных чисел, которые при делении на 133 дает в остатке 125 и при делении на 134 дает в остатке 125.

Решение 12.6.1

```
(defun div-rem (b c d e)
  (intersection (div b c) (div d e)))

(defun div (d r)
  (loop for a from 10000 to 99999
        when (= (rem a d) r) collect a))

> (div-rem 133 125 134 125)
(17947 35769 53591 71413 89235)
```

Решение 12.6.2

```
(defun div-rem (b c d e)
  (loop for a from 10000 to 99999 when (and (= (rem a b) c)
                                              (= (rem a d) e))
        collect a))

> (div-rem 133 125 134 125)
(17947 35769 53591 71413 89235)
```

Задача 12.7 same-digits.lisp

Среди всех четырехзначных чисел найти те, в записи которых ровно *n* одинаковых цифр.

Решение 12.7.1

```

(defun same-digits (a b n)
  (loop for e from a to b
        when (check-same n (int>digits e))
        collect e))

(defun check-same (n e)
  (loop for a in e thereis (= (count a e) n)))

(defun int>digits (n &optional ac)
  (if (zerop n) ac (int>digits (truncate n 10)
                                (cons (rem n 10) ac))))

> (same-digits 1000 9999 3)
(1000 1011 1101 1110 1112 1113 1114 1115 1116 1117 1118 1119 1121 1131
 1141 1151 1161 1171 1181 1191 1211 1222 1311 1333 1411 1444 1511 1555
 1611 1666 1711 1777 1811 1888 1911 1999 2000 2022 2111 2122 2202 2212
 2220 2221 2223 2224 2225 2226 2227 2228 2229 2232 2242 2252 2262 2272
 2282 2292 2322 2333 2422 2444 2522 2555 2622 2666 2722 2777 2822 2888
 2922 2999 3000 3033 3111 3133 3222 3233 3303 3313 3323 3330 3331 3332
 3334 3335 3336 3337 3338 3339 3343 3353 3363 3373 3383 3393 3433 3444
 3533 3555 3633 3666 3733 3777 3833 3888 3933 3999 4000 4044 4111 4144
 4222 4244 4333 4344 4404 4414 4424 4434 4440 4441 4442 4443 4445 4446
 4447 4448 4449 4454 4464 4474 4484 4494 4544 4555 4644 4666 4744 4777
 4844 4888 4944 4999 5000 5055 5111 5155 5222 5255 5333 5355 5444 5455
 5505 5515 5525 5535 5545 5550 5551 5552 5553 5554 5556 5557 5558 5559
 5565 5575 5585 5595 5655 5666 5755 5777 5855 5888 5955 5999 6000 6066
 6111 6166 6222 6266 6333 6366 6444 6466 6555 6566 6606 6616 6626 6636
 6646 6656 6660 6661 6662 6663 6664 6665 6667 6668 6669 6676 6686 6696
 6766 6777 6866 6888 6966 6999 7000 7077 7111 7177 7222 7277 7333 7377
 7444 7477 7555 7577 7666 7677 7707 7717 7727 7737 7747 7757 7767 7770
 7771 7772 7773 7774 7775 7776 7778 7779 7787 7797 7877 7888 7977 7999
 8000 8088 8111 8188 8222 8288 8333 8388 8444 8488 8555 8588 8666 8688
 8777 8788 8808 8818 8828 8838 8848 8858 8868 8878 8880 8881 8882 8883
 8884 8885 8886 8887 8889 8898 8988 8999 9000 9099 9111 9199 9222 9299
 9333 9399 9444 9499 9555 9599 9666 9699 9777 9799 9888 9899 9909 9919
 9929 9939 9949 9959 9969 9979 9989 9990 9991 9992 9993 9994 9995 9996
 9997 9998)

```

Задача 12.8 square-digits-times.lisp

В диапазоне чисел найти те, сумма квадратов цифр которых делится на n .

Решение 12.8.1

```

(defun int-digits (n &optional ac)
  (if (zerop n)
      ac
      (int-digits (truncate n 10) (cons (rem n 10) ac))))

```

```
(defun square-digits-times (a b n)
  (loop for m from a to b
        when (zerop (rem (loop for z in (int-digits m)
                               sum (* z z))
                          n))
        collect m))

> (square-digits-times 10 99 13)
(15 18 23 32 46 47 51 64 69 74 79 81 96 97)
> (square-digits-times 100 200 13)
(105 108 134 139 143 150 180 193)
```

Решение 12.8.2

```
(defun int-digits (n &optional ac)
  (if (zerop n)
      ac
      (int-digits (truncate n 10) (cons (rem n 10) ac))))

(defun square-digits-times (a b n)
  (cond ((= a b) nil)
        ((zerop (rem (loop for z in (int-digits a)
                               sum (* z z))
                      n))
         (cons a (square-digits-times (1+ a) b n)))
        ((square-digits-times (1+ a) b n))))

> (square-digits-times 10 99 13)
(15 18 23 32 46 47 51 64 69 74 79 81 96 97)
> (square-digits-times 100 200 13)
(105 108 134 139 143 150 180 193)
```

Задача 12.9 *sum-digits=n.lisp*

What sets Lisp apart is how easily it can be extended and adapted to different purposes, without changing its basic character. The simple-but-powerful syntax lets you focus on semantics rather than syntax, even if the syntax might not be just what you want.

Bob Kerns

В диапазоне чисел найти те, сумма цифр которых равна n.

Решение 12.9.1

```
(defun int-digits (n &optional ac)
  (if (zerop n)
      ac
      (int-digits (truncate n 10) (cons (rem n 10) ac))))
```

```
(defun sum-digits=n (a b n)
  (loop for m from a to b
        when (= (reduce #' + (int-digits m))
                 n)
        collect m))

> (sum-digits=n 100 999 13)
(139 148 157 166 175 184 193 229 238 247 256 265 274 283 292 319 328
 337 346 355 364 373 382 391 409 418 427 436 445 454 463 472 481 490
 508 517 526 535 544 553 562 571 580 607 616 625 634 643 652 661 670
 706 715 724 733 742 751 760 805 814 823 832 841 850 904 913 922 931
 940)
```

Задача 12.10 *rectangle.lisp*

Определить функцию, рисующую прямоугольник значками.

Решение 12.10.1

```
(defun rectangle (n m z)
  (format t "~v@{~v@{~A~:*~}~::~*~%~}" n m z))

> (rectangle 4 10 '*)
*****
*****
*****
*****
(10 *)
```

Задача 12.11 *random-matrix.lisp*

I think the most special reason is: code and data are both lists. So, you can use macros to operate on "code itself". All the other amazing things (CLOS, condition/restart, DSL, etc) are based on this point.

Albert Lee

Определить функцию, возвращающую список-матрицу размером n на m со значениями от 0 до z .

Решение 12.11.1

```
(defun random-matrixx (n m z)
  (loop repeat n collect
        (loop repeat m collect (random z))))

> (random-matrixx 5 7 100)
((18 15 89 97 94 99 53) (42 2 80 88 30 20 97) (79 80 23 72 43 55 98)
 (75 27 95 14 91 65 51) (70 66 37 8 80 20 99))
```

```
> (random-matrixx 7 3 10)
((7 9 0) (3 8 7) (6 3 7) (7 4 2) (2 6 3) (9 7 0) (6 3 1))
```

Задача 12.12 graftage.lisp

Определить функцию от трех аргументов-атомов *a b c*, которая возвращает список вида *(a (b a c) c)*.

Решение 12.12.1

```
(defun graftage (a b c)
  (cons a (cons (cons b (cons a (cons c nil))) (cons c nil))))

> (graftage 10 100 1000)
(10 (100 10 1000) 1000)
```

Решение 12.12.2

```
(defun graftage (a b c)
  (list a (list b a c) c))

> (graftage 10 100 1000)
(10 (100 10 1000) 1000)
```

Решение 12.12.3

```
(defun graftage (a b c)
  `(,a ,(list b a c) ,c))

> (graftage 10 100 1000)
(10 (100 10 1000) 1000)
```

Решение 12.12.4

```
(defun graftage (a b c)
  (list a `,(b ,a ,c) c))

> (graftage 10 100 1000)
(10 (100 10 1000) 1000)
```

Задача 12.13 process.lisp

Определить функцию от трёх аргументов. Если первый аргумент подсписок, то вернуть список аргументов при условии, что длина подсписка меньше либо равна 2, иначе вернуть подсписок без последнего элемента. Иначе вернуть список из второго и третьего аргументов.

Решение 12.13.1

```
(defun process (a b c)
  (if (listp a)
```

```

      (if (<= (length a) 2)
          (list a b c)
          (butlast a))
      (list b c)))

> (process 1 2 3)
(2 3)
> (process '(1 1) 2 3)
((1 1) 2 3)
> (process '(1) 2 3)
((1) 2 3)
> (process '(a b c) 2 3)
(A B)

```

Решение 12.13.2

```

(defun process (a b c)
  (if (listp a)
      (if (cddr a)
          (butlast a)
          (list a b c))
      (list b c)))

> (process 1 2 3)
(2 3)
> (process '(1 1) 2 3)
((1 1) 2 3)
> (process '(1) 2 3)
((1) 2 3)
> (process '(a b c) 2 3)
(A B)

```

Задача 12.14 *list&process.lisp*

Определить функцию от трёх аргументов. Если первый аргумент является строкой или символом, то вернуть список из второго и третьего аргументов, иначе вернуть список из второго, первого и третьего аргументов.

Решение 12.14.1

```

(defun list&process (a b c)
  (cond ((or (stringp a) (symbolp a)) (cons b (cons c nil)))
        (t (cons b (cons a (cons c nil))))))

> (list&process "a" 2 3)
(2 3)
> (list&process 1 2 3)
(2 1 3)

```

Решение 12.14.2

```

(defun list&process (a b c)

```



```

(if (or (stringp a) (symbolp a))
    (cons b (cons c nil))
    (cons b (cons a (cons c nil)))))

> (list&process "a" 2 3)
(2 3)
> (list&process 1 2 3)
(2 1 3)

```

Решение 12.14.3

```

(defun list&process (a b c)
  (cond ((or (stringp a) (symbolp a)) (list b c))
        (t (list b a c))))

> (list&process "a" 2 3)
(2 3)
> (list&process 1 2 3)
(2 1 3)

```

Решение 12.14.4

```

(defun list&process (&rest w)
  (cond ((or (stringp (car w)) (symbolp (car w))) (cdr w))
        (t (rotatef (car w) (cadr w) w))))

> (list&process "a" 2 3)
(2 3)
> (list&process 1 2 3)
(2 1 3)

```

Задача 12.15 min-max.lisp

Определить функцию четырёх аргументов, формирующую список, содержащий максимальное и минимальное числа. Функции min и max не использовать.

Решение 12.15.1

```

(defun min-max (a b c d)
  (cond ((<= a b c d) (list a d))
        ((<= a b d c) (list a c))
        ((<= a c b d) (list a d))
        ((<= a c d b) (list a b))
        ((<= a d b c) (list a c))
        ((<= a d c b) (list a b))
        ((<= b a c d) (list b d))
        ((<= b a d c) (list b c))
        ((<= b c a d) (list b d))
        ((<= b c d a) (list b a))
        ((<= b d a c) (list b c))
        ((<= b d c a) (list b a))
        ((<= c a b d) (list c d))

```

```

((<= c a d b) (list c b))
((<= c b a d) (list c d))
((<= c b d a) (list c a))
((<= c d a b) (list c b))
((<= c d b a) (list c a))
((<= d a b c) (list d c))
((<= d a c b) (list d b))
((<= d b a c) (list d c))
((<= d b c a) (list d a))
((<= d c a b) (list d b))
(t (list d a)))

```

```

> (min-max 4 1 2 3)
(1 4)

```

Задача 12.16 *insulation.lisp*

It's deceptively simple; it unifies code and data in a very elegant and consistent way; and it enjoys all the benefits of functional programming...

Gideon Shavit

Заданы следующие параметры каркасного здания: высота h , длина l , ширина w , толщина утепления стен плитным утеплителем – 0,2 м, пола и потолка рулонным утеплителем – 0,3 м. Рассчитать количество утеплителя обоих видов.

Решение 12.16.1

```

(defun insulation (h l w)
  (list (+ (* h l 0.2 2) (* h w 0.2 2)) (* l w 0.3 2)))

> (insulation 3 10 5)
(18.0 30.0)

```

Задача 12.17 *range.lisp*

Определить функцию, возвращающую список чисел в интервале от n до m с шагом d .

Решение 12.17.1

```

(defun step-range (n m d)
  (cond ((> n m) nil)
        (t (cons n (step-range (+ n d) m d)))))

> (step-range 2 8 2)
(2 4 6 8)
> (step-range 3 8 2)

```

```
(3 5 7)
> (step-range 3 9 2)
(3 5 7 9)
```

Решение 12.17.2

```
(defun step-range (n m d)
  (when (<= n m) (cons n (step-range (+ n d) m d)))))

> (step-range 2 8 2)
(2 4 6 8)
> (step-range 3 8 2)
(3 5 7)
> (step-range 3 9 2)
(3 5 7 9)
```

Решение 12.17.3

```
(defun step-range (n m d)
  (loop for a from n to m by d collect a))

> (step-range 2 8 2)
(2 4 6 8)
> (step-range 3 8 2)
(3 5 7)
> (step-range 3 9 2)
(3 5 7 9)
```

Структура 13 (функция (СПИСОК АТОМ АТОМ) > АТОМ

When all actions are used for feedback, the consequence of making mistakes will be a corrective and appropriate response, because everything everybody does matters... The more selective you are in the feedback you accept, the more insane your reasoning will become as you will necessarily reject corrective feedback that would have led to better reasoning.

Erik Naggum

APL is like a diamond. It has a beautiful crystal structure; all of its parts are related in a uniform and elegant way. But if you try to extend this structure in any way - even by adding another diamond - you get an ugly kludge. LISP, on the other hand, is like a ball of mud. You can add any amount of mud to it and it still looks like a ball of mud.

Joel Moses

Вы печатаете на клавиатуре заклинание, и вот экран дисплея оживает, показывая объекты, которых не было и могло не быть никогда. Но если хоть один символ, один пробел в магической формуле не находится строго на своем месте, волшебство не работает.

~

Всегда пишите код так, будто сопровождать его будет склонный к насилию психопат, который знает, где вы живете.

Мартин Голдинг

Задача 13.1 *sum-sequence.lisp*

Определить функцию, которая суммирует элементы одноуровневого числового списка *w*, расположенные на позициях от *n* до *m* включительно ($n < m$).

Решение 13.1.1

```
(defun sum-sequence (w n m)
  (cond ((= m 0) 0)
        ((= n 1) (+ (car w) (sum-sequence (cdr w) n (1- m)))))
```

```
((sum-sequence (cdr w) (1- n) (1- m))))
> (sum-sequence '(1 2 3 4 5 6 7) 3 5)
12
```

Решение 13.1.2

```
(defun sum-sequence (w n m)
  (reduce #'(lambda (acc) (+ (subseq w (1- n) m)) acc))
  0)
> (sum-sequence '(1 2 3 4 5 6 7) 3 5)
12
```

Задача 13.2 zeros-only.lisp

Определить рекурсивную функцию (`f w a b`), которая проверяет, действительно ли между элементами `a` и `b` в одноуровневом списке `w` нет иных элементов, кроме нулей (элемент `a` встречается раньше элемента `b`).

Решение 13.2.1

```
(defun zeros (w b)
  (cond ((null w) t)
        ((equalp (car w) b) t)
        ((equalp (car w) 0) (zeros (cdr w) b))
        (t nil)))

(defun zeros-only (w a b)
  (cond ((null w) t)
        ((equalp (car w) a) (zeros (cdr w) b))
        ((zeros-only (cdr w) a b))))

> (zeros-only '(1 2 3 0 0 7) 3 7)
T
> (zeros-only '(1 2 3 100 7) 3 7)
NIL
```

Задача 13.3 az>a.lisp

Определить функцию, которая возвращает копию строки текста, в которой все заданные буквы заменены на заданную букву.

Решение 13.3.1

```
(defun az>a (s w z)
  (coerce
   (loop for c across s
         if (find c w) collect z
         else collect c)
   'string))
```

```
> (az>a "congratulations" '(#\a #\e #\i #\o #\u) #\a)
"cangratalataans"
```

Задача 13.4 accord.lisp

It might not be completely unique to Lisp, but a very important feature is the interactivity it provides. With other languages, you tend to write a lot (maybe dozens or hundreds of pages) of code, then compile it, then test it. With lisp, you tend to write small components, evaluate them by passing in ad hoc data, then bundle them together into larger and larger systems. At the same time, you might leave a lot of corner-cases and error handling simply undefined. If you never reach them, you'll never notice. And if/when you do reach them, you'll likely be in a debugging state where you can look at the current context, decide what the appropriate response code should be, add it, and continue on without losing any context. Once the overall structure seems to work for the main examples you have in mind, you can fill in the details at your leisure. That sounds a bit vague, but the concrete thing is the rate of feed-back. Once you get going, there is kind of rapid back-and-forth dialog between you and the code that (a) is very addictive, and (b) is very productive. I know that modern IDE's capture a lot of that feature, but with lisp it is part of the system's DNA, not something grafted on later.

James McDonald

Определить предикат, проверяющий, что атом, стоящий на i -й позиции в списке, является заданным.

Решение 13.4.1

```
(defun accord (a n w)
  (cond ((null w) nil)
        ((zerop n) (eq (car w) a))
        (t (accord a (1- n) (cdr w)))))
```

```
> (accord 'z 2 '(a b z))
T
> (accord 'z 2 '(a b c))
```

Решение 13.4.2

```
(defun accord (a n w)
  (if (zerop n) (eq (car w) a) (accord a (1- n) (cdr w))))

> (accord 'z 2 '(a b z))
T
> (accord 'z 2 '(a b c))
NIL
```

Задача 13.5 counter.lisp

Определить функцию, возвращающую число вхождений двух атомов в простой список.

Решение 13.5.1

```
(defun counter (a b w)
  (+ (count a w) (count b w)))

> (counter 3 5 '(1 2 3 4 5 3 5 5))
5
```

Решение 13.5.2

```
(defun counter (a b w)
  (loop for e in w
        when (or (equalp e a) (equalp e b)) count a))

> (counter 3 5 '(1 2 3 4 5 3 5 5))
5
```

Задача 13.6 conditional-reverse.lisp

Определить функцию проверки одноуровневого списка чисел w на элементы со значениями от a до b, если таковые имеются, то вернуть реверсированный список w, если нет, то исходный.

Решение 13.6.1

```
(defun conditional-reverse (w a b)
  (cond ((> a b) w)
        ((member a w) (reverse w))
        ((conditional-reverse w (1+ a) b))))

> (conditional-reverse '(1 2 3 4 5) 0 1)
(5 4 3 2 1)
> (conditional-reverse '(1 2 3 4 5) 6 7)
(1 2 3 4 5)
> (conditional-reverse '() 6 7)
NIL
```

Задача 13.7 *check.lisp*

Conditionals, functions as first class objects, recursion, call by reference universally, garbage collection, no distinction between expressions and statements, symbols as a data type, code being represented by trees of symbols, having the whole language available at any time/no real distinction between runtime and compile time. These features had been in the Lisps for years before the other strains of languages even had them planned. Nowadays, many modern high-level languages have all but the last three features, which are responsible for Lisps being able to support arbitrary syntax.

Gary Hollis

Определить функцию, которая выдает истину только если ровно для n элементов списка w выполняется предикат p .

Решение 13.7.1

```
(defun check (p w n) (= (count-if p w) n))

> (check #'evenp '(1 2 3 4 5 6 7 8 10) 5)
T
```

Задача 13.8 *inside.lisp*

Определить предикат, проверяющий что элементы списка попали в заданный интервал.

Решение 13.8.1

```
(defun inside (w n m) (loop for a in w always (<= n a m)))

> (inside '(3 5 7) 3 7)
T
```

Решение 13.8.2 (автор – transformator.t, www.cyberforum.ru)

```
(defun inside (w n m) (every #'(lambda (a) (<= n a m)) w))

> (inside '(3 5 7) 3 7)
T
```


Структура 14 (функция (СПИСОК АТОМ АТОМ) > (СПИСОК))

The most uncomfortable thing you can do is set a 1 year goal and achieve it. Make it harder than something you've ever done before... Think about the things that you are comfortable doing every day... then double that... keep doubling this until you get to a point where you become really scared of doing it. For e.g. if you run 3km each day - keep doubling this amount until you get to a point where it really scares you. 6 km? 12 kms? 24km? 50km? 100km... does that scare you. Now set this really scare part as the goal you're going to achieve in 1 year. Then go out and achieve this goal. There is something that's going to happen when you do this... You'll realize this is going to be harder than it sounds... it's easier to just think about a goal but then when you have to sit down and plan for it everyday it becomes harder and harder. But the hardest part is to actually go out there and take the action steps that will help you achieve this goal. Taking the daily action steps is going to be harder still. You will have to use all your mental and physical strength to actually get yourself to achieve this goal. But here's the thing - something very cool happens in your mind, your physiology, your internal makeup when you actually do this. You become invincible.

Rizwan Aseem

Один из верных путей в истинное будущее (ведь есть и ложное будущее) - это идти в том направлении, в котором растёт твой страх.

~

Задача 14.1 *sbst.lisp*

Определить функцию, которая заменяет все вхождения данного элемента в списке на новый элемент.

Решение 14.1.1

```
(defun sbst (n m w)
```

```

(cond ((null w) nil)
      ((equal (car w) m) (cons n (sbst n m (cdr w))))
      ((cons (car w) (sbst n m (cdr w)))))

> (sbst 2 3 '(1 2 3))
(1 2 2)

```

Комментарии (построчно):

1. Определяем функцию **sbst** с тремя параметрами: новый атом **n**, старый атом **m** и список **w**.
2. Если (**cond**) пустой (**null**) список **w** функция **sbst** возвращает **nil**.
3. Если первый элемент (**car w**) равен **m**, то вставляет (**cons**) элемент **n** на первое место в результат вызова **sbst** с параметрами: **n**, **m** и списком без первого элемента (**cdr w**).
4. При невыполнении 1 и 2 условий вставляет (**cons**) первый элемент (**car w**) в результат вызова **sbst** с параметрами: **n**, **m** и списком без первого элемента (**cdr w**).

Решение 14.1.2

```

(defun sbst (w n m &aux (a (car w)))
  (when w (cons (if (equal a n) m a) (sbst (cdr w) n m))))

> (sbst '(1 a 3 a 5 a 7) 'a 'b)
(1 B 3 B 5 B 7)

```

Решение 14.1.3

```

(defun sbst (w n m) (loop for a in w collect (if (equal a n) m a)))

> (sbst '(1 a 3 a 5 a 7) 'a 'b)
(1 B 3 B 5 B 7)

```

Решение 14.1.4

```

(defun sbst (w n m) (substitute m n w))

> (sbst '(1 a 3 a 5 a 7) 'a 'b)
(1 B 3 B 5 B 7)

```

Решение 14.1.5

```

(defun sbst (w n m) (nsubstitute m n w))

> (sbst '(1 a 3 a 5 a 7) 'a 'b)
(1 B 3 B 5 B 7)

```

Задача 14.2 *drop-range.lisp*

Все списки содержат подсписки. Программировать с заходом в подсписки – параллельная рекурсия. Элементами списка могут быть атомы-символы, атомы-числа, подсписки из атомов-символов, атомов-чисел. Нужно: Уда-

лить из списка все атомы-числа из интервала $[a, b]$.

Решение 14.2.1

```
(defun drop-range (n m w)
  (when w ((lambda (a d)
              (cond ((listp a)
                     (cons (drop-range n m a) (drop-range n m d)))
                    ((and (numberp a) (<= n a m)) (drop-range n m d))
                    ((cons a (drop-range n m d))))))
    (car w) (cdr w))))

> (drop-range 3 5 '((1 2 a 4 8) ((3) b (u) 2)))
((1 2 A 8) (NIL B (U) 2))
```

Задача 14.3 list-goods.lisp

Прайс компьютерного магазина описывается списком (наименование фирма параметры цена): ((Терминал Acer 15 4500) (Терминал Sumsung 17 7000) (Матплата Asus 238 2500) (Процессор Intel 2.5 3000)). Определить функцию возвращающую список товаров с заданным наименованием и ценой.

Решение 14.3.1

```
(defun list-googs (m p w)
  (delete-if-not
   #'(lambda (a) (and (eq (car a) m) (= (caddr a) p))) w))

> (list-goods 'terminal 7000 '((terminal acer 15 4500) (terminal
samsung 17 7000) (motherboard intel 238 2500) (processor intel 2.5
3000)))
((TERMINAL SAMSUNG 17 7000))
```

Решение 14.3.2

```
(defun list-goods (n p w)
  (loop for (name firm parameters price) in w
        when (and (eq name n) (= price p))
        collect (list name firm parameters price)))

> (list-goods 'terminal 7000 '((terminal acer 15 4500) (terminal
samsung 17 7000) (motherboard intel 238 2500) (processor intel 2.5
3000)))
((TERMINAL SAMSUNG 17 7000))
```

Задача 14.4 select-range.lisp

Написать функцию, которая принимает произвольный одноуровневый список и возвращает только числа заданного диапазона.

Решение 14.4.1

```
(defun select-range (w x z &aux (a (car w)) (d (cdr w)))
  (cond ((null w) nil)
        ((and (numberp a) (<= x a z)) (cons a (select-range d x z)))
        ((select-range d x z))))
```

```
> (select-range '(a 1 2 3 4) 1 3)
(1 2 3)
```

Решение 14.4.2

```
(defun select-range (w x z)
  (delete-if-not #'(lambda (a) (and (numberp a) (<= x a z))) w))
```

```
> (select-range '(a 1 2 3 4) 1 3)
(1 2 3)
```

Задача 14.5 frame-elms.lisp

Определить функцию, которая добавляет заданный элемент слева и справа от другого заданного элемента.

Решение 14.5.1

```
(defun frame-elms (z u w &aux (a (car w)) (d (cdr w)))
  (cond ((null w) nil)
        ((atom a) (if (eq u a)
                       (cons z (cons a (cons z (frame-elms z u d))))
                       (cons a (frame-elms z u d))))
        ((cons (frame-elms z u a) (frame-elms z u d)))))
```

```
> (frame-elms 5 8 '((1 8 2) (3 (8 4))))
((1 5 8 5 2) (3 (5 8 5 4)))
```

Задача 14.6 reduce-combined.lisp

В числовом списке сложить или умножить каждые N соседних элемента, пример работы функции: (f #' + 2 '(2 3 4 5 6 7)) > (5 9 13)

Решение 14.6.1

```
(defun combine-elms (n w &optional (m n) ac acc)
  (cond ((null w) (reverse (cons (reverse ac) acc)))
        ((zerop m) (combine-elms n w n nil (cons (reverse ac) acc)))
        ((combine-elms n (cdr w) (1- m) (cons (car w) ac) acc))))
```

```
(defun reduce-combined (function n w)
  (mapcar #'(lambda (v) (reduce function v)) (combine-elms n w)))
```

```
> (reduce-combined #' + 2 '(2 3 4 5 6 7))
(5 9 13)
```

```
> (reduce-combined #'* 3 '(2 3 4 5 6 7))
(24 210)
```

Задача 14.7 insert-lets.lisp

Дан текст. В каждом слове вставить после заданного 3-буквенного сочетания заданное 2-буквенное.

Решение 14.7.1

```
(defun insert-lets (w a b)
  (cond ((null w) nil)
        ((eql (car w) a)
         (cons (intern (concatenate 'string
                                     (string a)
                                     (write-to-string b)))
                 (insert-lets (cdr w) a b)))
        ((cons (car w) (insert-lets (cdr w) a b)))))
```

```
> (insert-lets '(ab abc ac abd) 'abc 12)
(AB ABC12 AC ABD)
```

Решение 14.7.2

```
(defun insert-lets (w a b)
  (mapcar
   #'(lambda (x)
       (if (eql x a)
           (intern (concatenate 'string
                                 (string a)
                                 (write-to-string b)))
           x))
   w))
```

```
> (insert-lets '(ab abc ac abd) 'abc 12)
(AB ABC12 AC ABD)
```

Решение 14.7.3 (MuLisp)

```
(defun insert-lets (w a b)
  (mapcar #'(lambda (x) (if (eql x a) (pack (list a b)) x)) w))

> (insert-lets '(ab abc ac abd) 'abc 12)
(AB ABC12 AC ABD)
```

Задача 14.8 replace-elts.lisp

Заменить в исходном списке первое вхождение заданного значения другим.

Решение 14.8.1

```
(defun replace-elts (n m w) (nsubstitute n m w :count 1))

> (replace-elts 'z 'a '(a a))
(Z A)
```

Задача 14.9 *sort-cadr.lisp*

Отсортировать список вида ((A 3) (B 2) (C 5)) так, чтобы получилось ((C 5) (A 3) (B 2))

Решение 14.9.1

```
> (sort-cadr '((A 3) (B 2) (C 5)) #'> :key #'second)
((C 5) (A 3) (B 2))
```

Решение 14.9.2

```
> (sort-cadr '((A 3) (B 2) (C 5)) #'> :key #'cadr)
((C 5) (A 3) (B 2))
```

Задача 14.10 *insert-on.lisp*

Написать программу, которая добавляет элемент в список и ставит его за заданным элементом.

Решение 14.10.1

```
(defun insert-on (n m w)
  (cond ((null w) nil)
        ((eq (car w) n) (cons n (cons m (insert-on n m (cdr w)))))
        ((cons (car w) (insert-on n m (cdr w)))))

> (insert-on 3 100 '(1 2 3 4 5))
(1 2 3 100 4 5)
```

Задача 14.11 *a-*2-change.lisp*

Дан список произвольной длины, который состоит из числовых и символьных атомов. Определить функцию, которая числовые атомы увеличивает в 2 раза, а символьные меняет на атомы A

Решение 14.11.1

```
(defun a-*2-change (w a n)
  (cond ((null w) nil)
        ((numberp (car w)) (cons (* (car w) n)
                                   (a-*2-change (cdr w) a n)))
        (t (cons a (a-*2-change (cdr w) a n)))))

> (a-*2-change '(1 b 3 c 5 f) 'a 2)
(2 A 6 A 10 A)
```

Задача 14.12 *dive-n+m.lisp*

Дан многоуровневый список. Подсчитать количество атомов-чисел на первом и третьем уровнях вложенности.

Решение 14.12.1

```
(defun dive-count (n w &aux (a (car w)) (d (cdr w)))
  (cond ((null w) 0)
        ((= n 0) (count-if #'numberp w))
        ((atom a) (dive-count n d))
        ((+ (dive-count (1- n) a) (dive-count n d)))))

(defun dive-n+m (n m w)
  (+ (dive-count n w) (dive-count m w)))

> (dive-n+m 1 3 '(a(b(c (1))d e)f((2(g))3)))
2
```

Задача 14.13 *del-lev-n.lisp*

Заданы глубина подсписка и позиция. Удалить из всех имеющихся подсписков заданной глубины элементы, находящиеся на указанной позиции.

Решение 14.13.1

```
(defun del-lev-n (w n m)
  (when w ((lambda (a d)
              (cond ((atom a) (cons a (del-lev-n d n m)))
                    ((= n 0) (cons (del-n a m) (del-lev-n d n m)))
                    ((cons (del-lev-n a (1- n) m) (del-lev-n d n m))))))
    (car w) (cdr w))))

(defun del-n (w n)
  (cond ((null w) nil)
        ((= n 1) (cdr w))
        (t (cons (car w) (del-n (cdr w) (1- n))))))

> (del-lev-n '(a b (1 2 3 (1 2 3 (1 2 3))) c d) 0 2)
(A B (1 3 (1 2 3 (1 2 3))) C D)
> (del-lev-n '(a b (1 2 3 (1 2 3 (1 2 3))) c d) 1 2)
(A B (1 2 3 (1 3 (1 2 3))) C D)
> (del-lev-n '(a b (1 2 3 (1 2 3 (1 2 3))) c d) 2 2)
(A B (1 2 3 (1 2 3 (1 3))) C D)
> (del-lev-n '(a b (1 2 3 (1 2 3 (1 2 3))) c d) 3 2)
(A B (1 2 3 (1 2 3 (1 2 3))) C D)
```

Решение 14.13.2

```
(defun del-lev-n (w n m &aux (a (car w)) (d (cdr w)))
  (cond ((null w) nil)
        ((atom a) (cons a (del-lev-n d n m)))))
```

```

      ((= n 0) (cons (del-n a m) (del-lev-n d n m)))
      ((cons (del-lev-n a (1- n) m) (del-lev-n d n m)))))

(defun del-n (w n)
  (when w (if (= n 1)
              (cdr w)
              (cons (car w) (del-n (cdr w) (1- n))))))

> (del-lev-n '(a b (1 2 3 (1 2 3 (1 2 3))) c d) 0 2)
(A B (1 3 (1 2 3 (1 2 3))) C D)
> (del-lev-n '(a b (1 2 3 (1 2 3 (1 2 3))) c d) 1 2)
(A B (1 2 3 (1 3 (1 2 3))) C D)
> (del-lev-n '(a b (1 2 3 (1 2 3 (1 2 3))) c d) 2 2)
(A B (1 2 3 (1 2 3 (1 3))) C D)
> (del-lev-n '(a b (1 2 3 (1 2 3 (1 2 3))) c d) 3 2)
(A B (1 2 3 (1 2 3 (1 2 3))) C D)

```

Решение 14.13.3

```

(defun del-lev-n (w n m &aux (a (car w)) (d (cdr w)))
  (cond ((null w) nil)
        (t (cons (cond ((atom a) a)
                        ((= n 0) (del-n a m))
                        ((del-lev-n a (1- n) m)))
                  (del-lev-n d n m)))))

(defun del-n (w n)
  (cond ((null w) nil)
        ((= n 1) (cdr w))
        (t (cons (car w) (del-n (cdr w) (1- n))))))

> (del-lev-n '(a b (1 2 3 (1 2 3 (1 2 3))) c d) 0 2)
(A B (1 3 (1 2 3 (1 2 3))) C D)
> (del-lev-n '(a b (1 2 3 (1 2 3 (1 2 3))) c d) 1 2)
(A B (1 2 3 (1 3 (1 2 3))) C D)
> (del-lev-n '(a b (1 2 3 (1 2 3 (1 2 3))) c d) 2 2)
(A B (1 2 3 (1 2 3 (1 3))) C D)
> (del-lev-n '(a b (1 2 3 (1 2 3 (1 2 3))) c d) 3 2)
(A B (1 2 3 (1 2 3 (1 2 3))) C D)

```

Решение 14.13.4

```

(defun del-lev-n (w n m &aux (a (car w)) (d (cdr w)))
  (when w (cons (cond ((atom a) a)
                    ((= n 0) (del-n a m))
                    ((del-lev-n a (1- n) m)))
                (del-lev-n d n m))))

(defun del-n (w n)
  (when w (if (= n 1)
              (cdr w)
              (cons (car w) (del-n (cdr w) (1- n))))))

```



```

> (del-lev-n '(a b (1 2 3 (1 2 3 (1 2 3))) c d) 0 2)
(A B (1 3 (1 2 3 (1 2 3))) C D)
> (del-lev-n '(a b (1 2 3 (1 2 3 (1 2 3))) c d) 1 2)
(A B (1 2 3 (1 3 (1 2 3))) C D)
> (del-lev-n '(a b (1 2 3 (1 2 3 (1 2 3))) c d) 2 2)
(A B (1 2 3 (1 2 3 (1 3))) C D)
> (del-lev-n '(a b (1 2 3 (1 2 3 (1 2 3))) c d) 3 2)
(A B (1 2 3 (1 2 3 (1 2 3))) C D)

```

Задача 14.14 atoms-n-m.lisp

Определить функцию, возвращающую список атомов в заданном интервале уровней. Функция принимает три параметра: список, начальный уровень и конечный уровень.

Решение 14.14.1

```

(defun atom-level (w &optional (n 0) &aux (a (car w)) (d (cdr w)))
  (cond ((null w) nil)
        ((atom a) (cons (list a n) (atom-level d n)))
        ((nconc (atom-level a (1+ n)) (atom-level d n)))))

(defun atoms-n-m (w n m)
  (loop for a in (atom-level w)
        when (<= n (cadr a) m)
        collect (car a)))

> (atoms-n-m '(a b (((c d)) e f)) 0 3)
(A B C D E F)
> (atoms-n-m '(a b (((c d)) e f)) 1 3)
(C D E F)

```

Решение 14.14.2

```

(defun atom-level (w &optional (n 0) &aux (a (car w)) (d (cdr w)))
  (cond ((null w) nil)
        ((atom a) (cons (list a n) (atom-level d n)))
        ((nconc (atom-level a (1+ n)) (atom-level d n)))))

(defun atoms (w n m)
  (cond ((null w) nil)
        ((<= n (cadr w) m) (cons (caar w) (atoms (cdr w) n m)))
        ((atoms (cdr w) n m))))

(defun atoms-n-m (w n m) (atoms (atom-level w) n m))

> (atoms-n-m '(a b (((c d)) e f)) 0 3)
(A B C D E F)
> (atoms-n-m '(a b (((c d)) e f)) 1 3)
(C D E F)

```

Решение 14.14.3

```
(defun atom-level (w &optional (n 0) &aux (a (car w)) (d (cdr w)))
  (cond ((null w) nil)
        ((atom a) (cons (list a n) (atom-level d n)))
        ((nconc (atom-level a (1+ n)) (atom-level d n)))))
```

```
(defun atoms (w n m &aux (a (car w)))
  (cond ((null w) nil)
        ((<= n (cadr a) m) (cons (car a) (atoms (cdr w) n m)))
        ((atoms (cdr w) n m))))
```

```
(defun atoms-n-m (w n m) (atoms (atom-level w) n m))
```

```
> (atoms-n-m '(a b ((c d)) e f)) 0 3)
(A B C D E F)
> (atoms-n-m '(a b ((c d)) e f)) 1 3)
(C D E F)
```

Решение 14.14.4

```
(defun atom-level (w &optional (n 0) &aux (a (car w)) (d (cdr w)))
  (when w (nconc (if (atom a) (list (list a n)) (atom-level a (1+ n)))
                 (atom-level d n))))
```

```
(defun atoms (w n m &aux (a (car w)))
  (cond ((null w) nil)
        ((<= n (cadr a) m) (cons (car a) (atoms (cdr w) n m)))
        ((atoms (cdr w) n m))))
```

```
(defun atoms-n-m (w n m) (atoms (atom-level w) n m))
```

```
> (atoms-n-m '(a b ((c d)) e f)) 0 3)
(A B C D E F)
> (atoms-n-m '(a b ((c d)) e f)) 1 3)
(C D E F)
```

Решение 14.14.5

```
(defun atom-level (w &optional (n 0) &aux (a (car w)) (d (cdr w)))
  (when w (nconc (if (atom a) `((,a ,n)) (atom-level a (1+ n)))
                 (atom-level d n))))
```

```
(defun atoms (w n m &aux (a (car w)))
  (cond ((null w) nil)
        ((<= n (cadr a) m) (cons (car a) (atoms (cdr w) n m)))
        ((atoms (cdr w) n m))))
```

```
(defun atoms-n-m (w n m) (atoms (atom-level w) n m))
```

```
> (atoms-n-m '(a b ((c d)) e f)) 0 3)
(A B C D E F)
> (atoms-n-m '(a b ((c d)) e f)) 1 3)
(C D E F)
```

Решение 14.14.6

```
(defun atom-level (w &optional (n 0) &aux (a (car w)) (d (cdr w)))
  (when w (nconc (if (atom a) `((,a ,n)) (atom-level a (1+ n)))
    (atom-level d n))))

(defun atoms-n-m (w n m)
  (loop for a in (atom-level w)
    when (<= n (cadr a) m)
    collect (car a)))

> (atoms-n-m '(a b (((c d)) e f)) 0 3)
(A B C D E F)
> (atoms-n-m '(a b (((c d)) e f)) 1 3)
(C D E F)
```

Решение 14.14.7

```
(defun atom-level (w &optional (n 0) &aux (a (car w)))
  (when w (nconc (if (atom a) `((,a ,n)) (atom-level a (1+ n)))
    (atom-level (cdr w) n))))

(defun atoms-n-m (w n m)
  (loop for a in (atom-level w)
    when (<= n (cadr a) m)
    collect (car a)))

> (atoms-n-m '(a b (((c d)) e f)) 0 3)
(A B C D E F)
> (atoms-n-m '(a b (((c d)) e f)) 1 3)
(C D E F)
```

Решение 14.14.8

```
(defun atoms-n-m (w n m)
  (loop for a in
    (labels ((atom-level (v n)
      (when v (nconc (if (atom (car v))
        `((,(car v) ,n))
        (atom-level (car v) (1+ n)))
      (atom-level (cdr v) n))))
    (atom-level w 0))
    when (<= n (cadr a) m)
    collect (car a)))

> (atoms-n-m '(a b (((c d)) e f)) 0 3)
(A B C D E F)
> (atoms-n-m '(a b (((c d)) e f)) 1 3)
(C D E F)
```

Задача 14.15 *remove-n-m.lisp*

Определить функцию, удаляющую из списка все элементы с *n*-ного по *m*-ный.

Решение 14.15.1

```
(defun remove-n-m (w n m &optional (c 0))
  (cond ((null w) nil)
        ((<= n c m) (remove-n-m (cdr w) n m (1+ c)))
        ((cons (car w) (remove-n-m (cdr w) n m (1+ c))))))
```

```
> (remove-n-m '(a b c d e f g h) 2 5)
(A B G H)
```

Решение 14.15.2

```
(defun remove-n-m (w n m &optional (c 0))
  (when w (if (<= n c m)
              (remove-n-m (cdr w) n m (1+ c))
              (cons (car w)
                    (remove-n-m (cdr w) n m (1+ c))))))
```

```
> (remove-n-m '(a b c d e f g h) 2 5)
(A B G H)
```

Решение 14.15.3

```
(defun remove-n-m (w n m)
  (loop for a in w
        for b upto (1- (length w))
        unless (<= n b m) collect a))
```

```
> (remove-n-m '(a b c d e f g h) 2 5)
(A B G H)
```

Решение 14.15.4

```
(defun remove-n-m (w n m &optional (c 0))
  (when w (nconc (unless (<= n c m) `(, (car w)))
                 (remove-n-m (cdr w) n m (1+ c)))))
```

```
> (remove-n-m '(a b c d e f g h) 2 5)
(A B G H)
```

Задача 14.16 *meet-nums.lisp*

Определить функцию, вычисляющую количество последовательностей *n*-поряд идущих элементов числового списка, в каждой из которых все элементы попарно удовлетворяют заданному условию.

Решение 14.16.1

```

(defun ch (a w p) (every #'(lambda (e) (funcall p a e)) w))

(defun check (w p)
  (cond ((null w)
        ((ch (car w) (cdr w) p) (check (cdr w) p))
        (t nil)))

(defun meet-nums (w n p)
  (cond ((null (cddr w)) 0)
        ((check (subseq w 0 n) p)
         (+ 1 (meet-nums (cdr w) n p)))
        ((meet-nums (cdr w) n p))))

> (meet-nums '(1 1 3 5 4 5 2 3 2 2 1 1 1) 3 #'/=)
4

```

Решение 14.16.2

```

(defun ch (a w p) (every #'(lambda (e) (funcall p a e)) w))

(defun check (w p)
  (cond ((null w)
        ((ch (car w) (cdr w) p) (check (cdr w) p))))

(defun meet-nums (w n p)
  (cond ((null (cddr w)) 0)
        ((check (subseq w 0 n) p)
         (+ 1 (meet-nums (cdr w) n p)))
        ((meet-nums (cdr w) n p))))

> (meet-nums '(1 1 3 5 4 5 2 3 2 2 1 1 1) 3 #'/=)
4

```

Решение 14.16.3

```

(defun ch (a w p) (every #'(lambda (e) (funcall p a e)) w))

(defun check (w p)
  (if w (when (ch (car w) (cdr w) p) (check (cdr w) p)) t))

(defun meet-nums (w n p)
  (cond ((null (cddr w)) 0)
        ((check (subseq w 0 n) p)
         (+ 1 (meet-nums (cdr w) n p)))
        ((meet-nums (cdr w) n p))))

> (meet-nums '(1 1 3 5 4 5 2 3 2 2 1 1 1) 3 #'/=)
4

```

Решение 14.16.4

```

(defun ch (a w p) (every #'(lambda (e) (funcall p a e)) w))

```

```

(defun check (w p)
  (if w (when (ch (car w) (cdr w) p) (check (cdr w) p)) t))

(defun meet-nums (w n p)
  (if (cddr w)
      (if (check (subseq w 0 n) p)
          (+ 1 (meet-nums (cdr w) n p))
          (meet-nums (cdr w) n p))
      0))

> (meet-nums '(1 1 3 5 4 5 2 3 2 2 1 1 1) 3 #'/=)
4

```

Решение 14.16.5

```

(defun ch (a w p) (every #'(lambda (e) (funcall p a e)) w))

(defun check (w p)
  (if w (when (ch (car w) (cdr w) p) (check (cdr w) p)) t))

(defun meet-nums (w n p)
  (cond ((null (cddr w)) 0)
        ((+ (if (check (subseq w 0 n) p) 1 0)
             (meet-nums (cdr w) n p)))))

> (meet-nums '(1 1 3 5 4 5 2 3 2 2 1 1 1) 3 #'/=)
4

```

Решение 14.16.6

```

(defun ch (a w p) (every #'(lambda (e) (funcall p a e)) w))

(defun check (w p)
  (if w (when (ch (car w) (cdr w) p) (check (cdr w) p)) t))

(defun meet-nums (w n p)
  (if (cddr w)
      (+ (if (check (subseq w 0 n) p) 1 0)
         (meet-nums (cdr w) n p))
      0))

> (meet-nums '(1 1 3 5 4 5 2 3 2 2 1 1 1) 3 #'/=)
4

```

Задача 14.17 in-the-wake.lisp

Определить функцию, добавляющую атом в исходный список после атомов, обладающих заданным свойством.

Решение 14.17.1

```
(defun in-the-wake (w p z)
  (loop for a in w
        if (funcall p a) nconc (list a z)
        else nconc (list a)))

> (in-the-wake '(1 2 3 4 5) #'evenp 'a)
(1 2 A 3 4 A 5)
```

Решение 14.17.2

```
(defun in-the-wake (w p z)
  (loop for a in w
        if (funcall p a) nconc (list a z)
        else collect a))

> (in-the-wake '(1 2 3 4 5) #'evenp 'a)
(1 2 A 3 4 A 5)
```

Решение 14.17.3

```
(defun in-the-wake (w p z)
  (loop for a in w collect a
        when (funcall p a) collect z))

> (in-the-wake '(1 2 3 4 5) #'evenp 'a)
(1 2 A 3 4 A 5)
```

Задача 14.18 range-evenp.lisp

Определить функцию, которая выдает список четных чисел, расположенных между n и m включительно. Причем n может быть $> m$.

Решение 14.18.1

```
(defun range-evenp (n m w) (cond ((= n m) (when (evenp n) n))
                                   ((< n m) (range n m w))
                                   ((range m n w))))

(defun range (n m w &aux (a (car w)))
  (cond ((null w) nil)
        ((and (<= n a m) (evenp a)) (cons a (range n m (cdr w))))
        ((range n m (cdr w)))))

> (range-evenp 2 7 '(1 2 3 4))
(2 4)
```

Решение 14.18.2

```
(defun range-evenp (n m w) (cond ((= n m) (when (evenp n) n))
                                   ((< n m) (range n m w))
                                   ((range m n w))))
```

```
(defun range (n m w &optional acc &aux (a (car w)))
  (cond ((null w) (nreverse acc))
        ((and (<= n a m) (evenp a)) (range n m (cdr w) (cons a acc)))
        ((range n m (cdr w) acc))))
```

```
> (range-evenp 2 7 '(1 2 3 4))
(2 4)
```

Решение 14.18.3

```
(defun range-evenp (n m w) (cond ((= n m) (when (evenp n) n))
                                   (< n m) (range n m w))
                                   ((range m n w))))
```

```
(defun range (n m w &optional acc &aux (a (car w)))
  (if w
      (range n m (cdr w) (if (and (<= n a m) (evenp a))
                              (cons a acc)
                              acc))
      (nreverse acc)))
```

```
> (range-evenp 2 7 '(1 2 3 4))
(2 4)
```

Решение 14.18.4

```
(defun range-evenp (n m w) (cond ((= n m) (when (evenp n) n))
                                   (< n m) (range n m w))
                                   ((range m n w))))
```

```
(defun range (n m w)
  (remove-if-not #'(lambda (a) (and (<= n a m) (evenp a))) w))
```

```
> (range-evenp 2 7 '(1 2 3 4))
(2 4)
```

Задача 14.19 substitute-all.lisp

Определить функцию, заменяющую все вхождения одного атома на другой атом на верхнем уровне.

Решение 14.19.1

```
(defun substitute-all (a b w)
  (cond ((null w) nil)
        ((eq (car w) b) (cons a (substitute-all a b (cdr w))))
        (t (cons (car w) (substitute-all a b (cdr w))))))
```

```
> (substitute-all 'a 'b '(b b))
(A A)
```

Решение 14.19.2


```
(defun substitute-all (a b w)
  (when w (cons (if (eq (car w) b) a (car w))
                 (substitute-all a b (cdr w)))))

> (substitute-all 'a 'b '(b b))
(A A)
```

Решение 14.19.3

```
(defun substitute-all (a b w)
  (substitute a b w))

> (substitute-all 'a 'b '(b b))
(A A)
```

Задача 14.20 substitute-first.lisp

Определить функцию, заменяющую только первое вхождение одного атома на другой атом на верхнем уровне.

Решение 14.20.1

```
(defun substitute-first (a b w)
  (cond ((null w) nil)
        ((eq (car w) b) (cons a (cdr w)))
        (t (cons (car w) (substitute-first a b (cdr w))))))

> (substitute-first 'a 'b '(b b))
(A B)
```

Решение 14.20.2

```
(defun substitute-first (a b w)
  (substitute a b w :count 1))

> (substitute-first 'a 'b '(b b))
(A B)
```

Задача 14.21 stammer.lisp

Определить функцию, дублирующую заданное количество раз каждый элемент списка, удовлетворяющий некоторому предикату.

Решение 14.21.1

```
(defun stammer (p n w &aux (a (car w)) (d (cdr w)))
  (cond ((null w) nil)
        ((consp a) (cons (stammer p n a) (stammer p n d)))
        ((funcall p a) (nconc (make-list n :initial-element a)
                                (stammer p n d))))
```

```
((cons a (stammer p n d))))
```

```
> (stammer #'minusp 3 '(1 -2 3 (4 -5) 6 ((-8)) 7))
(1 -2 -2 -2 3 (4 -5 -5 -5) 6 ((-8 -8 -8)) 7)
```

Решение 14.21.2

```
(defun stammer (p n w &aux (a (car w)) (d (cdr w)))
  (when w (nconc
    (cond ((consp a) (list (stammer p n a)))
          ((funcall p a) (make-list n :initial-element a))
          ((list a)))
    (stammer p n d))))
```

```
> (stammer #'minusp 3 '(1 -2 3 (4 -5) 6 ((-8)) 7))
(1 -2 -2 -2 3 (4 -5 -5 -5) 6 ((-8 -8 -8)) 7)
```

Решение 14.21.3

```
(defun stammer (p n w &aux (a (car w)))
  (when w (nconc (cond ((consp a) `(,(stammer p n a)))
                    ((funcall p a) (loop repeat n collect a))
                    `(,a)))
    (stammer p n (cdr w)))))
```

```
> (stammer #'minusp 3 '(1 -2 3 (4 -5) 6 ((-8)) 7))
(1 -2 -2 -2 3 (4 -5 -5 -5) 6 ((-8 -8 -8)) 7)
```

Задача 14.22 *dive-next.lisp*

Определить функцию, которая по атому *z* и списку *w* выдает множество всех атомов в *w*, непосредственно следующих за каждым вхождением *z* в *w* на заданном уровне *n*.

Решение 14.22.1

```
(defun dive-elms (n w)
  (cond ((null w) nil)
        ((= n 0) w)
        ((atom (car w)) (dive-elms n (cdr w)))
        ((append (dive-elms (- n 1) (car w)) (dive-elms n (cdr w))))))
```

```
(defun next (w z)
  (cond ((null w) nil)
        ((eq (car w) z) (cons (cadr w) (next (cddr w) z)))
        ((next (cdr w) z))))
```

```
(defun dive-next (n w z)
  (next (dive-elms n w) z))
```

```
> (dive-next 0 '(a 1 (b 10 (c 100)) (b 10 (c 100 (d 1000)) e)) 'a)
(1)
```

```
> (dive-next 1 '(a 1 (b 10 (c 100)) (b 10 (c 100 (d 1000)) e)) 'b)
(10 10)
> (dive-next 2 '(a 1 (b 10 (c 100)) (b 10 (c 100 (d 1000)) e)) 'c)
(100 100)
> (dive-next 3 '(a 1 (b 10 (c 100)) (b 10 (c 100 (d 1000)) e)) 'd)
(1000)
```

Решение 14.22.2

```
(defun dive-elms (n w &aux (a (car w)) (d (cdr w)))
  (cond ((null w) nil)
        ((zerop n) w)
        ((atom a) (dive-elms n d))
        ((nconc (dive-elms (1- n) a) (dive-elms n d)))))

(defun next (w z &aux (a (car w)) (d (cdr w)))
  (cond ((null w) nil)
        ((eq a z) (cons (car d) (next (cdr d) z)))
        ((next d z))))

(defun dive-next (n w z)
  (next (dive-elms n w) z))

> (dive-next 0 '(a 1 (b 10 (c 100)) (b 10 (c 100 (d 1000)) e)) 'a)
(1)
> (dive-next 1 '(a 1 (b 10 (c 100)) (b 10 (c 100 (d 1000)) e)) 'b)
(10 10)
> (dive-next 2 '(a 1 (b 10 (c 100)) (b 10 (c 100 (d 1000)) e)) 'c)
(100 100)
> (dive-next 3 '(a 1 (b 10 (c 100)) (b 10 (c 100 (d 1000)) e)) 'd)
(1000)
```

Задача 14.23 name.lisp

Определить функцию, которая заменяет все вхождения одного атома на другой атом.

Решение 14.23.1

```
(defun name (w n m &aux (a (car w)) (d (cdr w)))
  (cond ((null w) nil)
        ((listp a) (cons (name a n m) (name d n m)))
        ((eq a n) (cons m (name d n m)))
        ((cons a (name d n m)))))

> (name '(defun add-10 (n) (+ n 10)) 'add-10 '10+)
(DEFUN 10+ (N) (+ N 10))
> (DEFUN 10+ (N) (+ N 10))
10+
> (10+ 1000)
1010
> (name '(defun plus (w) (if w (+ (car w) (plus (cdr w))) 0)) 'plus
```

```
'++++)
(DEFUN +++++ (W) (IF W (+ (CAR W) (++++ (CDR W))) 0))
> (DEFUN +++++ (W) (IF W (+ (CAR W) (++++ (CDR W))) 0))
++++
> (++++ ' (1 2 3 4))
10
```

Решение 14.23.2

```
(defun name (w n m &aux (a (car w)) (d (cdr w)))
  (when w (cons (cond ((listp a) (name a n m))
                  ((eq a n) m)
                  (t a))
                (name d n m))))

> (name '(defun add-10 (n) (+ n 10)) 'add-10 '10+)
(DEFUN 10+ (N) (+ N 10))
> (DEFUN 10+ (N) (+ N 10))
10+
> (10+ 1000)
1010
> (name '(defun plus (w) (if w (+ (car w) (plus (cdr w))) 0)) 'plus
'++++)
(DEFUN +++++ (W) (IF W (+ (CAR W) (++++ (CDR W))) 0))
> (DEFUN +++++ (W) (IF W (+ (CAR W) (++++ (CDR W))) 0))
++++
> (++++ ' (1 2 3 4))
10
```

Решение 14.23.3

```
(defun name (w n m &aux (a (car w)))
  (when w (cons (cond ((listp a) (name a n m))
                  ((eq a n) m)
                  (t a))
                (name (cdr w) n m))))

> (name '(defun add-10 (n) (+ n 10)) 'add-10 '10+)
(DEFUN 10+ (N) (+ N 10))
> (DEFUN 10+ (N) (+ N 10))
10+
> (10+ 1000)
1010
> (name '(defun plus (w) (if w (+ (car w) (plus (cdr w))) 0)) 'plus
'++++)
(DEFUN +++++ (W) (IF W (+ (CAR W) (++++ (CDR W))) 0))
> (DEFUN +++++ (W) (IF W (+ (CAR W) (++++ (CDR W))) 0))
++++
> (++++ ' (1 2 3 4))
10
```

Задача 14.24 change.lisp

Определить функцию от трех аргументов *w*, *e* и *z*, которая в простом списке *w* заменяет все атомы *e* на атомы *z* и наоборот *z* на *e*.

Решение 14.24.1

```
(defun change (w e z &aux (a (car w)))
  (when w (cons (cond ((eq a e) z)
                    ((eq a z) e)
                    (a))
                (change (cdr w) e z))))
```

```
> (change '(1 2 3 1 2 3) 1 2)
(2 1 3 2 1 3)
```

Решение 14.24.2

```
(defun change (w e z)
  (when w (cons (cond ((eq (car w) e) z)
                    ((eq (car w) z) e)
                    ((car w)))
                (change (cdr w) e z))))
```

```
> (change '(1 2 3 1 2 3) 1 2)
(2 1 3 2 1 3)
```

Решение 14.24.3

```
(defun change (w e z)
  (loop for a in w collect (cond ((eq a e) z)
                                ((eq a z) e)
                                (a))))
```

```
> (change '(1 2 3 1 2 3) 1 2)
(2 1 3 2 1 3)
```

Решение 14.24.4

```
(defun change (w e z)
  (mapcar #'(lambda (a) (cond ((eq a e) z)
                              ((eq a z) e)
                              (a)))
          w))
```

```
> (change '(1 2 3 1 2 3) 1 2)
(2 1 3 2 1 3)
```

Задача 14.25 range.lisp

Удалить элементы списка вне диапазона, заданного минимальным и максимальным значением.

Решение 14.25.1

```
(defun range (n m w &aux (a (car w)))
  (cond ((null w) nil)
        ((<= n a m) (cons a (range n m (cdr w))))
        (t (range n m (cdr w)))))
```

```
> (range 1 3 '(1 2 3 -4 7 2 11 5))
(1 2 3 2)
```

Решение 14.25.2

```
(defun range (n m w &optional ac &aux (a (car w)))
  (if w (range n m (cdr w) (if (<= n a m) (cons a ac) ac))
      (nreverse ac)))
```

```
> (range 1 3 '(1 2 3 -4 7 2 11 5))
(1 2 3 2)
```

Решение 14.25.3

```
(defun range (n m w)
  (loop for a in w when (<= n a m) collect a))
```

```
> (range 1 3 '(1 2 3 -4 7 2 11 5))
(1 2 3 2)
```

Решение 14.25.4

```
(defun range (n m w) (remove-if-not #'(lambda (a) (<= n a m)) w))
```

```
> (range 1 3 '(1 2 3 -4 7 2 11 5))
(1 2 3 2)
```

Задача 14.26 drop.lisp

Удалить четные элементы с номерами *n* и *m* и выполнить условие с вложенными списками.

Решение 14.26.1

```
(defun drop (w n m)
  (loop for a in w for b from 1
        if (listp a) collect (drop a n m)
        else unless (and (numberp a)
                          (evenp a)
                          (or (= b n) (= b m))))
  collect a))
```

```
> (drop '(a b (a b c 4 d e 5) 4 c d 5 e) 4 7)
(A B (A B C D E 5) C D 5 E)
```

Задача 14.27 *swap.lisp*

The nature of Lisp is that you have functions (which are first-class object), macros, symbolic bindings to values and functions, lists, which can contain functions, values, and other lists, and I think most importantly, metacircular evaluation. These are embodied in functions like eval and apply (you'll learn about those eventually). A dominant characteristic is that there is a subtle difference between code and data. This is commonly called "homoiconicity." It's a sophisticated way of saying that functions and data are made up of the same stuff. It's just a matter of how they are interpreted from moment to moment. What this means is that everything you use in the language is made up of the above elements—everything. In traditional procedural languages, you have reserved words, or commands, and then you have data. Commands and data are made of different stuff, and you have no means of adding to, or altering the command set. In Lisp, they're made out of the same stuff. So, it's possible to change everything in it (depending on how much leeway the dialect's implementation allows). This is what gives many beginners to Lisp headaches (I went through this as well), because if you've been programming in procedural languages for years, this just doesn't make any sense. How can data be code?!... When you're first learning it, you don't have to worry about what I just said. That can come later. Once you realize what I just said means, though, it opens up a new way of seeing what programming can be: You can kind of write your own ticket. You can write your own language, with its own semantics.

Mark Miller

Определить функцию, меняющую два атома местами по заданным номерам.

Решение 14.27.1

```
(defun swap (n m w) (rotatef (nth n w) (nth m w)) w)

> (swap 2 5 '(1 2 3 4 5 6 7))
(1 2 6 4 5 3 7)
```

Задача 14.28 *blast-np-elm.lisp*

You learn programming like a pro fastest by understanding the LISP code on page 13: Lisp as the Maxwell's equations of software. Just continue reading that article, even if you don't understand a thing. Don't look up things you don't understand. Read it a few times over and over until it makes more sense. If you understand that section of code, you can program anything. You will soon realise that most programming languages are similar and that you have understood the "Maxwell's equations of software", therefore you are able to get a hold of almost any popular programming language pretty quickly.

~

Определить функцию (f w n p), которая удаляет n элементов с конца списка w, удовлетворяющих некоторому предикату p.

Решение 14.28.1

```
(defun blast-np-elm (w n p) (remove-if p w :count n :from-end t))

> (blast-np-elm '(1 2 3 4 5 6 7 8) 2 #'evenp)
(1 2 3 4 5 7)
```

Задача 14.29 *swap.lisp*

As a side note, the language with which I had the most fun was LISP, it has since evolved into Common LISP (Welcome to Common-Lisp.net!). It is just as old of language as FORTRAN and COBOL. but totally different. In fact, it will warp your coding techniques and think in ways that you may never have thought before.

Wayne Cook

Дан односвязный список. Требуется напечатать исходный список. Поменять местами два элемента списка, номера которых введёт пользователь с клавиатуры. Напечатать получившийся список.

Решение 14.29.1

```
(defun swap (w) (print w) (rotatef (nth (read) w) (nth (read) w)) w)

> (swap '(a b c d))
(A B C D) 1
```


2
(A C B D)

Задача 14.30 *modify.lisp*

Lisp, Haskell, or ML - This will force you to think about programming in a completely different way. Your problem solving skills will improve by knowing how to think in different programming paradigms.

Thomas Amundsen

Определить функцию, заменяющую в списке *w* первое вхождение *a* на *b*.

Решение 14.30.1

```
(defun modify (w a b)
  (cond ((null w) nil)
        ((equalp (car w) a) (cons b (cdr w)))
        (t (cons (car w) (modify (cdr w) a b)))))
```

```
> (modify '(1 2 3 1 2) 2 20)
(1 20 3 1 2)
```

Решение 14.30.2

```
(defun modify (w a b)
  (when w (if (equalp (car w) a) (cons b (cdr w))
              (cons (car w) (modify (cdr w) a b)))))
```

```
> (modify '(1 2 3 1 2) 2 20)
(1 20 3 1 2)
```

Задача 14.31 *re.lisp*

Arguably Lisp (or Clojure) is the easiest programming language to learn since it has the least syntax of any of the languages.

Russel Winder

Определить функцию трех аргументов (список, старый атом, новый атом), возвращающую список, в котором все вхождения старого элемента заменены на новый элемент на всех уровнях списка.

Решение 14.31.1

```
(defun re (w u e &aux (a (car w)))
  (when w (cons (cond ((listp a) (re a u e))
                  ((eq a u) e)
                  (a)))))
```

```

      (a))
    (re (cdr w) u e)))

```

```

> (re '(q w e (q w ((e)))) 'e 'z)
(Q W Z (Q W ((Z))))

```

Решение 14.31.2

```

(defun re (w u e)
  (loop for a in w collect
    (cond ((listp a) (re a u e))
          ((eq a u) e)
          (a))))

```

```

> (re '(q w e (q w ((e)))) 'e 'z)
(Q W Z (Q W ((Z))))

```

Решение 14.31.3

```

(defun re (w u e)
  (loop for a in w
    collect (typecase a
      (list (re a u e))
      (otherwise (if (eq a u) e a)))))

```

```

> (re '(q w e (q w ((e)))) 'e 'z)
(Q W Z (Q W ((Z))))

```

Решение 14.31.4

```

(defun re (w u e)
  (loop for a in w
    collect (typecase a
      (list (re a u e))
      (t (if (eq a u) e a)))))

```

```

> (re '(q w e (q w ((e)))) 'e 'z)
(Q W Z (Q W ((Z))))

```

Задача 14.32 srch.lisp

Определить функцию, отбирающую числовые элементы двумерного массива, принадлежащие заданному интервалу.

Решение 14.32.1

```

(defun srch (n m arr)
  (loop for a in (array>list arr)
    when (<= n a m) collect a))

(defun array>list (a)
  (loop for i below (array-total-size a)

```

```

collect (row-major-aref a i)))

> (srch 20 40 #2A((11 44 55 55) (22 33 44 55)))
(22 33)
> (srch 5 7 #2A((7 8 2 3 1) (5 3 2 6 3) (9 3 5 2 0)))
(7 5 6 5)
> (srch 2 6 #2A((7 8 2 3 1) (5 3 2 6 3) (9 3 5 2 0)))
(2 3 5 3 2 6 3 3 5 2)

```

Задача 14.33 *alter.lisp*

One of the many fun and profound things about Lisp is that it only has a few primitive "functions" — car, cdr, cons, atom, equal, lambda, read, etc. — and most of these are extremely simple and small in any typical machine code. One route would be to code these by hand and get a minimal but universal Lisp going to then use as the basis for the next levels of HLLs.

Alan Kay

Дан список *w* и два числа *m* и *n*. Если последний элемент списка чётное число, то поменять местами элементы с номерами *m* и *n*. Иначе — вывести список, в котором элемент с номерами *m* будет возведён в степень *n*.

Решение 14.33.1

```

(defun alter (w m n &aux (v (copy-list w)))
  (cond ((evenp (car (last v)))
         (rotatef (nth (1- m) v) (nth (1- n) v)) v)
        ((setf (nth (1- m) v)
                 (expt (nth (1- m) v) n)) v)))

> (alter '(5 2 3 4 8 0 2 4 1 6) 3 9)
(5 2 1 4 8 0 2 4 3 6)
> (alter '(2 3 8 4 0 1) 2 3)
(2 27 8 4 0 1)

```

Решение 14.33.2

```

(defun alter (w m n &aux (v (copy-list w)))
  (if (evenp (car (last v)))
      (rotatef (nth (1- m) v) (nth (1- n) v))
      (setf (nth (1- m) v) (expt (nth (1- m) v) n)))
  v)

> (alter '(5 2 3 4 8 0 2 4 1 6) 3 9)
(5 2 1 4 8 0 2 4 3 6)
> (alter '(2 3 8 4 0 1) 2 3)
(2 27 8 4 0 1)

```

Задача 14.34 insert.lisp

Show me your ~/.emacs and I will tell you who you are.

Bogdan Maryniuk

Определить функцию, вставляющую элемент в определённое место списка.

Решение 14.34.1

```
(defun insert (w e p)
  (if (or (zerop p) (null w)) (cons e w)
      (cons (car w) (insert (cdr w) e (1- p)))))
```

```
> (insert '(1 2 3 4 5 6) 'a 2)
(1 2 A 3 4 5 6)
```

Решение 14.34.2

```
(defun insert (e n w)
  (if (and (plusp n) w) (cons (car w) (insert e (1- n) (cdr w)))
      (cons e w)))
```

```
> (insert 'z 4 '(0 1 2 3 4 5 6))
(0 1 2 3 Z 4 5 6)
```

Задача 14.35 sbst.lisp

When you choose a language, you're choosing more than a set of technical trade-offs—you're choosing a community.

Joshua Bloch

Keep your code absolutely simple. Keep looking at your functions and figure out how you simplify further.

John Romero

Определить функцию, которая в одномерном списке w заменяет элемент a на элемент b.

Решение 14.35.1

```
(defun sbst (w a b)
  (cond ((null w) nil)
        ((equalp (car w) a) (cons b (sbst (cdr w) a b)))
        (t (cons (car w) (sbst (cdr w) a b)))))
```

```
> (sbst '(a b c a b c) 'c 'z)
(a b z c a b c)
```

```
(A B Z A B Z)
```

Решение 14.35.2

```
(defun sbst (w a b)
  ((lambda (w a b e)
    (when w (cons (if (equalp e a) b e) (sbst (cdr w) a b))))
   w a b (car w)))

> (sbst '(a b c a b c) 'c 'z)
(A B Z A B Z)
```

Решение 14.35.3

```
(defun sbst (w a b)
  (when w ((lambda (w a b e)
    (cons (if (equalp e a) b e) (sbst (cdr w) a b)))
   w a b (car w))))

> (sbst '(a b c a b c) 'c 'z)
(A B Z A B Z)
```

Решение 14.35.4

```
(defun sbst (w a b)
  (let ((e (car w)))
    (when w (cons (if (equalp e a) b e) (sbst (cdr w) a b)))))

> (sbst '(a b c a b c) 'c 'z)
(A B Z A B Z)
```

Решение 14.35.5

```
(defun sbst (w a b)
  (when w (let ((e (car w)))
    (cons (if (equalp e a) b e) (sbst (cdr w) a b)))))

> (sbst '(a b c a b c) 'c 'z)
(A B Z A B Z)
```

Решение 14.35.6

```
(defun sbst (w a b)
  (when w (cons (if (equalp (car w) a) b (car w))
    (sbst (cdr w) a b))))

> (sbst '(a b c a b c) 'c 'z)
(A B Z A B Z)
```

Решение 14.35.7

```
(defun sbst (w a b &aux (e (car w)))
  (when w (cons (if (equalp e a) b e) (sbst (cdr w) a b))))
```

```
> (subst '(a b c a b c) 'c 'z)
(A B Z A B Z)
```

Решение 14.35.8

```
(defun sbst (w a b)
  (loop for e in w collect (if (equalp e a) b e)))

> (subst '(a b c a b c) 'c 'z)
(A B Z A B Z)
```

Решение 14.35.9

```
(defun sbst (w a b)
  (mapcar #'(lambda (e) (if (equalp e a) b e)) w))

> (subst '(a b c a b c) 'c 'z)
(A B Z A B Z)
```

Решение 14.35.10

```
(defun sbst (w a b) (substitute b a w))

> (subst '(a b c a b c) 'c 'z)
(A B Z A B Z)
```

Задача 14.36 *select-p.lisp*

Определите функцию `(f n v p)`, которая выдает список всех элементов списка `v`, удовлетворяющих некоторому предикату `p` и встречающихся в исходном списке более `n` раз.

Решение 14.36.1

```
(defun select-p (n w p &optional v &aux (a (car w)))
  (cond ((null w) (reverse v))
        ((and (not (member a v)) (funcall p a) (> (count a w) n))
         (select-p n (cdr w) p (cons a v)))
        ((select-p n (cdr w) p v))))

> (selectv-p 1 '(0 1 0 1) #'oddp)
(1)
```

Структура 15 (функция (СПИСОК) (СПИСОК) АТОМ) > АТОМ

What people "want" is a function of what they learn is available. If you wish to sell something, you'd better understand that you can't give people what they want in the market today, because what they want today is what they can already get. You have to discover what they really want, and find some way to give that physical shape.

Erik Naggum

Programming in Lisp is like playing with the primordial forces of the universe. It feels like lightning between your fingertips. No other language even feels close.

Glenn Ehrlich

У хорошего путешественника нет точных планов и намерения попасть куда-то.

Лао Цзы

Learn Emacs in ten years.

Theresa O'Connor

The last dinosaurs lived closer to us than they did to the first dinosaurs.

~

Задача 15.1 *capital.lisp*

Дан список стран *w* и список столиц этих стран *v*. По заданной стране *c* выдать название столицы.

Решение 15.1.1

```
(defun capital (c w v &aux (m (position c w))) (when m (nth m v)))  
  
> (capital 'Germany '(RF USA Germany UK) '(Moscow Washington Berlin  
London))  
BERLIN
```

Решение 15.1.2

```
(defun capital (c w v)
  (car (loop for a in w
            for b in v when (equal a c) collect b)))

> (capital 'Germany '(RF USA Germany UK) '(Moscow Washington Berlin
London))
BERLIN
```

Решение 15.1.3

```
(defun capital (c w v)
  ((lambda (a b)
    (cond ((null w) nil)
          ((eq a c) b)
          ((capital c (cdr w) (cdr v))))))
  (car w) (car v)))

> (capital 'Germany '(RF USA Germany UK) '(Moscow Washington Berlin
London))
BERLIN
```

Решение 15.1.4

```
(defun capital (c w v)
  (car (loop for (a b) in (mapcar #'list w v)
            when (eq a c) collect b)))

> (capital 'Germany '(RF USA Germany UK) '(Moscow Washington Berlin
London))
BERLIN
```


Структура 16 (функция (СПИСОК) (СПИСОК) АТОМ) > (СПИСОК)

Lisp was far more powerful and flexible than any other language of its day; in fact, it is still a better design than most languages of today, twenty-five years later. Lisp freed ITS's hackers to think in unusual and creative ways. It was a major factor in their successes, and remains one of hackerdom's favorite languages.

Eric Raymond

The most powerful programming language is Lisp. If you don't know Lisp (or its variant, Scheme), you don't appreciate what a powerful language is. Once you learn Lisp you will see what is missing in most other languages.

Richard Stallman

Задача 16.1 *gen-arrangement.lisp*

Определить функцию генерации всех возможных расстановок знаков арифметических операций в заданном шестизначном числе так, чтобы результатом полученного выражения было число 100 (цифры исходного числа допускаются задавать через пробел).

Решение 16.1.1

```
(defun pm (v &optional r)
  (dolist (a v r)
    (dolist (b v)
      (dolist (c v)
        (dolist (d v)
          (dolist (e v)
            (push (list a b c d e) r)))))))

(defun mx (w v)
  (when w (cons (car w) (mx v (cdr w)))))

(defun th (w p)
  (subseq w (1- p) (+ p 2)))

(defun ta (w p)
  (subseq w (+ p 2)))
```

```

(defun tb (w p)
  (subseq w 0 (1- p)))

(defun pu (w)
  (list (cadr w) (car w) (caddr w)))

(defun sb (z w &aux (p (position z w)))
  (append (tb w p) (list (eval (pu (th w p))) (ta w p))))

(defun ev (w)
  (cond ((null (cdr w)) (car w))
        ((member '/' w) (ev (sb '/' w)))
        ((member '* w) (ev (sb '* w)))
        ((member '+ w) (ev (sb '+ w)))
        ((ev (sb '- w)))))

(defun gen-arrangement (w v z)
  (mapcar
   #'(lambda (e)
       (mx w e)) (mapcan
                  #'(lambda (a)
                      (when (= (ev (mx w a)) z)
                        (list a)))
                  (pm v)))))

> (gen-arrangement '(5 2 3 4 5 6) '(+ - * /) 100)
((5 / 2 / 3 * 4 * 5 * 6) (5 * 2 * 3 * 4 * 5 / 6))
> (gen-arrangement '(5 2 3 4 5 6) '(+ - * /) 50)
((5 + 2 * 3 / 4 * 5 * 6))

```

Примеры по функциям:

```

> (pm '(+ - * /))
((/ / / /) (/ / / / *) ... (+ + + + *) (+ + + + -) (+ + + + +))
mx - смешивает поочередно знаки и числа:
> (mx '(2 5 7 4 2 4) '(+ * - / +))
(2 + 5 * 7 - 4 / 2 + 4);
> (mx '(5 2 3 4 5 6) '(+ + + + *))
(5 + 2 + 3 + 4 + 5 * 6)
> (th '(5 + 2 + 3 + 4 + 5 * 6) 9)
(5 * 6)
> (ta '(5 + 2 + 3 + 4 + 5 * 6) 9)
NIL
> (tb '(5 + 2 + 3 + 4 + 5 * 6) 9)
(5 + 2 + 3 + 4 +)
> (pu '(5 * 6))
(* 5 6)
> (sb '* '(5 + 2 + 3 + 4 + 5 * 6))
(5 + 2 + 3 + 4 + 30)
> (ev '(5 + 2 + 3 + 4 + 5 * 6))

```

Задача 16.2 insert-els.lisp

In the end, Lisp won me over because it turns out that it is the mother of all languages. You can bend it and turn it into whatever language you want, from the most flexible and reflective interpreted scripting language to the most efficient and static compiled production system.

Pascal Costanza

Напишите функцию, которая вставляла бы на заданное место списка элементы второго списка-аргумента.

Решение 16.2.1

```
(defun insert-els (w v n)
  (cond ((zerop n) (append v w))
        ((cons (car w) (insert-els (cdr w) v (1- n))))))
```

```
> (insert-els '(1 2 3) '(a b) 2)
(1 2 A B 3)
```

Решение 16.2.2

```
(defun insert-els (w v n)
  (cond ((null w) nil)
        ((> n 0) (cons (car w) (insert-els (cdr w) v (1- n))))
        ((nconc v w))))
```

```
> (insert-els '(1 2 3) '(a b) 2)
(1 2 A B 3)
```

Решение 16.2.3

```
(defun insert-els (w v n)
  (if (zerop n)
      (nconc v w)
      (cons (car w) (insert-els (cdr w) v (1- n)))))
```

```
> (insert-els '(1 2 3) '(a b) 2)
(1 2 A B 3)
```

Решение 16.2.4

```
(defun insert-els (w v n)
  (nconc (subseq w 0 n) v (subseq w n)))
```

```
> (insert-els '(1 2 3) '(a b) 2)
(1 2 A B 3)
```

Задача 16.3 *insert-list.lisp*

Опишите функцию, которая вставляла бы на заданное место списка другой список.

Решение 16.3.1

```
(defun insert-list (w v n)
  (if (zerop n)
      (cons v w)
      (cons (car w)
             (insert-list (cdr w) v (1- n)))))

> (insert-list '(1 2 3) '(a b) 2)
(1 2 (A B) 3)
```

Решение 16.3.2

```
(defun insert-list (w v n)
  (loop for a in w
        for i upfrom 0
        when (= i n) collect v
        collect a))

> (insert-list '(1 2 3) '(a b) 2)
(1 2 (A B) 3)
```

Задача 16.4 *bus-ticket.lisp*

Robert Morris wrote the ordering system, which shoppers used to place orders. Trevor Blackwell wrote the image generator and the manager, which merchants used to retrieve orders, view statistics, and configure domain names etc. I wrote the editor, which merchants used to build their sites. The ordering system and image generator were written in C and C++, the manager mostly in Perl, and the editor in Lisp.

Paul Graham on Viaweb

Дан автобусный билет с номером, состоящим из 6 цифр. Расставить между цифрами знаки арифметических операций '+', '-', '*', '/' (целочисленное деление) таким образом, чтобы значение полученного выражения было равно 100.

Решение 16.4.1

```
(defun pm (v &optional r)
  (dolist (a v r)
    (dolist (b v)
```

```

(dolist (c v)
  (dolist (d v)
    (dolist (e v)
      (push (list a b c d e) r))))))

(defun mx (w v)
  (when w (cons (car w) (mx v (cdr w)))))

(defun th (w p)
  (subseq w (1- p) (+ p 2)))

(defun ta (w p)
  (subseq w (+ p 2)))

(defun tb (w p)
  (subseq w 0 (1- p)))

(defun pu (w)
  (list (cadr w) (car w) (caddr w)))

(defun sb (z w &aux (p (position z w)))
  (append (tb w p) (list (eval (pu (th w p))) (ta w p))))

(defun ev (w)
  (cond ((null (cdr w)) (car w))
        ((member '/' w) (ev (sb '/' w)))
        ((member '* w) (ev (sb '* w)))
        ((member '+ w) (ev (sb '+ w)))
        ((ev (sb '- w)))))

(defun bus-ticket (w v z)
  (mapcar
   #'(lambda (e)
       (mx w e)) (mapcan
                  #'(lambda (a)
                      (when (= (ev (mx w a)) z)
                        (list a)))
                  (pm v))))

> (bus-ticket '(5 2 3 4 5 6) '(+ - * /) 100)
((5 / 2 / 3 * 4 * 5 * 6) (5 * 2 * 3 * 4 * 5 / 6))

```

Задача 16.5 run-function.lisp

Даны списки *v* и *w*. В списке *v* все элементы различные. Определить функцию высшего порядка, которая выдает список результатов применения функции к элементу из *v* и списку *w*.

Решение 16.5.1

```

(defun run-function (f v w)
  (mapcar #'(lambda (a) (funcall f a w)) v))

```

```
> (run-function #'count '(a b) '(c a b a d))
(2 1)
```

Задача 16.6 *insert-elms.lisp*

Список *w*, элементами которого являются списки. Заменить *n*-й элемент списка *w* элементами списка *v*.

Решение 16.6.1

```
(defun insert-elms (w n v)
  (cond ((zerop n) (nconc v (cdr w)))
        ((cons (car w) (insert-elms (cdr w) (1- n) v)))))

> (insert-elms '((a b) (c d) (e f)) 1 '(x y z))
((A B) X Y Z (E F))
```

Решение 16.6.2

```
(defun insert-elms (w n v)
  (if (zerop n)
      (nconc v (cdr w))
      (cons (car w) (insert-elms (cdr w) (1- n) v))))

> (insert-elms '((a b) (c d) (e f)) 1 '(x y z))
((A B) X Y Z (E F))
```

Задача 16.7 *insert.lisp*

Would you buy a book proudly stating on the cover that its reader is a dummy? Or would you think "of course it's ironic"?

Erik Naggum

Дан список, который состоит из подсписков. Напишите функцию, которая вставляла бы на заданное место элементы второго списка-аргумента во все подписки данного списка.

Решение 16.7.1

```
(defun insert (v n w)
  (when w (cons (insert-elms v n (car w)) (insert v n (cdr w)))))

(defun insert-elms (v n w)
  (cond ((zerop n) (ncon v w))
        ((cons (car w) (insert-elms v (1- n) (cdr w)))))

> (insert '(9 10) 2 '((1 2 3) (4 5 6) (7 8 9)))
((1 2 9 10 3) (4 5 9 10 6) (7 8 9 10 9))
```

Задача 16.8 insert-after.lisp

On the upside, we certainly have been able to write quite advanced software that we might not have otherwise managed. A million lines of Lisp code, including its fair share of macros and DSLs, would be so many more million lines of code without the syntactic abstraction made possible by Lisp. However hard and expensive it was with Lisp, I can only imagine how many times worse it would have been with anything else.

Faré Rideau

Дан список, который состоит из подсписков. Напишите функцию, которая вставляла бы после *n*-х элементов каждого подсписка элементы второго списка-аргумента.

Решение 16.8.1

```
(defun insert-after (v n w)
  (mapcar #'(lambda (a)
              (if (and (listp a) (nth n a))
                  (insert-elms v n a)
                  a))
          w))

(defun insert-elms (v n w)
  (if (minusp n)
      (append v w)
      (cons (car w) (insert-elms v (1- n) (cdr w))))))

> (insert-after '(9 10) 1 '((1) (4 5) (7 8 9)))
((1) (4 5 9 10) (7 8 9 10 9))
> (insert-after '(9 10) 1 '((1) (4 5) (7 8 9) 11))
((1) (4 5 9 10) (7 8 9 10 9) 11)
```

Решение 16.8.2

```
(defun insert (v n w)
  (mapcar #'(lambda (a)
              (if (listp a) (insert-elms v n a) a)) w))

(defun insert-elms (v n w)
  (if (minusp n)
      (append v w)
      (cons (car w) (insert-elms v (1- n) (cdr w))))))

> (insert '(9 10) 1 '((1) (4 5) (7 8 9) 11))
((1 NIL 9 10) (4 5 9 10) (7 8 9 10 9) 11)
```

Задача 16.9 *dive-change.lisp*

...By policy, LISP has never really catered to mere mortals... And, of course, mere mortals have never really forgiven LISP for not catering to them.

Larry Wall

Написать программу замены атомов на заданном уровне новыми атомами. Новые атомы вводить в виде списка. Атомы, содержащиеся в исходном списке имеют уровень 0, атомы, содержащиеся в следующем вложенном списке, имеют уровень 1 и т. д.

Решение 16.9.1

```
(defparameter c -1)

(defun dive-change (n w v)
  (progn (setf c -1) (dive n w v)))

(defun dive (n w v)
  (cond ((null w) nil)
        ((and (atom (car w)) (zerop n))
         (setf c (1+ c)) (cons (elt v c) (dive n (cdr w) v)) )
        ((listp (car w))
         (cons (dive (1- n) (car w) v) (dive n (cdr w) v)))
        ((cons (car w) (dive n (cdr w) v)))))

> (dive-change 0 '(1 (2 (3)) ((4)) ((5))) ' (a b c d e))
(A (2 (3)) ((4)) ((5)))
> (dive-change 1 '(1 (2 (3)) ((4)) ((5))) ' (a b c d e))
(1 (A (3)) ((4)) ((5)))
```

Решение 16.9.2

```
(defparameter c -1)

(defun dive-change (n w v)
  (progn (setf c -1) (dive n w v)))

(defun dive (n w v)
  (cond ((null w) nil)
        ((and (atom (car w)) (zerop n))
         (setf c (1+ c)) (cons (nth c v) (dive n (cdr w) v)) )
        ((cons (if (listp (car w))
                    (dive (1- n) (car w) v)
                    (car w))
                 (dive n (cdr w) v)))))

> (dive-change 0 '(1 (2 (3)) ((4)) ((5))) ' (a b c d e))
```



```
(A (2 (3)) ((4)) ((5)))
> (dive-change 1 '(1 (2 (3)) ((4)) ((5))) '(a b c d e))
(1 (A (3)) ((4)) ((5)))
```

Решение 16.9.3

```
(defparameter c -1)

(defun dive-change (n w v) (progn (setf c -1) (dive n w v)))

(defun dive (n w v)
  (cond ((null w) nil)
        ((listp (car w))
         (cons (dive (1- n) (car w) v) (dive n (cdr w) v)))
        ((zerop n)
         (setf c (1+ c))
         (cons (elt v c) (dive n (cdr w) v)))
        ((cons (car w) (dive n (cdr w) v)))))

> (dive-change 0 '(1 (2 (3)) ((4)) ((5))) '(a b c d e))
(A (2 (3)) ((4)) ((5)))
> (dive-change 1 '(1 (2 (3)) ((4)) ((5))) '(a b c d e))
(1 (A (3)) ((4)) ((5)))
```

Задача 16.10 set-diff.lisp

In fact Lisp is one of the few languages that even have a macro system, let alone a reader macro system which makes it possible to create any language one desires. Macros in Lisp make it possible for the programmer to create language constructs. If you look at the Common Lisp LOOP macro you will notice that it in itself is a Domain-specific language. That is the power of Lisp: it turns a programmer into a language designer.

Alisa Dolinsky

Определить функцию, которая возвращает список, содержащий элементы, имеющиеся в списке *w*, но отсутствующие в списке *v* и удовлетворяющие некоторому предикату *p*.

Решение 16.10.1

```
(defun set-diff (w v p) (remove-if-not p (set-difference w v)))

> (set-diff '(1 2 3 4 5 6 7) '(1 2 3) #'oddp)
(5 7)
```

Решение 16.10.2

```
(defun set-diff (w v p)
  (loop for a in w
        when (and (not (member a v)) (funcall p a))
        collect a))
```

```
> (set-diff '(1 2 3 4 5 6 7) '(1 2 3) #'oddp)
(5 7)
```

Решение 16.10.3

```
(defun set-diff (w v p)
  (loop for a in w
        unless (or (member a v) (not (funcall p a)))
        collect a))
```

```
> (set-diff '(1 2 3 4 5 6 7) '(1 2 3) #'oddp)
(5 7)
```

Решение 16.10.4

```
(defun set-diff (w v p)
  (cond ((null w) nil)
        ((or (member (car w) v) (not (funcall p (car w))))
         (set-diff (cdr w) v p))
        (t (cons (car w) (set-diff (cdr w) v p)))))
```

```
> (set-diff '(1 2 3 4 5 6 7) '(1 2 3) #'oddp)
(5 7)
```

Решение 16.10.5

```
(defun set-diff (w v p &aux (a (car w)))
  (cond ((null w) nil)
        ((or (member a v) (not (funcall p a))) (set-diff (cdr w) v p))
        ((cons a (set-diff (cdr w) v p)))))
```

```
> (set-diff '(1 2 3 4 5 6 7) '(1 2 3) #'oddp)
(5 7)
```

Решение 16.10.6

```
(defun set-diff (w v p &aux (a (car w)))
  (when w (if (or (member a v) (not (funcall p a)))
              (set-diff (cdr w) v p)
              (cons a (set-diff (cdr w) v p)))))
```

```
> (set-diff '(1 2 3 4 5 6 7) '(1 2 3) #'oddp)
(5 7)
```

Решение 16.10.7

```
(defun set-diff (w v p &optional ac &aux (a (car w)))
  (cond ((null w) (nreverse ac))
```

```

      ((or (member a v) (not (funcall p a)))
       (set-diff (cdr w) v p ac))
      ((set-diff (cdr w) v p (cons a ac)))))

> (set-diff '(1 2 3 4 5 6 7) '(1 2 3) #'oddp)
(5 7)

```

Решение 16.10.8

```

(defun set-diff (w v p &aux (a (car w)))
  (when w (if (and (not (member a v)) (funcall p a))
              (cons a (set-diff (cdr w) v p))
              (set-diff (cdr w) v p))))

> (set-diff '(1 2 3 4 5 6 7) '(1 2 3) #'oddp)
(5 7)

```

Решение 16.10.9

```

(defun set-diff (w v p &optional ac &aux (a (car w)))
  (cond ((null w) (nreverse ac))
        ((and (not (member a v)) (funcall p a))
         (set-diff (cdr w) v p (cons a ac)))
        ((set-diff (cdr w) v p ac))))

> (set-diff '(1 2 3 4 5 6 7) '(1 2 3) #'oddp)
(5 7)

```

Решение 16.10.10

```

(defun set-diff (w v p &optional ac &aux (a (car w)))
  (if w (set-diff
        (cdr w)
        v
        p
        (if (and (not (member a v)) (funcall p a))
            (cons a ac)
            ac))
        (nreverse ac)))

> (set-diff '(1 2 3 4 5 6 7) '(1 2 3) #'oddp)
(5 7)

```

Решение 16.10.11

```

(defun set-diff (w v p &optional ac &aux (a (car w)))
  (if w (set-diff
        (cdr w)
        v
        p
        (if (or (member a v) (not (funcall p a)))
            ac
            (cons a ac)))
        (nreverse ac)))

```

```
(nreverse ac)))
```

```
> (set-diff '(1 2 3 4 5 6 7) '(1 2 3) #'oddp)
(5 7)
```

Решение 16.10.12

```
(defun set-diff (w v p) (set-difference (remove-if-not p w) v))

> (set-diff '(1 2 3 4 5 6 7) '(1 2 3) #'oddp)
(5 7)
```

Задача 16.11 *insert-after-x.lisp*

The greatest thing about Common Lisp is that the standardization process drew from years and years of experience and organic growth. This makes the language feel singularly solid and practical. Most languages feel immature compared to CL.

Marijn Haverbeke

Создать функцию, вставляющую в список *w* элементы списка *v* после каждого вхождения *x* в *w*.

Решение 16.11.1

```
(defun insert-after-x (w x v &aux (a (car w)))
  (cond ((null w) nil)
        ((equalp x a)
         (append (cons a v) (insert-after-x (cdr w) x v)))
        (t (cons a (insert-after-x (cdr w) x v)))))

> (insert-after-x '(1 2 1 5 3) 1 '(+ * z))
(1 + * Z 2 1 + * Z 5 3)
```

Решение 16.11.2

```
(defun insert-after-x (w x v &aux (a (car w)))
  (cond ((null w) nil)
        ((equalp x a)
         (cons a (append v (insert-after-x (cdr w) x v))))
        (t (cons a (insert-after-x (cdr w) x v)))))

> (insert-after-x '(1 2 1 5 3) 1 '(+ * z))
(1 + * Z 2 1 + * Z 5 3)
```

Решение 16.11.3

```
(defun insert-after-x (w x v &aux (a (car w)))
  (when w (cons a (if (equalp x a)
```

```
(append v (insert-after-x (cdr w) x v))
(insert-after-x (cdr w) x v))))
```

```
> (insert-after-x '(1 2 1 5 3) 1 '(+ * z))
(1 + * Z 2 1 + * Z 5 3)
```

Решение 16.11.4

```
(defun insert-after-x (w x v &aux (a (car w)))
  (when w (cons a (append (if (equalp x a) v nil)
                          (insert-after-x (cdr w) x v)))))
```

```
> (insert-after-x '(1 2 1 5 3) 1 '(+ * z))
(1 + * Z 2 1 + * Z 5 3)
```

Решение 16.11.5

```
(defun insert-after-x (w x v)
  (loop for a in w append (cons a (if (equalp x a) v nil)))))
```

```
> (insert-after-x '(1 2 1 5 3) 1 '(+ * z))
(1 + * Z 2 1 + * Z 5 3)
```

Задача 16.12 *inter>.lisp*

Летит чайка — белый воздушный парус; за чайкой по откосу мотается её тень и тоже летит, чуть-чуть отставая от поздней одинокой чайки.

Алексей Чапыгин «Лободыры»

Определить функцию, которая из двух списков чисел извлекает общие числа большие данного и помещает их в третий список.

Решение 16.12.1

```
(defun inter> (n w v)
  (remove-if-not #'(lambda (a) (> a n)) (intersection w v)))
```

```
> (inter> 3 '(1 2 3 4 5 6) '(2 3 4 5 6 7))
(4 5 6)
```

Структура 17 (функция (СПИСОК) (СПИСОК) (СПИСОК)) > (АТОМ)

Part of what makes Lisp distinctive is that it is designed to evolve. You can use Lisp to define new Lisp operators. As new abstractions become popular (object-oriented programming, for example), it always turns out to be easy to implement them in Lisp. Like DNA, such a language does not go out of style.

Paul Graham «ANSI Common Lisp»

Lisp is the most sophisticated programming language I know. It is literally decades ahead of the competition... it is not possible (as far as I know) to actually use Lisp seriously before reaching the point of no return.

Christian Lynbech

Обучение детей программированию противоречит современной теории обучения. Разве интересно составлять планы, овладевать дисциплиной в организации мышления, уделять внимание деталям и учиться быть самокритичным?

Алан Перлис

Работу программистов следует оценивать не по их изобретательности и логике, а по полноте анализа каждой ситуации.

Алан Перлис

Those who do not know Lisp are doomed to re-invent it.

Erik Naggum

Задача 17.1 *prefix-suffix.lisp*

Даны три списка, элементами которых являются атомы. Определить функцию, возвращающую `t` если первый список является префиксом второго, а третий список суффиксом второго, в противном случае `-nil`.

Решение 17.1.1

```
(defun prefix-suffix (p  
                      w
```

```

      s
      &aux
      (pn (length p))
      (wn (length w))
      (sn (length s))
      (a (mismatch p w))
      (b (mismatch (reverse s) (reverse w))))
    (and p
      w
      s
      (> wn pn)
      (> wn sn)
      (> (1+ a) pn)
      (> (1+ b) sn)))
> (prefix-suffix '(1 2) '(1 2 3 4 5) '(4 5))
T
> (prefix-suffix '(1 2 3) '(1 2 3) '(1 2 3))
NIL
> (prefix-suffix '(1 3) '(1 2 3 4 5) '(4 5))
NIL

```

Структура 18 (функция (СПИСОК) (СПИСОК) (СПИСОК)) > (СПИСОК)

Greenspun's Tenth Rule of Programming: any sufficiently complicated C or Fortran program contains an ad hoc informally-specified bug-ridden slow implementation of half of Common Lisp.

Philip Greenspun

Computer programming is like the ability or skill to see what Picasso saw from all the different angles at once. If it is an art, the crucial element of art is to look at things from an angle that produces new insight or at least has that potential.

Erik Naggum

Lisp is a language for doing what you've been told is impossible.

Kent Pitman

Задача 18.1 *replace-elms.lisp*

Даны три списка, элементами которых являются атомы. Заменить в первом списке все вхождения элементов из второго списка на элементы из третьего под соответствующими порядковыми номерами.

Решение 18.1.1

```
(defun replace-elms (v1 v2 v3)
  (mapcar #'(lambda (a) (if
                        (member a v2)
                        (nth (position a v2) v3)
                        a))
    v1))

> (replace-elms '(1 2 3 a b c) '(3 a b) '(33 aa bb))
(1 2 33 AA BB C)
```

Решение 18.1.2

```
(defun change (v2 v3 a)
  (cond ((null v2) nil)
        ((eq (car v2) a) (car v3))
        ((change (cdr v2) (cdr v3) a))))
```



```
(defun replace-elms (v1 v2 v3)
  (loop for a in v1
        if (member a v2) collect (change v2 v3 a)
        else collect a))

> (replace-elms '(1 2 3 a b c) '(3 a b) '(33 aa bb))
(1 2 33 AA BB C)
```

Решение 18.1.3

```
(defun change (v2 v3 a)
  (when v2 (if (eq (car v2) a)
               (car v3)
               (change (cdr v2) (cdr v3) a))))

(defun replace-elms (v1 v2 v3)
  (loop for a in v1
        collect (if (member a v2)
                    (change v2 v3 a)
                    a)))

> (replace-elms '(1 2 3 a b c) '(3 a b) '(33 aa bb))
(1 2 33 AA BB C)
```

Решение 18.1.4

```
(defun change (v2 v3 a)
  (when v2 (if (eq (car v2) a)
               (car v3)
               (change (cdr v2) (cdr v3) a))))

(defun replace-elms (v1 v2 v3 &aux (a (car v1)))
  (when v1 (cons (if (member a v2)
                    (change v2 v3 a)
                    a)
                (replace-elms (cdr v1) v2 v3))))

> (replace-elms '(1 2 3 a b c) '(3 a b) '(33 aa bb))
(1 2 33 AA BB C)
```

Задача 18.2 *remove-numbers.lisp*

Даны два списка и третий - контрольный. Составить программу, которая удаляет из обоих списков элементы, которые входят в контрольный список. Например, контрольный (1 5 7 8), два других (4 5 3 8) и (1 2 3 6 8), из них получить списки (4 3) и (2 3 6).

Решение 18.2.1

```
(defun clear (w z)
  (remove-if #'(lambda (a) (member a z)) w))
```

```
(defun remove-numbers (w v z)
  (values (clear w z) (clear v z)))

> (remove-numbers '(4 5 3 8) '(1 2 3 6 8) '(1 5 7 8))
(4 3)
(2 3 6)
```

Задача 18.3 unique-elms.lisp

I think code is a medium, it's more than just a job, it's new way we as a species communicate. We model the problems and phenomena of the world in code. In this sense, code is more about expression. And we'll never stop expressing ourselves in this manner. So I think we'll see a proliferation of the ways in which we use code - from outside the software vocation and into almost every field of study. I imagine in 25 years we'll be teaching Math, Chemistry, Biology, Physics, Geometry, most STEM topics, through code. You won't learn to program so you can build applications, but rather you'll learn to program so you can express ideas outside the realm of software. Code is the lingua franca of the information age and I don't see that changing anytime soon.

Avi Flombaum

Заданы три списка. Сформировать из них новый список, элементы которого это - элементы каждого исходного, не повторяющиеся в двух других.

Решение 18.3.1

```
(defun unique (v vv vvv)
  (cond ((null v) nil)
        ((or (member (car v) vv) (member (car v) vvv))
         (unique (cdr v) vv vvv))
        ((cons (car v) (unique (cdr v) vv vvv)))))

(defun unique-elms (a b c)
  (nconc (unique a b c) (unique b c a) (unique c a b)))

> (unique-elms '(1 2 3) '(3 4 5) '(5 6 7))
(1 2 4 6 7)
```

Решение 18.3.2 (muLISP)

```
(setq lst1 '(1 2 3 4))
(setq lst2 '(6 3 8))
```

```
(setq lst3 '(9 2 1))

(defun unique (v vv vvv)
  (cond ((null v) nil)
        ((or (member (car v) vv) (member (car v) vvv))
         (unique (cdr v) vv vvv))
        ((cons (car v) (unique (cdr v) vv vvv)))))

(defun unique-elms (a b c)
  (append (unique a b c) (unique b c a) (unique c a b)))

> (unique-elms lst1 lst2 lst3)
(4 6 8 9)
```

Задача 18.4 *place-cons.lisp*

Emacs is one of the only three fully dynamic systems I know of, the other two being Lisp (any Common Lisp implementation or any Lisp Machine from the past) and Smalltalk (or any derivation of Smalltalk like Squeak and Pharo). These are far superior models for interaction with the machine and programming it than a non-dynamic non-meta programmable C/assembly based OS like UNIX (or its derivatives) or Windows.

Andrea Ferro

Расставить cons в выражении, чтобы объединить три элемента (car (cdr '(y u i))) (car (cdr '(g1 g2 g3))) (car(cdr(cdr '(kk ll mm jjj)))). Нужно объединить первое выражение со вторым, второе с третьим. Написать для этого именованную функцию.

Решение 18.4.1

```
(defun place-cons (a b c)
  (cons a (cons b (cons c nil))))

> (place-cons (car(cdr '(y u i))) (car(cdr '(g1 g2 g3))) (car(cdr(cdr '(kk ll mm jjj)))))
(U G2 MM)
```

Решение 18.4.2

```
(defun place-cons (a b c)
  (cons (car (cdr a))
        (cons (car (cdr b))
              (cons (car (cdr (cdr c)))
                    nil)))))

> (place-cons '(y u i) '(g1 g2 g3) '(kk ll mm jjj))
```

```
(U G2 MM)
```

Решение 18.4.3

```
(defun place-cons (a b c)
  (cons (cadr a) (cons (cadr b) (cons (caddr c) nil))))

> (place-cons '(y u i) '(g1 g2 g3) '(kk ll mm jjj))
(U G2 MM)
```

Задача 18.5 *replace-a-b.lisp*

There are many “famous” programming languages and they're famous for many different reasons. Some are famous for historical reasons like Smalltalk, Lisp, Fortran, Cobol, C. Some are famous for their beauty and elegance like Smalltalk and Lisp. Some are famous for their wide applicability like Java and Python. Some are infamous for their flaws like JavaScript and PHP. There is no one language that is uniquely famous.

Richard Kenneth Eng

Определить функцию, заменяющую все вхождения одного списка или атома на другой список или атом.

Решение 18.5.1

```
(defun replace-a-b (w a b)
  (cond ((null w) nil)
        ((equalp (car w) a) (cons b (replace-a-b (cdr w) a b)))
        ((listp (car w)) (cons (replace-a-b (car w) a b)
                                (replace-a-b (cdr w) a b)))
        ((cons (car w) (replace-a-b (cdr w) a b)))))

> (replace-a-b '(a c ((b ((c)) d))) 'c '(1 2 3))
(A (1 2 3) ((B (((1 2 3))) D)))
> (replace-a-b '(A (1 2 3) ((B (((1 2 3))) D))) '(1 2 3) 'c)
(A C ((B ((C)) D)))
```

Решение 18.5.2

```
(defun replace-a-b (w a b)
  (when w
    (cons (cond ((equalp (car w) a) b)
          ((listp (car w)) (replace-a-b (car w) a b))
          (t (car w)))
          (replace-a-b (cdr w) a b))))

> (replace-a-b '(a c ((b ((c)) d))) 'c '(1 2 3))
```

```
(A (1 2 3) ((B (((1 2 3))) D)))
> (replace-a-b '(A (1 2 3) ((B (((1 2 3))) D))) '(1 2 3) 'c)
(A C ((B ((C)) D)))
```

Задача 18.6 reveal.lisp

I finally understood that the half page of code on the bottom of page 13 of the Lisp 1.5 manual was Lisp in itself. These were "Maxwell's Equations of Software!"

Alan Kay

Три дочери писательницы Дорис Кей - Джуди, Айрис и Линда тоже очень талантливы. Они приобрели известность в разных видах искусств - пении, балете и кино. Все они живут в разных городах, поэтому Дорис часто звонит им в Париж, Рим и Чикаго.

1. Джуди живет не в Париже, а Линда - не в Риме.
2. Парижанка не снимается в кино.
3. Та, кто живет в Риме, певица.
4. Линда равнодушна к балету.

Кто где живет и чем занимается?

Решение 18.6.1

```
(defun cartesian-product (w v)
  (mapcan (lambda (e) (mapcar (lambda (a) `(,a ,e)) w)) v))

(defun congregate (a b c)
  (mapcar #'(lambda (a) (cons (car a) (cadr a)))
    (cartesian-product a (cartesian-product b c))))

(defun rac (w a c)
  (remove-if #'(lambda (e) (and (eq (car e) a)
                                (eq (caddr e) c)))
    w))

(defun rbc (w b c)
  (remove-if #'(lambda (e) (and (eq (cadr e) b)
                                (eq (caddr e) c)))
    w))

(defun robncnbc (w b c)
  (remove-if #'(lambda (e)
    (or (and (eq (cadr e) b)
              (not (eq (caddr e) c)))
        (and (not (eq (cadr e) b))
              (eq (caddr e) c))))
    w))

(defun rab (w a b)
  (remove-if #'(lambda (e) (and (eq (car e) a)
```

```

                                (eq (cadr e) b)))
      w))

(defun check (w v)
  (cond ((null w) nil)
        ((eq (car w) (car v))
         ((check (cdr w) (cdr v)))))

(defun consistent (w &optional ac &aux (a (car w)))
  (cond ((null w) (when (cddr ac) ac))
        ((null ac) (consistent (cdr w) (cons a ac)))
        ((or (check a (car ac)) (check a (cadr ac)))
         (consistent (cdr w) ac))
        ((consistent (cdr w) (cons a ac)))))

(defun reveal (w &aux (v (consistent w)))
  (cond ((null w) nil)
        (v v)
        ((reveal (cdr w)))))

> (reveal (rab (robncnbc (rbc (rac (rac (congregate '(ju ir li) '(si
ba ci) '(ch ro pa)) 'ju 'pa) 'li 'ro) 'ci 'pa) 'si 'ro) 'li 'ba))
((IR BA PA) (JU SI RO) (LI CI CH))
; (Айрис Балет Париж) (Джуди пение Рим) (Линда кино Чикаго)

```

Решение 18.6.2

```

(defun cartesian-product (w v)
  (mapcan (lambda (e) (mapcar (lambda (a) `(,a ,e)) w)) v))

(defun congregate (a b c)
  (mapcar #'(lambda (a) (cons (car a) (cadr a)))
          (cartesian-product a (cartesian-product b c))))

(defun rac (w a c)
  (remove-if #'(lambda (e) (and (eq (car e) a)
                                (eq (caddr e) c)))
            w))

(defun rbc (w b c)
  (remove-if #'(lambda (e) (and (eq (cadr e) b)
                                (eq (caddr e) c)))
            w))

(defun robncnbc (w b c)
  (remove-if #'(lambda (e)
                  (or (and (eq (cadr e) b)
                          (not (eq (caddr e) c)))
                      (and (not (eq (cadr e) b))
                          (eq (caddr e) c))))
            w))

(defun rab (w a b)

```

```
(remove-if #'(lambda (e) (and (eq (car e) a)
                                (eq (cadr e) b)))
            w))

(defun check (w v)
  (when w (if (eq (car w) (car v)) t (check (cdr w) (cdr v)))))

(defun consistent (w &optional ac &aux (a (car w)))
  (cond ((null w) (when (caddr ac) ac))
        ((or (check a (car ac)) (check a (cadr ac)))
         (consistent (cdr w) ac))
        ((consistent (cdr w) (cons a ac)))))

(defun reveal (w &aux (v (consistent w)))
  (when w (if v v (reveal (cdr w)))))

> (reveal (rab (robcncb) (rbc (rac (rac (congregate '(ju ir li) '(si
ba ci) '(ch ro pa)) 'ju 'pa) 'li 'ro) 'ci 'pa) 'si 'ro) 'li 'ba))
;(IR BA PA) (JU SI RO) (LI CI CH))
;(Айрис балет Париж) (Джуди пение Рим) (Линда кино Чикаго)
```

```

(defun check (w v)
  (when w (if (eq (car w) (car v)) t (check (cdr w) (cdr v)))))

(defun consistent (w &optional ac &aux (a (car w)))
  (if w
    (consistent (cdr w)
      (if (or (check a (car ac)) (check a (cadr ac)))
        ac
        (cons a ac)))
    (when (cddr ac) ac)))

(defun reveal (w &aux (v (consistent w)))
  (when w (if v v (reveal (cdr w)))))

> (reveal (rab (robncnbc (rbc (rac (rac (congregate '(ju ir li) '(si
ba ci) '(ch ro pa)) 'ju 'pa) 'li 'ro) 'ci 'pa) 'si 'ro) 'li 'ba))
((IR BA PA) (JU SI RO) (LI CI CH))
; (Айрис балет Париж) (Джуди пение Рим) (Линда кино Чикаго)

```

Задача 18.7 *replace-atom.lisp*

...Emacs is not a text editor. It's a dynamically programmable operating system sporting a programmable programming language on top of which several programs (which in Emacs parlance are known as modes), most of which are text editors, have been developed. And this is what makes it completely different than statically programmable systems like Windows/Unix that are based on non dynamic non programmable programming languages.

Andrea Ferro

Определить функцию, осуществляющую замену элементов списка *v* на соответствующие элементы списка *z* в списке *w*.

Решение 18.7.1

```

(defun replace-atom (w v z &aux (a (car w)) (d (cdr w)))
  (cond ((null w) nil)
        ((atom a) (cons (if (member a v)
                              (nth (position a v) z)
                              a)
                          (replace-atom d v z)))
        ((cons (replace-atom a v z) (replace-atom d v z)))))

> (replace-atom '((a b) a (c (a (a d)))) '(a b) '(3 8))
((3 8) 3 (C (3 (3 D))))

```

Решение 18.7.2


```
(defun replace-atom (w v z &aux (a (car w)))
  (when w (cons (if (atom a)
                    (if (member a v)
                        (nth (position a v) z)
                        a)
                  (replace-atom a v z))
                (replace-atom (cdr w) v z))))

> (replace-atom '((a b) a (c (a (a d)))) '(a b) '(3 8))
((3 8) 3 (C (3 (3 D))))
```

Решение 18.7.3

```
(defun replace-atom (w v z)
  (loop for a in w
        collect (if (atom a)
                    (if (member a v)
                        (nth (position a v) z)
                        a)
                    (replace-atom a v z))))

> (replace-atom '((a b) a (c (a (a d)))) '(a b) '(3 8))
((3 8) 3 (C (3 (3 D))))
```

Решение 18.7.4 * XLisp+ 2.1d

```
(defun replace-atom (w v z)
  (cond ((null w) nil)
        ((atom (car w)) (cons (if (member (car w) v)
                                   (nth (our-position (car w) v 0) z)
                                   (car w))
                              (replace-atom (cdr w) v z)))
        (t (cons (replace-atom (car w) v z)
                  (replace-atom (cdr w) v z)))))

(defun our-position (a w n)
  (cond ((null w) nil)
        ((eq (car w) a) n)
        (t (our-position a (cdr w) (+ n 1)))))

> (replace-atom '((a b) a (c (a (a d)))) '(a b) '(3 8))
((3 8) 3 (C (3 (3 D))))
```

Задача 18.8 *exclusive.lisp*

Определить функцию, формирующую из 3-х списков новый, изъяв элементы, которые совпадают хотя бы в 2-х списках.

Решение 18.8.1

```
(defun exclusive (a b c)
  (append (unique a (append b c))
```

```

(unique b (append a c))
(unique c (append a b)))

(defun unique (w v &optional ac &aux (a (car w)))
  (cond ((null w) (nreverse ac))
        ((or (member a v) (member a ac)) (unique (cdr w) v ac))
        ((unique (cdr w) v (cons a ac)))))

> (exclusive '(a b c d e) '(c d e f g) '(e f g h i))
(A B H I)

```

Решение 18.8.2

```

(defun exclusive (a b c)
  (nconc (unique a (append b c))
         (unique b (append a c))
         (unique c (append a b))))

(defun unique (w v)
  (loop for a in (remove-duplicates w)
        unless (member a v) collect a))

> (exclusive '(a b c d e) '(c d e f g) '(e f g h i))
(A B H I)

```

Задача 18.9 clear.lisp

Scala, Clojure, and Haskell have made some headway but they are nowhere close to being mainstream languages. According to Indeed.com, there are only about 6,000 job postings for Scala, 500 for Clojure, and 400 for Haskell. And 300 for Elixir. There are 67,000 postings for Java, 55,000 for Python, 19,000 for Ruby, and 15,000 for Perl. OOP is laughing at FP.

Richard Kenneth Eng

Даны несколько списков. Определить функцию, возвращающую первый список таким, чтобы он не содержал элементы остальных списков.

Решение 18.9.1

```

(defun clear (w &rest v)
  (loop for a in w
        unless (member a (loop for b in v append b))
        collect a))

> (clear '(1 2 3 4 5 6 7) '(1 3) '(5 7))
(2 4 6)

```

Решение 18.9.2

```
(defun clear (w &rest v)
  (loop for a in w
        unless (some #'(lambda (z) (member a z)) v)
        collect a))
```

```
> (clear '(1 2 3 4 5 6 7) '(1 3) '(5 7))
(2 4 6)
```

Решение 18.9.3

```
(defun clear (w &rest v)
  (set-difference w (loop for b in v append b)))
```

```
> (clear '(1 2 3 4 5 6 7) '(1 3) '(5 7))
(2 4 6)
```

Решение 18.9.4

```
(defun clear (w &rest v) (set-difference w (reduce #'append v)))
```

```
> (clear '(1 2 3 4 5 6 7) '(1 3) '(5 7))
(2 4 6)
```

Решение 18.9.5

```
(defun clear (w &rest v)
  (labels ((-clear (w v &aux (a (car w)))
            (cond ((null w) nil)
                  ((member a v) (-clear (cdr w) v))
                  ((cons a (-clear (cdr w) v))))))
    (-clear w (reduce #'append v))))
```

```
> (clear '(1 2 3 4 5 6 7) '(1 3) '(5 7))
(2 4 6)
```

Решение 18.9.6

```
(defun clear (w &rest v)
  (labels ((-clear (w v &aux (a (car w)))
            (when w (if (member a v) (-clear (cdr w) v)
                        (cons a (-clear (cdr w) v))))))
    (-clear w (reduce #'append v))))
```

```
> (clear '(1 2 3 4 5 6 7) '(1 3) '(5 7))
(2 4 6)
```

Решение 18.9.7

```
(defun -clear (w v &aux (a (car w)))
  (when w (if (member a v) (-clear (cdr w) v)
              (cons a (-clear (cdr w) v)))))
```

```
(defun clear (w &rest v) (-clear w (reduce #'append v)))

> (clear '(1 2 3 4 5 6 7) '(1 3) '(5 7))
(2 4 6)
```

Решение 18.9.8

```
(defun clear (v &rest w) (d v w))

(defun -clear (v w)
  (if w (-clear (set-difference v (car w)) (cdr w))
    v))

> (clear '(1 2 3 4 5 6 7) '(1 3) '(5 7))
(2 4 6)
```

Решение 18.9.9

```
(defun clear (v &rest w) (d v w))

(defun -clear (v w)
  (loop for a in w do (setf v (set-difference v a))) v)

> (clear '(1 2 3 4 5 6 7) '(1 3) '(5 7))
(2 4 6)
```

Задача 18.10 1-3-6.lisp

If you compare the design of Unix to that of a Lisp machine, the Lisp machine probably had a more elaborate and well thought-out design, but following the “worse is better” philosophy, our smartphones and datacenters today ended up running a descendent of Unix, not a descendent of Lisp machines.

Christophe De Dinechin

Даны три списка. Определить функцию, возвращающую список из элементов с номерами: 1, 3 и 6.

Решение 18.10.1

```
(defun 1-3-6 (a b c)
  (list (first a) (third b) (sixth c)))

> (1-3-6 '((pi) v (h j k)) '(r yu (h kj kl)) '(u ii oo ll pp (3 4 5)))
((PI) (H KJ KL) (3 4 5))
```

Решение 18.10.2

```
(defun 1-3-6 (a b c)
  (list (car a) (caddr b) (cadr (cddddr c))))

> (1-3-6 '((pi) v (h j k)) '(r yu (h kj kl)) '(u ii oo ll pp (3 4 5)))
((PI) (H KJ KL) (3 4 5))
```

Решение 18.10.3

```
(defun 1-3-6 (a b c)
  `(,(car a) ,(caddr b) ,(cadr (cddddr c))))

> (1-3-6 '((pi) v (h j k)) '(r yu (h kj kl)) '(u ii oo ll pp (3 4 5)))
((PI) (H KJ KL) (3 4 5))
```

Решение 18.10.4

```
(defun 1-3-6 (a b c)
  `(,(first a) ,(third b) ,(sixth c)))

> (1-3-6 '((pi) v (h j k)) '(r yu (h kj kl)) '(u ii oo ll pp (3 4 5)))
((PI) (H KJ KL) (3 4 5))
```

Структура 19 (функция (СПИСОК) АТОМ АТОМ АТОМ) > АТОМ

A human being should be able to change a diaper, plan an invasion, butcher a hog, conn a ship, design a building, write a sonnet, balance accounts, build a wall, set a bone, comfort the dying, take orders, give orders, cooperate, act alone, solve equations, analyze a new problem, pitch manure, program a computer, cook a tasty meal, fight efficiently, die gallantly. Specialization is for insects.

Robert A. Heinlein

Cleopatra lived closer in time to the moon landing than the construction of the Great Pyramids.

~

Задача 19.1 *cocoa-bag-capacity.lisp*

Фермер собрал 7 т какао бобов. Из них 1 т он продал на шоколадную фабрику, а остальные поместил в 200 мешков поровну. Сколько килограммов какао бобов в каждом мешке?

Решение 19.1.1

```
(defun cocoa-bag-capacity (harvest factory number-of-bags)
  (/ (* (- harvest factory) 1000) number-of-bags))

> (cocoa-bag-capacity 7 1 200)
30
```

Задача 19.2 *left-rectangle.lisp*

Написать функцию, возвращающую определенный интеграл методом левых прямоугольников.

Решение 19.2.1

```
(defun left-rectangle (f a b n &aux (d (/ (- b a) n)))
  (* d (loop for x from a below b by d sum (funcall f x))))

> (float (left-rectangle (lambda (x) x) 0 5 100))
12.375
> (float (left-rectangle (lambda (x) (* x x)) 0 5 100))
```

```
41.04375
> (left-rectangle (lambda (x) (/ 1 (log x))) 2 5 5)
2.8637977
```

Задача 19.3 right-rectangle.lisp

Написать функцию, возвращающую определенный интеграл методом правых прямоугольников.

Решение 19.3.1

```
(defun right-rectangle (f a b n &aux (d (/ (- b a) n)))
  (* d (loop for x from b above a by d sum (funcall f x))))

> (float (right-rectangle (lambda (x) x) 0 5 100))
12.625
> (float (right-rectangle (lambda (x) (* x x)) 0 5 100))
42.29375
> (right-rectangle (lambda (x) (/ 1 (log x))) 2 5 5)
2.3709817
```

Задача 19.4 midpoint-rectangle.lisp

Написать функцию, возвращающую определенный интеграл методом средних прямоугольников.

Решение 19.4.1

```
(defun midpoint-rectangle (f a b n &aux (d (/ (- b a) n)))
  (* d (loop for x from (+ a (/ d 2)) below b by d
    sum (funcall f x))))

> (float (midpoint-rectangle (lambda (x) x) 0 5 100))
12.5
> (float (midpoint-rectangle (lambda (x) (* x x)) 0 5 100))
41.665627
> (midpoint-rectangle (lambda (x) (/ 1 (log x))) 2 5 5)
2.5757823
```

Задача 19.5 trapezium.lisp

Написать функцию, возвращающую определенный интеграл методом трапеции.

Решение 19.5.1

```
(defun trapezium (f a b n &aux (d (/ (- b a) n)))
  (* 0.5 d (+ (funcall f a)
    (* 2 (loop for x from (+ a d) below b by d
      sum (funcall f x)))
    (funcall f b))))
```

```
> (trapezium (lambda (x) x) 0 5 100)
12.5
> (trapezium (lambda (x) (* x x)) 0 5 100)
41.66875
> (trapezium (lambda (x) (/ 1 (log x))) 2 5 5)
2.6173894
```

Задача 19.6 *simpson.lisp*

...Emacs is not a text editor. It's a dynamically programmable operating system sporting a programmable programming language on top of which several programs (which in Emacs parlance are known as modes), most of which are text editors, have been developed. And this is what makes it completely different than statically programmable systems like Windows/Unix that are based on non dynamic non programmable programming languages.

Andrea Ferro

Написать функцию, возвращающую определенный интеграл методом Симпсона.

Решение 19.6.1

```
(defun simpson (f a b n
  &aux
    (h (/ (- b a) n))
    (s1 (funcall f (+ a (/ h 2))))
    (s2 0))
  (loop for i from 1 below n do
    (let ((z (+ a (* h i))))
      (incf s1 (funcall f (+ z (/ h 2))))
      (incf s2 (funcall f z)))
    finally (return (* (/ h 6) (+ (funcall f a)
      (funcall f b)
      (* s1 4)
      (* s2 2))))))

> (float (simpson (lambda (x) x) 0 5 100))
12.5
> (float (simpson (lambda (x) (* x x)) 0 5 100))
41.666668
> (simpson (lambda (x) (/ 1 (log x))) 2 5 5)
2.589651
```


Структура 20 (функция (СПИСОК АТОМ АТОМ АТОМ) > (СПИСОК)

Unformed people delight in the gaudy and in novelty. Cooked people delight in the ordinary.

Erik Naggum

Mammoths still walked the earth when the Great Pyramids were being built.

~

Задача 20.1 over-conditional-range.lisp

Определить функцию, которая возвращает список элементов списка *w*, расположенных между *n*-ным и *m*-ным элементами включительно и удовлетворяющих некоторому предикату *p*.

Решение 20.1.1

```
(defun conditional-range (w n m p
                        &optional (z 0)
                        &aux (a (car w)) (d (cdr w)))
  (cond ((or (null w) (> z m)) nil)
        ((and (<= n z m) (funcall p a))
         (cons a (conditional-range d n m p (1+ z))))
        ((conditional-range d n m p (1+ z)))))

(defun over-conditional-range (w n m p)
  (if (<= n m)
      (conditional-range w n m p)
      (conditional-range w m n p)))

> (over-conditional-range '(1 2 3 4 5 6 7 12 33 45 57) 4 7 'oddp)
(5 7)
```

Решение 20.1.2

```
(defun conditional-range (w n m p
                        &optional ac (z 0)
                        &aux (a (car w)) (d (cdr w)))
  (cond ((or (null w) (> z m)) (nreverse ac))
        (t (conditional-range d n m p
                               (if (and (<= n z m) (funcall p a))
                                   (cons a ac)
                                   ac)
                               (1+ z)))))
```

```
(defun over-conditional-range (w n m p)
  (if (<= n m)
      (conditional-range w n m p)
      (conditional-range w m n p)))

> (over-conditional-range '(1 2 3 4 5 6 7 12 33 45 57) 4 7 'oddp)
(5 7)
```

Решение 20.1.3

```
(defun conditional-range (w n m p
  &optional ac (z 0)
  &aux (a (car w)) (d (cdr w)))
  (if (or (null w) (> z m))
      (nreverse ac)
      (conditional-range d n m p
        (if (and (<= n z m) (funcall p a))
            (cons a ac)
            ac)
        (1+ z))))

(defun over-conditional-range (w n m p)
  (if (<= n m)
      (conditional-range w n m p)
      (conditional-range w m n p)))

> (over-conditional-range '(1 2 3 4 5 6 7 12 33 45 57) 4 7 'oddp)
(5 7)
```

Решение 20.1.4

```
(defun conditional-range (w n m p)
  (loop for a in w for b from 1 to m
        when (and (>= b n) (funcall p a))
        collect a))

(defun over-conditional-range (w n m p)
  (if (<= n m)
      (conditional-range w n m p)
      (conditional-range w m n p)))

> (over-conditional-range '(1 2 3 4 5 6 7 12 33 45 57) 4 7 'oddp)
(5 7)
```

Задача 20.2 sub-matrix-ranges.lisp

Дана матрица. Определить функцию, которая возвращает подматрицу по заданным диапазонам.

Решение 20.2.1

```
(defun sub-matrix-ranges (w n m j k)
```

```

(loop for a in w for b from 0
      when (<= n b m)
      collect (loop for d in a for e from 0
                    when (<= j e k) collect d)))

> (sub-matrix-ranges '((0 1 2) (3 4 5) (6 7 8)) 0 0 1 2)
((1 2))

```

Задача 20.3 sub-elms-ranges.lisp

Дана матрица. Определить функцию, которая возвращает список элементов по заданным диапазонам.

Решение 20.3.1

```

(defun sub-elms-ranges (w n m j k)
  (loop for a in w for b from 0
        when (<= n b m)
        nconc (loop for d in a for e from 0
                    when (<= j e k) collect d)))

> (sub-elms-ranges '((0 1 2) (3 4 5) (6 7 8)) 0 0 1 2)
(1 2)

```

Задача 20.4 sub-matrix-coordinates.lisp

Дана матрица. Определить функцию, которая возвращает подматрицу по заданным координатам.

Решение 20.4.1

```

(defun sub-matrix-coordinates (w n j m k)
  (loop for a in w for b from 0
        when (<= n b m)
        collect (loop for d in a for e from 0
                      when (<= j e k) collect d)))

> (sub-matrix-coordinates '((0 1 2) (3 4 5) (6 7 8)) 0 0 1 2)
((0 1 2) (3 4 5))

```

Задача 20.5 sub-elms-coordinates.lisp

Дана матрица. Определить функцию, которая возвращает список элементов по заданным координатам.

Решение 20.5.1

```

(defun sub-elms-coordinates (w n j m k)
  (loop for a in w for b from 0
        when (<= n b m)
        nconc (loop for d in a for e from 0

```

```

when (<= j e k) collect d)))

> (sub-elms-coordinates '((0 1 2) (3 4 5) (6 7 8)) 0 0 1 2)
(0 1 2 3 4 5)

```

Задача 20.6 *ects-grades.lisp*

What we have to learn to do we learn by doing.

Aristotle «Ethica Nicomahea II»

Определить функцию для подсчета ECTS (англ. European Credit Transfer and Accumulation System — Европейская система перевода и накопления кредитов).

Решение 20.6.1

```

(defun ects-grades (ws hw t1 t2 e1 e2)
  (grades (+ (workshop (average ws)) hw t1 t2 e1 e2)))

(defun grades (r)
  (cond ((<= 90 r 100) (list r 'a))
        ((<= 82 r 89) (list r 'b))
        ((<= 74 r 81) (list r 'c))
        ((<= 64 r 73) (list r 'd))
        ((<= 60 r 63) (list r 'e))
        ((<= 35 r 59) (list r 'fx))
        ((<= 1 r 34) (list r 'f))))

(defun workshop (n)
  (cond ((<= 4.6 n 5.0) 20)
        ((<= 4.1 n 4.6) 16)
        ((<= 3.6 n 4.1) 12)
        ((<= 3.1 n 3.6) 8)
        ((<= 2.6 n 3.1) 4)
        ((<= 1.1 n 2.6) 2)
        ((<= 0 n 1.1) 0)))

(defun average (w)
  (float (/ (reduce #'+ w) (length w))))

> (ects-grades '(0 3 4 4 5 0 4 4 5 0 3 0 5 4 4 4 5) 10 15 15 20 20)
(88 B)

```

Структура 21 (функция (СПИСОК) (СПИСОК) АТОМ АТОМ) > (СПИСОК)

Emacs is written in Lisp, which is the only computer language that is beautiful.

Neal Stephenson «In the beginning... was the command line»

No existing programming language will still be in active use in 100 years, except perhaps LISP :-)

Dan Shappir

The language God would have used to implement the Universe.

Svein Ove Aas

Задача 21.1 *balance.lisp*

Даны два числовых списка w и v , k и n – целые числа ($k, n = 1, 2, 3$). Получить разность между k -м элементом списка w и n -м элементом списка v .

Решение 21.1.1

```
(defun n-element (n w)
  (cond ((null w) nil)
        ((zerop n) (car w))
        ((n-element (- n 1) (cdr w)))))

(defun balance (w v k n) (- (n-element k w) (n-element n v)))

> (balance '(2 4 6) '(1 3 5) 0 0)
1
```

Задача 21.2 *run-over.lisp*

Даны два числовых одноуровневых списка $x=(x_1, x_2, \dots, x_n)$ и $y=(y_1, y_2, \dots, y_n)$ одинаковой длины, g – некоторая функция от двух переменных, определенная в области чисел (умножение, сложение, возведение в степень и т.д.). Например, $g(x, y)=x*y$, где x и y – числа. Определить функционал $(f\ g\ a\ x\ y)$, где g – функциональный аргумент, a – заданное число, а x и y – числовые списки, выдающий в качестве значения новый список, образованный из чисел $g(x_i, y_i)$, не превосходящих заданное a (x_i, y_i – элементы исходных списков).

Решение 21.2.1

```
(defun drop>n (w n)
  (cond ((null w) nil)
        ((< (car w) n) (cons (car w) (drop>n (cdr w) n)))
        ((drop>n (cdr w) n))))

(defun run-over (g a x y)
  (drop>n (mapcar g x y) a))

> (run-over #'* 50 '(1 2 3) '(11 22 33))
(11 44)
```

Решение 21.2.2

```
(defun drop>n (w n)
  (cond ((null w) nil)
        ((< (car w) n) (cons (car w) (drop>n (cdr w) n)))
        ((drop>n (cdr w) n))))

(defun run-over (g a x y) (drop>n (mapcar (intern g) x y) a))

> (run-over "*" 50 '(1 2 3) '(11 22 33))
(11 44)
```

Решение 21.2.3

```
(defun drop>n (w n)
  (when w (if (< (car w) n)
              (cons (car w) (drop>n (cdr w) n))
              (drop>n (cdr w) n))))

(defun run-over (g a x y) (drop>n (mapcar (intern g) x y) a))

> (run-over "*" 50 '(1 2 3) '(11 22 33))
(11 44)
```

Решение 21.2.4

```
(defun run-over (g a x y)
  (remove-if #'(lambda (e) (> e a)) (mapcar g x y)))

> (run-over #'* 50 '(1 2 3) '(11 22 33))
(11 44)
```

Задача 21.3 divided.lisp

Заданы списки целых чисел, вернуть список чисел, которые делятся на два заданных числа.

Решение 21.3.1

```
(defun divided (n m &rest w)
```

```
(remove-if-not
 #'(lambda (a)
      (and (zerop (mod a n))
            (zerop (mod a m)))))
(reduce #'nconc w))

> (divided 2 3 '(6 12) '(6 7) '(11 12))
(6 12 6 12)
```

Задача 21.4 *divided-infinity.lisp*

Дан список делителей и произвольное количество числовых списков. Вернуть список чисел, которые делятся без остатка на каждый из делителей.

Решение 21.4.1

```
(defun divided-infinity (v &rest w)
  (remove-if-not
   #'(lambda (a)
        (every #'(lambda (b) (zerop (mod a b))) v))
   (reduce #'nconc w)))

> (divided-infinity '(2 3 5) '(30 40) '(300 400))
(30 300)
```

Задача 21.5 *xinsert-elms.lisp*

Получить "простой" список включением элементов списка *v* между *n*-ым и *n+1*-ым элементами списка *w*, если некоторый объект *a* совпадает с *k*-ым элементом списка *v*.

Решение 21.5.1

```
(defun xinsert-elms (w v n a k)
  (if (equalp (nth k v) a)
      (append (subseq w 0 n)
                (cons (nth n w) v)
                (nthcdr (1+ n) w))
      w))

> (xinsert-elms '(1 2 3 4 5) '(a b) 2 'b 1)
(1 2 3 A B 4 5)
```

Послесловие

Given current trends in scientific progress, someday it may be possible for beings in the future to send a message to the past. Similarly, if advanced beings exist in parallel universes, then perhaps they are already trying to send us a message. In either case, their technologies should be sufficiently advanced for them to insert a message inside the box...

We await the appearance of a message.

Clifford A. Pickover
«Sex, Drugs, Einstein, & Elves»

```
(cons 'лисп (cons 'умеет (cons 'ждать (cons '|| nil))))
```

Well, you did ask for a sign.

God

Литература

Абельсон Х., Ледин К., Льюис Г. Атака битов: твоя жизнь, свобода и благополучие в цифровую эпоху. – СПб: Символ- Плюс, 2009. – 392 с.

Абельсон Х., Сассман Д. Структура и интерпретация компьютерных программ. Пер. с англ. – М.: Добросвет, 2004. – 576 с.

Бенерт К. Н., Герасин М. Л. Введение в Лисп: Учебное пособие по спецкурсу. – Сыктывкар: Сыктывкарский государственный университет, 1994. – 131 с.

Большакова Е. И., Груздева Н. В. Основы программирования на языке Лисп: Учебное пособие. – М.: Издательский отдел факультета ВМК МГУ имени М.В.Ломоносова; МАКС Пресс, 2010 – 112 с.

Большая поваренная книга программиста: Common Lisp. – Режим доступа:
<http://najomi.org/common-lisp>

Бондарев С. В. Программирование на языке Лисп для систем искусственного интеллекта: Методические указания. – Севастополь: Изд-во СевНТУ, 2013. – 68 с.

Бушенко Д. Изучаем Emacs. Эпизоды 01-13. – Режим доступа:
<https://www.youtube.com/watch?v=1Asfqt13G-0>

Водолазкий В. Первые шаги в GNU Common Lisp. – Режим доступа:
<http://www.cardarmy.ru/proekt/lisp/art1.htm>

Городня Л. В. Основы функционального программирования. Курс лекций. Учебное пособие. М: ИНТУИТ, 2004. – 280 с.

Грэм П. ANSI Common Lisp. – Пер. с англ. – СПб.: Символ-Плюс, 2012. – 448 с.

Могилев А. В., Пак Н. И., Хённер Е. К. Информатика: учеб. пособие для студ. пед. вузов. – М.: Издательский центр «Академия», 2012. – 848 с.

Неспейвода Н. Н., Скопин И. Н. Основания программирования. – 2002. – 913 с.

Окасаки К. Чисто функциональные структуры данных. – Пер. с англ. – М.: Д М К Пресс, 2016. – 252 с.

Семенов М. Ю. Язык Лисп для персональных ЭВМ. – М.: Изд-во Моск. ун-та, 1989. – 74 с.

Хювёнен Э., Сеппянен И. Мир Лиспа. В 2-х т. Т.1: Введение в язык Лисп и функциональное программирование. . Пер. с финск.– М.: Мир, 1990. – 447 с.

Хювёнен Э., Сеппянен И. Мир Лиспа. В 2-х т. Т.2: Методы и системы программирования. Пер. с финск.– М.: Мир, 1990. – 319 с.

Шень А. Программирование: Теоремы и задачи. – М.: МЦНМО, 2007. – 296 с.

Abelson H., Ledeen K., Lewis H. Blown to Bits: Your Life, Liberty, and Happiness After the Digital Explosion. – Upper Saddle River, NJ: Addison-Wesley Professional. – 384 p.

Allen C., Dhagat M. LISP Primer. – 1992. – 134 p. – Mode of access:
<http://www.svbug.com/documentation/lisp/>

Articulate Common Lisp: How to write Common Lisp in 2017 - an initiation manual for the uninitiated. – Mode of access: <http://articulate-lisp.com>

Barski C. Land of Lisp: Learn to Program in Lisp, One Game at a Time! – San Francisco, CA: No Starch Press, – 2011. – 504 p.

Burgemeister B. Common Lisp Quick Reference. – 2014. – 52 p. – Mode of access:
<http://clqr.boundp.org/clqr-a4-booklet-all.pdf>

Category: Common Lisp. – Mode of access:
http://rosettacode.org/wiki/Category:Common_Lisp

Cooper D. J. Basic Lisp Techniques. – 2003. – 92 p. – Mode of access:
http://www.franz.com/resources/educational_resources/cooper.book.pdf

Common Lisp HyperSpec. – Mode of access:
ftp://ftp.lispworks.com/pub/software_tools/reference/HyperSpec-7-0.tar.gz

Costanza P. Highly Opinionated Guide to Lisp – 2013. – Mode of access: <http://www.p-cos.net/lisp/guide.html>

Debugging Lisp Part 1: Recompilation. – Mode of access: <http://malisper.me/debugging-lisp-part-1-recompilation/>

Debugging Lisp Part 2: Inspecting. – Mode of access: <http://malisper.me/debugging-lisp-part-2-inspecting/>

Debugging Lisp Part 3: Redefining Classes. – Mode of access: <http://malisper.me/debugging-lisp-part-3-redefining-classes/>

Debugging Lisp Part 4: Restarts. – Mode of access: <http://malisper.me/debugging-lisp-part-4-restarts/>

Debugging Lisp Part 5: Miscellaneous. – Mode of access: <http://malisper.me/debugging-lisp-part-5-miscellaneous/>

Deployment tools for standalone Common Lisp applications. – Mode of access: <https://shinmera.github.io/deploy/>

Draft Proposed American National Standard for Information Systems – Programming Language – Common Lisp X3J13/94-101R. – Mode of access: http://cvberry.com/downloads/ansi_cl_standard_draft_nice.pdf

EECS 325 Readings. – Mode of access: <http://www.cs.northwestern.edu/academics/courses/325/readings/readings.php>

Forbus K. D., De Kleer J. Building Problem Solvers (Artificial Intelligence). Cambridge, MA: A Bradford Book The MIT Press, 1993. – 716 p.

Gat E. The Complete Idiot’s Guide to Common Lisp Packages. - 14 p. – Mode of access: <http://www.flownet.com/gat/packages.pdf>

Getting Started: Guidelines, hints and resources for those who are starting out with Lisp. – Mode of access: <https://cliki.net/Getting+Started>

Gordon G. J. Common LISP Hints. – Mode of access: <http://www.n-a-n-o.com/lisp/cmuccl-tutorials/LISP-tutorial.html>

Graham P. Hackers & Painters: Big Ideas from the Computer Age. – Beijing, Cambridge, Farnham, Koln, Paris, Sebastopol, Taipei, Tokyo: O'Reilly Media, 2004. – 272 p.

Graham P. On Lisp. – Upper Saddle River, NJ: Prentice Hall, 1993. – 413 p.

Hahn H. Harley Hahn's Emacs Field Guide. – New York, NY: Apress, 2016. – 276 p.

Herda M. Common Lisp UltraSpec. – Mode of access: <http://phoe.tymoon.eu/clus/doku.php>

Hoyte D. Let Over Lambda. – Lulu, 2010. – 384 p.

Knott G. D. Interpreting LISP. – 1997. – 103 p. – Mode of access: <http://www.civilized.com/files/lispbook.pdf>

Lamkins D. B. Successful Lisp: How to Understand and Use Common Lisp. – BookFix – 2004. – 360 p. – Mode of access: <http://psg.com/~dlamkins/sl/contents.html>

Learn X in Y minutes: Where X=Common Lisp. – Mode of access: <https://learnxinyminutes.com/docs/common-lisp/>

Learning GNU Emacs / D. Cameron, J. Elliott, M. Loy, E. Raymond, B. Rosenblatt. - Sebastopol, CA: O'Reilly Media, 2004. – 536 p.

Levine N. Lisp and elements of style. – Mode of access: <https://www.nicklevine.org/declarative/lectures/>

Lisp: Уроки 1-10. – Режим доступа: <http://cybern.ru/category/other-2/lisp>

Lisp: Common Lisp, Racket, Clojure, Emacs Lisp. – Mode of access: <http://hyperpolyglot.org/lisp>

Lisp-fan: Блог для фанатов лиспа. – Режим доступа: <https://www.lisp-fan.ru>

Lisp Hackers: Interviews with 100x More Productive Programmers / Edited by V. Dyomkin. – Leanpub, 2013. – 77 p. – Mode of access: <https://leanpub.com/lisphackers/read>

Lisp journey. – Mode of access: <https://lisp-journey.gitlab.io/about/>

Luger G. F. Artificial Intelligence: Structures and Strategies for Complex Problem Solving. – Boston, MA: Pearson Education, 2011. – 754 p.

Luger G. F., Stubblefield W. A. AI Algorithms, Data Structures, and Idioms in Prolog, Lisp, and Java. – Boston, MA: Pearson Education, 2008. – 447 p.

McCarthy J. LISP 1.5 Programmer's Manual. – Cambridge, MA: The M.I.T. Press, 1962. – 112 p. – Mode of access: <http://www.softwarepreservation.org/projects/LISP/book/>

Michalewicz Z., Fogel D. B. How to solve it: Modern Heuristics. – Berlin: Springer, 2004. – 554 p.

Naming conventions: Some symbol naming conventions, distilled from CLHS, the Lisp FAQ, and comp.lang.lisp. – Mode of access: <https://cliki.net/naming%20conventions>

Norvig P., Pitman K. Tutorial on Good Lisp Programming Style. 1993. – 116 p. – Mode of access: <https://www.cs.umd.edu/~nau/cmsc421/norvig-lisp-style.pdf>

Norvig P. Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp. – San Francisco, CA: Morgan Kaufmann, 1992. – 946 p.

Novak G. S. Lisp Programming: Lecture Notes. –1985. – 130 p. – Mode of access: <ftp://www.cs.utexas.edu/pub/AI-Lab/tech-reports/UT-AI-TR-85-6.pdf>

Penev P. Lisp Web Tales: My attempts at learning web development, foolishly using Common Lisp, and even more foolishly, writing about it in public. – Leanpub, 2013. – 91 p. – Mode of access: <https://leanpub.com/lispwebtales>

Petersen M. Mastering Emacs. – 2015. – 280 p. – Mode of access: <https://www.masteringemacs.org>

Pierce D. R., Shapiro S. C. Common Lisp: A Brief Tutorial. – 2004 – 32 p. – Mode of access: <http://curry.ateneo.net/~jpv/cs171/LispTutorial.pdf>

Polya G. How to solve it. – Princeton and Oxford: Princeton University Press, 2004. – 253 p.

Queinnee Ch. Lisp in Small Pieces. – Cambridge: Cambridge University Press, 1996. – 534 p.

Russell S. J., Norvig P. Artificial Intelligence A Modern Approach. – Englewood Cliffs, NJ: Prentice Hall, 1995. – 932 p.

Seibel P. Practical Common Lisp. – New York, NY: Apress, 2005. – 499 p.

Shapiro S. C. Common Lisp: An Interactive Approach. – New York, NY: Computer Science Press, 1992. – 322 p.

SLIME User Manual, version 2.14. – Mode of access: <https://common-lisp.net/project/slime/doc/html/>

Solutions to Ninety-Nine Lisp Problems. – Mode of access: <https://informatimago.com/develop/lisp/l99/index.html>

Steele G. L. Common Lisp the Language, 2nd edition. – Digital Press, 1990. – 1029 p.

Tanimoto S. L. The Elements of Artificial Intelligence. An introduction Using Lisp. – Seattle, Washington: Computer Science Press, 1987. – 529 p.

The Common Lisp Cookbook . – Mode of access: <http://cl-cookbook.sourceforge.net>

Tornhill A. Lisp for the Web. – Leanpub, 2015. – 45 p. – Mode of access: <https://leanpub.com/lispweb>

Touretzky D. S. Common Lisp: A Gentle Introduction to Symbolic Computation. – Redwood City: Carnegie Mellon University: The Benjamin/Cummings Publishing Company. – 1990. – 587 p.

Watson M. Loving Lisp, or the Savvy Programmer's Secret Weapon. – Leanpub, 2011. – 170 p. – Mode of access: <https://leanpub.com/lovinglisp/read>

Weber G. Episodic Learner Model: The Adaptive Remote Tutor. – Mode of access: <http://art2.ph-freiburg.de/Lisp-Course>

Weitz E. Common Lisp Recipes: A Problem-Solution Approach. – New York, NY: Apress, 2015. – 744 p.

Указатель функций

/

/=, 180, 249, 336, 516, 1033

1

1–, 79, 81, 84, 90, 91, 101, 105, 107, 109, 110,
111, 113, 114, 116, 121, 131, 132, 134,
135, 136, 137, 143, 144, 146, 147, 158,
165, 167, 169, 170, 175, 177, 178, 179,
182, 185, 188, 189, 190, 191, 192, 193,
194, 196, 197, 201, 205, 206, 207, 208,
219, 220, 225, 226, 249, 251, 252, 253,
254, 255, 256, 258, 259, 290, 299, 308,
311, 344, 345, 346, 409, 430, 441, 457,
458, 486, 488, 530, 548, 552, 563, 591,
592, 593, 613, 614, 615, 616, 618, 647,
661, 663, 677, 678, 680, 681, 698, 699,
703, 704, 707, 708, 709, 727, 728, 733,
734, 735, 747, 748, 749, 750, 781, 790,
791, 792, 793, 794, 795, 798, 801, 809,
812, 813, 817, 818, 826, 827, 828, 831,
833, 846, 853, 854, 856, 860, 861, 862,
872, 879, 884, 885, 891, 893, 895, 896,
898, 899, 900, 901, 902, 911, 912, 917,
922, 923, 925, 933, 934, 937, 945, 946,
947, 948, 950, 952, 955, 956, 958, 960,
961, 962, 963, 971, 972, 975, 976, 977,
981, 986, 990, 991, 992, 993, 994, 995,
998, 1001, 1002, 1003, 1004, 1008, 1028,
1040, 1047, 1074, 1075, 1092, 1093, 1096,
1099, 1102, 1111, 1116, 1119, 1128, 1129,
1130, 1131, 1136, 1139, 1140, 1144, 1151,
1159, 1160, 1165, 1166, 1167, 1168, 1169,
1170, 1171, 1172, 1173, 1192, 1193, 1203,
1207, 1208

1+, 79, 80, 89, 90, 101, 102, 103, 109, 110,
115, 121, 127, 128, 129, 135, 137, 138,
139, 162, 170, 175, 180, 182, 185, 194,
195, 204, 205, 206, 207, 208, 216, 218,

219, 220, 225, 226, 232, 249, 258, 259,
260, 262, 263, 271, 274, 277, 280, 283,
284, 285, 288, 290, 291, 292, 297, 300,
302, 305, 308, 309, 311, 316, 317, 318,
320, 322, 328, 332, 333, 335, 337, 351,
359, 361, 364, 365, 366, 373, 377, 391,
392, 396, 398, 401, 402, 409, 410, 425,
426, 430, 431, 441, 452, 455, 457, 472,
474, 475, 496, 505, 506, 508, 515, 519,
526, 533, 534, 543, 546, 547, 549, 551,
555, 558, 561, 563, 584, 585, 594, 595,
600, 609, 612, 616, 617, 627, 631, 632,
635, 647, 648, 655, 661, 663, 664, 672,
678, 689, 697, 703, 707, 708, 709, 724,
727, 728, 737, 747, 748, 749, 750, 781,
790, 791, 792, 793, 794, 795, 796, 805,
807, 811, 813, 815, 816, 817, 825, 826,
828, 834, 841, 846, 847, 848, 849, 851,
852, 858, 868, 869, 870, 872, 874, 876,
880, 884, 885, 886, 887, 888, 892, 893,
897, 903, 904, 905, 913, 914, 916, 922,
926, 927, 932, 936, 952, 954, 987, 988,
996, 997, 998, 1001, 1007, 1013, 1040,
1057, 1072, 1073, 1096, 1101, 1104, 1105,
1121, 1131, 1141, 1142, 1143, 1144, 1172,
1173, 1179, 1197, 1198, 1203

A

abs, 156, 158, 248, 303, 304, 517, 615, 671,
672, 673, 699, 827, 828, 901, 932, 978,
979, 980, 1102
adjoin, 75, 368, 473, 477, 478, 568, 676,
682, 732, 733, 940, 966, 967, 968
alpha-char-p, 97, 98, 99, 125, 145, 153,
154, 174, 184, 186, 201, 221, 222, 223,
230, 231, 245, 438, 564, 582, 583, 843,
862, 863, 969
and, 58, 59, 66, 69, 77, 83, 84, 86, 87, 88, 98,
99, 101, 108, 117, 123, 124, 126, 131, 132,
134, 135, 138, 139, 143, 144, 146, 154,
155, 156, 157, 161, 163, 164, 165, 166,
167, 174, 175, 176, 177, 180, 185, 209,
215, 216, 217, 218, 219, 220, 221, 227,

- 228, 229, 230, 233, 242, 247, 250, 260,
 - 261, 263, 264, 265, 267, 270, 271, 279,
 - 281, 285, 290, 295, 296, 306, 309, 311,
 - 312, 313, 314, 316, 321, 324, 325, 326,
 - 327, 328, 329, 330, 331, 334, 335, 336,
 - 338, 340, 341, 342, 343, 347, 350, 353,
 - 362, 364, 365, 369, 370, 371, 372, 374,
 - 378, 379, 386, 391, 393, 395, 398, 399,
 - 401, 405, 406, 407, 408, 409, 410, 411,
 - 412, 413, 416, 421, 422, 426, 428, 429,
 - 430, 432, 441, 446, 449, 454, 468, 471,
 - 481, 489, 491, 492, 499, 501, 503, 504,
 - 512, 513, 515, 522, 523, 524, 526, 528,
 - 538, 539, 540, 548, 551, 565, 570, 572,
 - 573, 576, 577, 578, 589, 595, 596, 600,
 - 615, 637, 638, 643, 644, 663, 682, 683,
 - 684, 686, 687, 688, 689, 698, 705, 710,
 - 712, 713, 725, 726, 734, 754, 755, 757,
 - 758, 759, 760, 761, 768, 769, 784, 797,
 - 798, 799, 801, 802, 804, 805, 806, 807,
 - 810, 824, 826, 830, 832, 833, 834, 836,
 - 839, 842, 845, 847, 854, 857, 858, 861,
 - 862, 869, 870, 871, 873, 875, 880, 894,
 - 902, 905, 910, 911, 913, 918, 919, 920,
 - 921, 922, 923, 924, 925, 928, 929, 933,
 - 935, 937, 948, 952, 954, 956, 959, 969,
 - 970, 971, 973, 977, 995, 996, 997, 1002,
 - 1003, 1006, 1007, 1008, 1010, 1013, 1017,
 - 1018, 1021, 1022, 1023, 1024, 1026, 1027,
 - 1033, 1035, 1046, 1047, 1056, 1060, 1061,
 - 1062, 1066, 1070, 1072, 1074, 1079, 1081,
 - 1082, 1083, 1084, 1087, 1088, 1089, 1094,
 - 1097, 1098, 1099, 1102, 1104, 1107, 1108,
 - 1110, 1111, 1114, 1115, 1118, 1119, 1121,
 - 1122, 1126, 1128, 1130, 1132, 1133, 1135,
 - 1136, 1147, 1148, 1154, 1155, 1156, 1159,
 - 1160, 1162, 1163, 1165, 1167, 1168, 1171,
 - 1172, 1174, 1175, 1176, 1179, 1182, 1183,
 - 1184, 1185, 1186, 1187, 1190, 1192, 1197,
 - 1198, 1200, 1203, 1206, 1207, 1208, 1209
- append, 75, 165, 205, 251, 252, 253, 254,
- 255, 256, 257, 268, 275, 280, 293, 311,
 - 341, 373, 394, 405, 415, 423, 424, 432,
 - 442, 445, 448, 449, 450, 451, 472, 483,
 - 484, 487, 495, 497, 498, 507, 514, 521,
 - 532, 535, 554, 558, 559, 568, 589, 594,
 - 597, 620, 632, 644, 649, 651, 653, 668,
 - 671, 679, 697, 705, 706, 707, 719, 746,
 - 747, 769, 772, 776, 779, 780, 792, 793,
 - 794, 797, 802, 808, 886, 915, 922, 935,
 - 954, 955, 957, 958, 961, 962, 983, 994,
 - 995, 1010, 1018, 1031, 1032, 1042, 1052,
 - 1058, 1063, 1069, 1070, 1073, 1079, 1080,
 - 1083, 1084, 1085, 1092, 1150, 1166, 1167,
 - 1169, 1171, 1176, 1177, 1183, 1189, 1190,
 - 1191, 1192, 1203
- apply, 68, 74, 76, 102, 109, 110, 126, 127,
- 141, 163, 171, 188, 189, 190, 279, 281,
 - 283, 287, 289, 300, 303, 304, 305, 307,
 - 310, 314, 318, 320, 332, 347, 348, 349,
 - 386, 391, 411, 414, 430, 439, 441, 442,
 - 443, 447, 453, 454, 457, 458, 469, 485,
 - 507, 508, 510, 526, 528, 537, 548, 549,
 - 550, 551, 556, 564, 566, 574, 576, 581,
 - 583, 586, 587, 588, 589, 590, 600, 602,
 - 608, 611, 612, 615, 627, 628, 629, 633,
 - 637, 638, 679, 705, 708, 741, 742, 765,
 - 785, 786, 787, 816, 842, 891, 919, 931,
 - 965, 966, 967, 981, 1004, 1014, 1020,
 - 1026, 1032, 1040, 1063, 1064, 1088, 1155
- aref, 226, 237, 238, 409, 739, 740, 741, 920,
- 931, 1001, 1088, 1159
- array-dimension, 391, 739, 741, 742,
- 919, 931, 1088
- array-dimensions, 391, 741, 742, 919,
- 931
- array-total-size, 740, 741, 920, 931,
- 1158
- assoc, 608, 807
- atom, 70, 71, 73, 74, 78, 80, 81, 204, 205,
- 206, 207, 208, 267, 268, 269, 272, 273,
 - 275, 276, 277, 280, 281, 288, 289, 292,
 - 293, 294, 295, 296, 300, 301, 308, 310,
 - 312, 313, 314, 315, 316, 317, 318, 319,
 - 321, 322, 324, 325, 333, 334, 338, 339,
 - 341, 342, 362, 368, 370, 371, 372, 373,
 - 377, 390, 393, 394, 406, 412, 419, 425,
 - 426, 433, 434, 441, 442, 447, 452, 460,
 - 464, 467, 469, 473, 474, 475, 477, 478,
 - 482, 483, 487, 493, 494, 496, 497, 499,
 - 500, 520, 532, 534, 535, 540, 546, 547,
 - 554, 555, 566, 567, 568, 575, 576, 595,
 - 596, 607, 608, 617, 619, 631, 632, 635,
 - 639, 640, 643, 645, 647, 648, 649, 650,
 - 652, 653, 654, 656, 658, 661, 663, 667,
 - 669, 670, 674, 700, 701, 712, 713, 726,
 - 727, 728, 738, 739, 740, 741, 747, 749,
 - 750, 759, 763, 772, 773, 774, 776, 777,
 - 778, 779, 783, 790, 791, 792, 793, 794,
 - 795, 796, 800, 808, 809, 886, 887, 888,
 - 889, 890, 895, 896, 903, 911, 912, 913,
 - 914, 915, 917, 919, 921, 922, 931, 941,
 - 947, 948, 950, 952, 954, 957, 961, 962,

963, 968, 971, 978, 979, 980, 984, 987,
997, 1003, 1010, 1015, 1016, 1017, 1018,
1027, 1046, 1047, 1051, 1052, 1057, 1058,
1059, 1062, 1063, 1074, 1075, 1076, 1077,
1078, 1079, 1080, 1081, 1082, 1083, 1085,
1086, 1136, 1139, 1140, 1141, 1142, 1143,
1150, 1151, 1159, 1172, 1188, 1189

B

butlast, 75, 126, 127, 134, 251, 252, 253,
350, 351, 355, 445, 451, 486, 507, 526,
547, 589, 590, 665, 680, 706, 707, 712,
719, 720, 746, 754, 755, 760, 761, 802,
803, 898, 899, 900, 901, 941, 954, 955,
957, 958, 959, 1006, 1007, 1061, 1062,
1067, 1092, 1093, 1119, 1124

C

caaar, 668

caadr, 393, 668, 669, 798

caar, 96, 97, 98, 102, 152, 153, 154, 266,
271, 298, 299, 310, 443, 454, 497, 498,
499, 502, 509, 517, 518, 530, 531, 548,
574, 580, 645, 669, 731, 776, 784, 785,
786, 787, 798, 805, 806, 884, 1141

cadar, 285, 300, 497, 498, 499, 517, 518,
530, 531, 548, 645, 679, 731, 776, 777,
778, 784, 785, 798, 805, 806, 851, 852,
1141

caddar, 74, 309, 777, 784, 785, 805, 806,
1011

cadddr, 74, 148, 149, 150, 151, 152, 244,
245, 270, 271, 298, 329, 330, 503, 518,
733, 883, 1135

caddr, 148, 149, 150, 151, 152, 162, 245,
264, 269, 270, 271, 279, 309, 329, 330,
336, 396, 422, 423, 424, 435, 443, 503,
514, 519, 574, 631, 663, 694, 776, 777,
778, 814, 860, 861, 926, 971, 1011, 1097,
1098, 1099, 1166, 1169, 1184, 1185, 1186,
1187, 1193

cadr, 74, 81, 96, 97, 98, 102, 113, 135, 148,
149, 150, 151, 152, 153, 154, 162, 164,
168, 198, 199, 203, 232, 240, 241, 245,
270, 274, 278, 279, 285, 286, 299, 300,
306, 307, 310, 320, 329, 331, 333, 336,
338, 362, 367, 375, 377, 381, 386, 387,
388, 389, 396, 410, 411, 413, 414, 416,
417, 418, 422, 423, 424, 425, 428, 430,

434, 435, 438, 443, 445, 446, 449, 451,
453, 454, 462, 463, 479, 480, 481, 492,
505, 506, 509, 513, 514, 517, 518, 519,
521, 526, 529, 530, 532, 533, 543, 548,
549, 551, 553, 556, 569, 570, 571, 574,
580, 584, 585, 591, 592, 593, 597, 598,
599, 607, 608, 610, 611, 613, 615, 619,
631, 638, 640, 641, 645, 647, 648, 655,
657, 659, 660, 663, 664, 665, 682, 687,
688, 702, 705, 706, 714, 718, 719, 720,
730, 731, 732, 739, 740, 741, 742, 744,
755, 761, 764, 767, 776, 777, 778, 794,
795, 796, 798, 814, 831, 841, 882, 883,
902, 926, 940, 959, 969, 973, 1008, 1019,
1020, 1022, 1025, 1026, 1030, 1031, 1085,
1125, 1138, 1141, 1142, 1143, 1150, 1166,
1169, 1184, 1185, 1186, 1187, 1188, 1193

car, 67, 70, 74, 78, 80, 81, 94, 95, 99, 100,
102, 111, 112, 113, 114, 116, 117, 123,
124, 126, 127, 135, 140, 141, 148, 149,
150, 151, 152, 162, 163, 165, 168, 174,
184, 198, 199, 202, 203, 204, 205, 206,
207, 208, 221, 222, 223, 224, 232, 240,
241, 244, 245, 257, 259, 262, 263, 264,
265, 266, 267, 268, 269, 270, 271, 272,
273, 274, 275, 276, 277, 278, 279, 280,
281, 282, 283, 284, 285, 286, 287, 288,
289, 290, 291, 292, 293, 295, 296, 297,
298, 300, 301, 302, 303, 304, 305, 306,
307, 308, 309, 310, 311, 312, 313, 315,
316, 317, 318, 319, 320, 321, 322, 323,
325, 326, 327, 328, 329, 330, 331, 332,
333, 335, 336, 338, 339, 340, 341, 342,
343, 347, 351, 352, 354, 357, 358, 359,
360, 361, 362, 363, 364, 365, 366, 367,
368, 369, 371, 372, 373, 375, 376, 377,
379, 380, 381, 385, 386, 387, 388, 389,
390, 391, 392, 393, 394, 396, 397, 398,
399, 401, 402, 406, 408, 410, 411, 413,
414, 415, 416, 417, 418, 419, 420, 421,
422, 423, 424, 425, 426, 427, 428, 432,
433, 434, 435, 436, 437, 438, 439, 440,
441, 442, 443, 444, 445, 446, 447, 448,
449, 450, 451, 452, 453, 454, 455, 458,
459, 460, 461, 462, 463, 464, 465, 466,
467, 468, 469, 470, 471, 472, 473, 474,
475, 476, 477, 478, 479, 480, 481, 482,
483, 484, 485, 486, 487, 488, 489, 490,
491, 492, 493, 494, 495, 496, 497, 498,
499, 501, 502, 503, 504, 505, 506, 508,
509, 510, 511, 512, 513, 514, 515, 516,
517, 518, 519, 520, 521, 522, 523, 524,

525, 526, 527, 528, 529, 531, 532, 533,
 534, 535, 536, 538, 539, 540, 541, 543,
 544, 546, 547, 548, 549, 550, 551, 553,
 554, 555, 556, 558, 559, 560, 561, 562,
 564, 566, 567, 568, 569, 570, 571, 572,
 573, 574, 575, 577, 579, 580, 581, 582,
 583, 584, 585, 586, 587, 588, 589, 590,
 591, 592, 593, 594, 595, 596, 597, 598,
 599, 600, 601, 602, 605, 606, 607, 608,
 609, 610, 611, 612, 615, 617, 618, 619,
 620, 621, 622, 623, 624, 625, 626, 629,
 630, 631, 632, 634, 635, 636, 637, 638,
 639, 640, 641, 642, 643, 644, 645, 646,
 647, 648, 649, 650, 651, 652, 653, 654,
 655, 656, 657, 659, 660, 661, 662, 663,
 664, 665, 666, 667, 668, 669, 670, 671,
 672, 674, 675, 676, 677, 679, 680, 681,
 682, 683, 684, 685, 686, 687, 688, 689,
 690, 691, 692, 695, 696, 698, 699, 700,
 701, 702, 703, 705, 706, 707, 708, 709,
 710, 711, 712, 713, 714, 716, 717, 718,
 719, 720, 721, 722, 723, 724, 725, 726,
 727, 728, 729, 730, 731, 732, 737, 738,
 739, 740, 741, 742, 743, 744, 746, 747,
 748, 749, 750, 751, 752, 753, 754, 755,
 757, 758, 759, 760, 761, 763, 764, 765,
 766, 767, 768, 769, 772, 773, 774, 775,
 776, 777, 778, 779, 780, 781, 782, 783,
 784, 786, 787, 789, 790, 791, 792, 793,
 794, 795, 796, 797, 798, 800, 802, 803,
 804, 805, 806, 807, 808, 809, 814, 815,
 831, 841, 846, 850, 851, 852, 855, 856,
 857, 858, 859, 861, 862, 864, 865, 866,
 867, 880, 881, 882, 883, 884, 885, 886,
 887, 888, 889, 890, 891, 892, 893, 895,
 896, 897, 898, 899, 900, 901, 902, 903,
 904, 910, 911, 912, 913, 914, 915, 916,
 917, 918, 919, 920, 921, 922, 923, 924,
 925, 926, 927, 928, 930, 931, 932, 933,
 934, 936, 937, 938, 939, 940, 941, 942,
 943, 944, 945, 946, 947, 948, 949, 950,
 951, 952, 953, 954, 955, 956, 957, 958,
 959, 960, 961, 962, 963, 964, 965, 966,
 967, 968, 969, 971, 972, 973, 974, 975,
 976, 977, 978, 979, 980, 982, 983, 984,
 985, 987, 988, 989, 992, 993, 994, 995,
 996, 997, 998, 999, 1000, 1001, 1002,
 1003, 1004, 1005, 1006, 1007, 1008, 1009,
 1010, 1013, 1014, 1015, 1016, 1017, 1018,
 1019, 1020, 1022, 1024, 1025, 1026, 1027,
 1028, 1029, 1030, 1031, 1032, 1033, 1034,
 1035, 1036, 1037, 1039, 1042, 1043, 1044,

1045, 1046, 1047, 1048, 1049, 1050, 1051,
 1053, 1054, 1055, 1056, 1057, 1058, 1059,
 1060, 1061, 1062, 1063, 1066, 1067, 1068,
 1069, 1070, 1071, 1072, 1073, 1074, 1075,
 1076, 1077, 1078, 1080, 1081, 1082, 1083,
 1084, 1085, 1086, 1088, 1090, 1091, 1097,
 1098, 1099, 1125, 1128, 1129, 1130, 1131,
 1134, 1135, 1136, 1137, 1138, 1139, 1140,
 1141, 1142, 1143, 1144, 1145, 1146, 1147,
 1148, 1149, 1150, 1151, 1152, 1153, 1154,
 1157, 1159, 1160, 1161, 1162, 1164, 1165,
 1166, 1167, 1168, 1169, 1170, 1171, 1172,
 1173, 1174, 1175, 1176, 1177, 1180, 1181,
 1182, 1183, 1184, 1185, 1186, 1187, 1188,
 1189, 1190, 1191, 1192, 1193, 1197, 1198,
 1201, 1202
 case, 134, 135, 227, 228, 229, 427, 430, 435,
 493, 598, 602, 628, 873, 1047, 1204
 cdaaar, 298
 cdaadr, 668
 cdaddr, 278
 cddadr, 668
 cdddr, 290, 417, 418, 422, 423, 424, 435,
 492, 550, 551, 607, 608, 694, 695
 cdr, 70, 74, 78, 79, 80, 81, 94, 95, 99, 100,
 102, 112, 113, 116, 117, 123, 124, 126,
 127, 134, 135, 141, 165, 184, 186, 196,
 197, 198, 199, 202, 203, 204, 205, 206,
 207, 208, 221, 223, 224, 251, 252, 253,
 254, 255, 256, 257, 258, 259, 262, 263,
 265, 266, 267, 268, 269, 270, 271, 272,
 273, 274, 275, 276, 277, 278, 279, 280,
 281, 282, 283, 284, 285, 286, 287, 288,
 290, 291, 292, 293, 295, 296, 297, 298,
 300, 301, 302, 303, 304, 305, 306, 307,
 308, 309, 310, 311, 312, 313, 315, 316,
 317, 318, 319, 320, 321, 322, 323, 325,
 326, 327, 328, 330, 331, 332, 333, 335,
 336, 338, 339, 341, 342, 343, 350, 351,
 352, 354, 355, 356, 357, 358, 359, 360,
 361, 362, 363, 364, 365, 366, 367, 368,
 369, 371, 372, 373, 375, 376, 377, 379,
 380, 381, 385, 386, 387, 388, 389, 390,
 391, 392, 393, 394, 396, 397, 398, 399,
 401, 402, 406, 408, 410, 411, 413, 415,
 417, 418, 420, 421, 422, 425, 426, 427,
 432, 433, 434, 436, 437, 438, 439, 440,
 441, 442, 443, 444, 445, 446, 447, 448,
 449, 450, 451, 452, 453, 454, 455, 458,
 459, 460, 461, 462, 463, 464, 465, 466,
 467, 468, 469, 470, 471, 472, 473, 474,

475, 476, 477, 478, 479, 480, 481, 482,
 483, 484, 485, 486, 487, 488, 489, 490,
 491, 492, 493, 494, 495, 496, 497, 498,
 499, 501, 502, 505, 506, 507, 508, 509,
 510, 511, 512, 513, 514, 515, 516, 517,
 518, 519, 520, 521, 522, 523, 524, 525,
 526, 527, 528, 529, 530, 531, 532, 533,
 534, 535, 536, 537, 538, 539, 540, 543,
 544, 546, 547, 548, 549, 550, 551, 554,
 555, 556, 558, 559, 560, 561, 562, 564,
 565, 566, 567, 568, 569, 570, 571, 572,
 573, 574, 575, 577, 579, 580, 582, 583,
 584, 585, 586, 587, 588, 589, 590, 591,
 592, 593, 594, 595, 596, 597, 598, 599,
 601, 602, 605, 606, 607, 608, 609, 610,
 611, 612, 617, 618, 619, 620, 621, 623,
 624, 625, 626, 627, 631, 632, 635, 636,
 637, 639, 640, 641, 642, 643, 644, 645,
 646, 647, 648, 649, 650, 651, 652, 653,
 654, 655, 656, 657, 658, 659, 660, 661,
 662, 663, 664, 665, 666, 667, 668, 669,
 670, 671, 672, 674, 675, 676, 677, 679,
 680, 681, 682, 683, 684, 687, 688, 689,
 690, 691, 692, 695, 696, 698, 699, 700,
 701, 702, 703, 706, 707, 708, 709, 710,
 711, 712, 713, 714, 716, 717, 719, 721,
 722, 723, 724, 725, 726, 727, 728, 729,
 731, 732, 737, 738, 739, 743, 744, 745,
 746, 747, 748, 749, 750, 751, 752, 753,
 754, 757, 758, 759, 761, 762, 763, 764,
 765, 766, 767, 768, 769, 772, 773, 774,
 775, 776, 777, 778, 779, 780, 781, 782,
 783, 784, 785, 786, 787, 789, 790, 791,
 792, 793, 794, 795, 796, 797, 798, 800,
 802, 803, 804, 805, 806, 807, 808, 809,
 815, 823, 831, 846, 855, 856, 857, 862,
 864, 865, 866, 867, 880, 881, 882, 883,
 884, 885, 886, 887, 888, 889, 890, 891,
 892, 893, 895, 896, 897, 898, 899, 900,
 901, 902, 903, 904, 910, 911, 912, 913,
 914, 915, 916, 917, 918, 919, 920, 921,
 922, 923, 924, 925, 926, 927, 928, 930,
 931, 932, 933, 934, 936, 937, 938, 939,
 941, 942, 943, 944, 945, 946, 947, 948,
 949, 950, 951, 952, 953, 954, 955, 956,
 957, 959, 960, 961, 962, 963, 964, 965,
 966, 967, 968, 969, 971, 972, 973, 974,
 975, 976, 977, 978, 979, 980, 981, 982,
 983, 984, 985, 986, 987, 988, 989, 992,
 993, 994, 995, 996, 997, 998, 999, 1000,
 1001, 1002, 1003, 1004, 1005, 1006, 1007,
 1008, 1009, 1010, 1011, 1013, 1014, 1015,

1016, 1017, 1018, 1019, 1020, 1021, 1022,
 1024, 1025, 1026, 1027, 1028, 1029, 1030,
 1031, 1032, 1033, 1034, 1035, 1036, 1037,
 1039, 1042, 1043, 1044, 1045, 1046, 1047,
 1048, 1049, 1050, 1051, 1053, 1054, 1055,
 1056, 1057, 1058, 1059, 1060, 1061, 1062,
 1063, 1066, 1067, 1068, 1069, 1070, 1071,
 1072, 1073, 1074, 1075, 1076, 1077, 1078,
 1080, 1081, 1082, 1083, 1085, 1086, 1090,
 1091, 1097, 1098, 1099, 1125, 1128, 1129,
 1130, 1131, 1134, 1135, 1136, 1137, 1138,
 1139, 1140, 1141, 1142, 1143, 1144, 1145,
 1146, 1147, 1148, 1149, 1150, 1151, 1152,
 1153, 1154, 1157, 1158, 1159, 1160, 1161,
 1162, 1164, 1165, 1166, 1167, 1168, 1169,
 1170, 1171, 1172, 1173, 1174, 1175, 1176,
 1177, 1180, 1181, 1182, 1183, 1184, 1186,
 1187, 1188, 1189, 1190, 1191, 1192, 1197,
 1198, 1201, 1202

ceiling, 678, 721

char, 64, 65, 66, 97, 98, 99, 108, 109, 117,

122, 125, 126, 145, 148, 149, 150, 153,
 154, 159, 168, 171, 174, 181, 184, 186,
 201, 203, 211, 212, 215, 217, 221, 222,
 223, 227, 228, 230, 231, 234, 235, 236,
 241, 242, 245, 303, 304, 305, 357, 358,
 359, 384, 399, 400, 409, 410, 438, 484,
 493, 512, 513, 540, 561, 562, 564, 565,
 582, 583, 597, 598, 712, 757, 771, 781,
 782, 797, 798, 829, 843, 862, 863, 864,
 865, 866, 969, 1045, 1063, 1064, 1113

char<, 66, 117, 227, 228, 230, 231, 358, 493,
 1063, 1064

char=, 122, 126, 241, 242, 712, 797, 798,
 1113

character, 241, 357, 358, 384, 408, 409,
 410, 1121

char-code, 64, 65, 108, 203, 358, 512, 513,
 564, 565, 597, 598, 757, 829

char-greaterp, 540

check-type, 409, 410

code-char, 64, 108, 357, 829

coerce, 64, 66, 102, 135, 148, 158, 181, 196,
 201, 228, 229, 234, 235, 246, 303, 304,
 405, 508, 525, 561, 562, 564, 565, 582,
 583, 820, 841, 865, 1129

concatenate, 65, 66, 94, 95, 96, 97, 98,
 102, 109, 110, 111, 113, 114, 120, 123,
 124, 125, 126, 127, 140, 141, 142, 144,
 145, 149, 152, 153, 154, 159, 161, 162,
 163, 164, 171, 173, 174, 175, 181, 184,

186, 188, 189, 190, 194, 195, 203, 204,
 205, 206, 207, 208, 209, 210, 211, 221,
 222, 223, 224, 228, 229, 230, 231, 235,
 245, 246, 260, 286, 287, 303, 318, 347,
 348, 349, 381, 382, 404, 408, 438, 505,
 506, 549, 564, 565, 581, 583, 602, 603,
 604, 613, 614, 627, 628, 629, 770, 771,
 813, 814, 815, 825, 839, 843, 845, 849,
 850, 851, 859, 860, 861, 862, 863, 866,
 874, 895, 932, 969, 1006, 1007, 1012,
 1063, 1064, 1096, 1097, 1106, 1109, 1137
 cond, 77, 78, 80, 89, 90, 91, 94, 95, 98, 99,
 101, 102, 103, 104, 110, 112, 113, 115,
 116, 120, 121, 122, 123, 124, 125, 126,
 127, 128, 129, 135, 136, 141, 146, 154,
 158, 165, 169, 170, 171, 172, 177, 178,
 179, 180, 183, 185, 192, 193, 195, 196,
 198, 199, 202, 203, 204, 205, 206, 207,
 208, 210, 216, 218, 222, 244, 247, 262,
 265, 266, 267, 268, 269, 270, 271, 272,
 273, 274, 275, 276, 277, 278, 280, 281,
 282, 283, 284, 285, 288, 290, 292, 293,
 295, 296, 297, 298, 300, 301, 302, 303,
 304, 305, 306, 308, 309, 310, 311, 312,
 313, 315, 316, 318, 319, 320, 321, 322,
 324, 325, 326, 327, 328, 330, 331, 332,
 333, 335, 336, 338, 339, 340, 342, 351,
 354, 357, 358, 359, 360, 361, 362, 364,
 365, 366, 367, 368, 371, 372, 373, 375,
 376, 377, 378, 379, 380, 381, 385, 386,
 387, 388, 389, 390, 391, 392, 393, 394,
 396, 397, 398, 399, 400, 401, 406, 410,
 411, 413, 415, 416, 417, 419, 420, 421,
 422, 424, 425, 426, 427, 432, 433, 434,
 437, 438, 440, 441, 442, 443, 444, 447,
 448, 449, 450, 451, 452, 453, 454, 455,
 456, 457, 458, 459, 460, 461, 462, 463,
 464, 466, 467, 469, 470, 471, 472, 473,
 474, 475, 476, 477, 478, 479, 481, 482,
 483, 485, 486, 487, 488, 489, 490, 491,
 493, 494, 495, 496, 497, 498, 499, 500,
 501, 502, 505, 506, 508, 509, 510, 511,
 512, 513, 514, 517, 518, 520, 521, 522,
 523, 524, 525, 526, 528, 530, 531, 532,
 533, 534, 535, 536, 537, 538, 539, 540,
 543, 546, 547, 548, 550, 551, 554, 555,
 558, 559, 560, 561, 562, 564, 566, 567,
 568, 569, 570, 571, 572, 573, 574, 577,
 579, 580, 581, 582, 583, 584, 585, 586,
 587, 588, 589, 591, 596, 598, 599, 601,
 602, 605, 606, 607, 608, 609, 610, 612,
 617, 618, 619, 620, 622, 623, 624, 625,
 626, 628, 629, 631, 632, 635, 636, 639,
 640, 641, 642, 643, 644, 645, 646, 647,
 648, 649, 651, 652, 653, 654, 655, 656,
 657, 659, 660, 661, 662, 663, 665, 666,
 667, 669, 670, 671, 672, 674, 675, 676,
 680, 681, 685, 689, 690, 691, 692, 695,
 696, 698, 699, 700, 701, 703, 710, 711,
 712, 713, 714, 715, 716, 722, 723, 724,
 725, 726, 727, 728, 732, 737, 738, 739,
 743, 746, 747, 748, 749, 750, 751, 753,
 754, 755, 757, 758, 760, 761, 763, 764,
 765, 768, 772, 774, 775, 776, 777, 778,
 781, 782, 784, 785, 786, 787, 789, 790,
 791, 792, 793, 794, 795, 796, 797, 800,
 805, 806, 807, 808, 812, 813, 816, 817,
 818, 821, 822, 825, 828, 831, 841, 845,
 846, 847, 848, 851, 852, 853, 855, 857,
 858, 860, 861, 864, 868, 869, 870, 876,
 878, 879, 880, 881, 882, 883, 884, 886,
 887, 888, 889, 890, 891, 892, 893, 895,
 896, 897, 901, 903, 904, 905, 910, 911,
 912, 913, 914, 915, 916, 917, 919, 921,
 922, 923, 924, 926, 927, 928, 930, 931,
 933, 934, 936, 938, 939, 940, 941, 943,
 944, 945, 946, 947, 948, 949, 950, 951,
 952, 953, 954, 955, 956, 957, 959, 960,
 961, 962, 963, 964, 965, 967, 968, 969,
 971, 972, 973, 974, 975, 976, 978, 979,
 980, 981, 982, 983, 984, 985, 986, 987,
 988, 989, 992, 993, 994, 995, 996, 998,
 999, 1000, 1002, 1005, 1008, 1009, 1013,
 1014, 1015, 1016, 1017, 1018, 1019, 1021,
 1022, 1024, 1025, 1028, 1029, 1030, 1031,
 1033, 1034, 1035, 1036, 1037, 1038, 1042,
 1044, 1045, 1047, 1048, 1049, 1050, 1051,
 1053, 1054, 1055, 1057, 1058, 1059, 1060,
 1062, 1063, 1066, 1067, 1069, 1071, 1072,
 1073, 1075, 1077, 1078, 1080, 1081, 1082,
 1083, 1085, 1086, 1089, 1090, 1091, 1096,
 1097, 1098, 1102, 1104, 1105, 1109, 1111,
 1119, 1121, 1124, 1125, 1126, 1128, 1129,
 1130, 1131, 1134, 1135, 1136, 1137, 1138,
 1139, 1140, 1141, 1142, 1144, 1145, 1146,
 1147, 1148, 1149, 1150, 1151, 1152, 1153,
 1154, 1157, 1158, 1159, 1160, 1162, 1164,
 1166, 1167, 1169, 1170, 1172, 1173, 1174,
 1175, 1176, 1180, 1182, 1183, 1184, 1186,
 1187, 1188, 1189, 1190, 1191, 1197, 1200,
 1201, 1202
 cons, 58, 69, 70, 73, 74, 80, 81, 100, 101,
 102, 103, 109, 110, 113, 116, 127, 141,
 151, 165, 170, 171, 177, 178, 179, 180,

181, 182, 183, 184, 185, 188, 189, 190,
 191, 192, 193, 194, 196, 197, 198, 201,
 202, 203, 204, 207, 210, 213, 216, 218,
 219, 221, 222, 223, 225, 237, 238, 243,
 244, 245, 249, 251, 252, 253, 258, 259,
 266, 267, 268, 272, 275, 280, 287, 293,
 297, 299, 300, 304, 305, 327, 328, 332,
 336, 341, 342, 343, 344, 345, 346, 347,
 348, 349, 356, 357, 369, 372, 373, 377,
 386, 394, 398, 406, 407, 411, 413, 414,
 415, 417, 419, 420, 421, 422, 424, 425,
 426, 427, 432, 433, 434, 435, 436, 437,
 438, 439, 440, 441, 442, 444, 445, 447,
 448, 449, 450, 451, 452, 453, 454, 455,
 456, 458, 459, 460, 461, 462, 463, 464,
 465, 466, 467, 468, 469, 470, 471, 472,
 474, 475, 476, 477, 478, 479, 480, 481,
 482, 483, 484, 485, 486, 487, 488, 489,
 490, 491, 492, 493, 494, 496, 497, 498,
 499, 501, 502, 503, 504, 505, 506, 507,
 509, 510, 511, 512, 513, 514, 515, 517,
 518, 519, 520, 521, 522, 523, 524, 526,
 527, 528, 529, 530, 531, 532, 533, 534,
 535, 536, 537, 538, 539, 540, 541, 543,
 544, 546, 547, 548, 549, 550, 552, 554,
 556, 558, 559, 560, 561, 562, 564, 565,
 566, 567, 568, 569, 570, 571, 572, 573,
 574, 575, 576, 577, 579, 580, 582, 583,
 584, 585, 586, 587, 588, 589, 590, 591,
 592, 593, 594, 595, 596, 598, 599, 600,
 601, 602, 605, 606, 607, 608, 609, 610,
 611, 612, 615, 616, 617, 618, 619, 620,
 621, 622, 623, 624, 625, 626, 629, 630,
 631, 632, 633, 635, 636, 637, 638, 639,
 640, 641, 642, 643, 644, 645, 646, 648,
 649, 650, 651, 652, 654, 656, 657, 659,
 660, 661, 662, 663, 664, 665, 666, 667,
 668, 669, 670, 671, 672, 674, 675, 676,
 677, 679, 680, 681, 682, 683, 684, 686,
 687, 688, 689, 690, 691, 692, 693, 694,
 695, 696, 698, 699, 700, 701, 702, 703,
 704, 706, 707, 708, 709, 710, 711, 712,
 713, 716, 717, 719, 720, 721, 722, 723,
 724, 725, 726, 727, 728, 729, 731, 733,
 734, 735, 737, 738, 739, 740, 743, 744,
 746, 747, 748, 749, 750, 751, 753, 757,
 758, 759, 761, 763, 764, 765, 766, 767,
 768, 769, 772, 773, 774, 775, 776, 777,
 778, 781, 782, 786, 787, 788, 789, 790,
 791, 792, 793, 794, 795, 796, 797, 800,
 802, 803, 804, 805, 806, 807, 808, 809,
 821, 822, 823, 828, 831, 841, 845, 846,

847, 848, 849, 850, 851, 852, 853, 855,
 856, 857, 858, 859, 862, 864, 865, 866,
 867, 868, 869, 870, 872, 876, 879, 886,
 890, 905, 911, 915, 919, 921, 924, 931,
 937, 938, 939, 940, 941, 942, 943, 944,
 945, 946, 948, 949, 950, 951, 952, 953,
 954, 955, 956, 957, 958, 959, 960, 961,
 962, 963, 964, 965, 966, 967, 968, 969,
 971, 972, 973, 974, 975, 976, 977, 978,
 979, 980, 981, 982, 983, 984, 985, 987,
 988, 989, 990, 991, 992, 993, 994, 995,
 996, 997, 998, 999, 1000, 1001, 1004,
 1005, 1006, 1007, 1008, 1009, 1010, 1011,
 1018, 1020, 1028, 1029, 1030, 1031, 1033,
 1034, 1035, 1036, 1037, 1039, 1042, 1043,
 1044, 1045, 1046, 1047, 1048, 1049, 1050,
 1051, 1052, 1053, 1054, 1055, 1056, 1057,
 1058, 1059, 1060, 1061, 1062, 1066, 1067,
 1068, 1070, 1071, 1072, 1073, 1074, 1075,
 1076, 1077, 1078, 1079, 1080, 1081, 1082,
 1083, 1084, 1085, 1086, 1090, 1091, 1092,
 1093, 1097, 1098, 1099, 1105, 1116, 1117,
 1120, 1121, 1123, 1124, 1125, 1126, 1127,
 1134, 1135, 1136, 1137, 1138, 1139, 1140,
 1141, 1142, 1144, 1147, 1148, 1149, 1150,
 1151, 1152, 1153, 1154, 1157, 1159, 1160,
 1161, 1162, 1165, 1167, 1168, 1169, 1170,
 1171, 1172, 1173, 1174, 1175, 1176, 1177,
 1181, 1182, 1183, 1184, 1185, 1186, 1187,
 1188, 1189, 1190, 1191, 1197, 1198, 1202,
 1203, 1204
 consp, 205, 206, 207, 208, 209, 290, 434,
 481, 485, 486, 499, 566, 889, 950, 1149,
 1150
 constantly, 407, 419, 1116
 copy-list, 258, 259, 504, 637, 1073, 1159
 copy-seq, 1002, 1047
 count, 68, 89, 90, 120, 121, 134, 135, 140,
 143, 144, 163, 168, 185, 186, 202, 203,
 205, 206, 207, 208, 209, 210, 211, 213,
 214, 221, 222, 223, 224, 245, 262, 263,
 265, 266, 272, 279, 280, 281, 283, 284,
 285, 288, 289, 297, 298, 305, 306, 315,
 316, 317, 323, 324, 325, 327, 328, 329,
 332, 333, 334, 335, 336, 346, 347, 348,
 349, 351, 352, 361, 362, 363, 364, 365,
 367, 368, 374, 388, 393, 395, 396, 398,
 401, 402, 419, 420, 438, 447, 448, 471,
 482, 483, 484, 496, 497, 500, 511, 518,
 524, 530, 532, 544, 561, 577, 584, 587,
 592, 593, 596, 613, 614, 630, 631, 638,
 639, 643, 644, 645, 647, 658, 666, 673,

679, 681, 682, 690, 691, 695, 696, 697,
703, 714, 718, 719, 727, 728, 731, 733,
736, 737, 739, 740, 741, 749, 750, 764,
767, 771, 772, 773, 776, 777, 778, 779,
806, 807, 824, 835, 839, 840, 841, 842,
849, 850, 851, 852, 859, 863, 864, 865,
866, 867, 880, 881, 884, 885, 888, 889,
890, 895, 896, 897, 898, 905, 913, 914,
915, 916, 917, 918, 919, 925, 926, 927,
928, 934, 935, 941, 942, 943, 944, 955,
977, 983, 996, 997, 1001, 1012, 1013,
1055, 1056, 1069, 1070, 1120, 1131, 1132,
1138, 1139, 1149, 1156, 1162, 1170
count-if, 68, 205, 206, 207, 208, 209, 272,
280, 281, 283, 289, 306, 323, 324, 325,
329, 333, 334, 351, 361, 365, 398, 402,
496, 497, 577, 587, 613, 614, 647, 727,
728, 749, 750, 764, 859, 895, 896, 897,
905, 915, 916, 917, 926, 996, 997, 1132,
1139
count-if-not, 306, 334, 915, 926

D

decf, 1047, 1116
declaim, 238
decode-universal-time, 1116
defconstant, 64, 167, 439, 494, 495, 878
define-modify-macro, 756
defmacro, 396, 397, 423, 467, 556, 730, 756
defpackage, 58, 976
defparameter, 232, 237, 238, 271, 408,
409, 423, 517, 518, 552, 683, 684, 685,
686, 687, 688, 712, 1001, 1037, 1038,
1172, 1173
defstruct, 299, 344, 345, 346, 521, 522,
530, 536, 552, 616, 704, 733, 734, 735,
990, 991
delete, 100, 102, 103, 120, 125, 145, 174,
184, 186, 221, 222, 223, 245, 265, 266,
280, 281, 289, 417, 419, 420, 438, 454,
482, 499, 500, 504, 507, 513, 524, 533,
534, 538, 545, 580, 586, 587, 630, 769,
825, 843, 862, 863, 885, 941, 942, 944,
955, 957, 969, 1029, 1046, 1070, 1097,
1098, 1099, 1135, 1136
delete-duplicates, 100, 419, 482, 1029
delete-if, 102, 103, 120, 125, 145, 174,
184, 186, 221, 222, 223, 245, 280, 281,
289, 417, 419, 420, 438, 500, 504, 507,
513, 524, 545, 580, 586, 587, 630, 769,

825, 843, 862, 863, 885, 941, 942, 955,
957, 969, 1046, 1135, 1136
delete-if-not, 102, 103, 120, 125, 145,
174, 184, 186, 221, 222, 223, 245, 289,
438, 504, 524, 580, 586, 587, 769, 843,
862, 863, 942, 969, 1135, 1136
destructuring-bind, 155, 383, 1047
digit-char-p, 64, 148, 149, 150, 171, 181,
201, 215, 217, 234, 235, 236, 242, 303,
304, 305, 561, 562, 564, 863, 864, 865, 866
do, 110, 114, 117, 119, 122, 139, 141, 142,
152, 157, 182, 184, 189, 194, 195, 205,
206, 207, 208, 210, 211, 212, 216, 218,
221, 222, 223, 224, 226, 230, 231, 233,
237, 238, 241, 242, 314, 315, 317, 374,
377, 379, 382, 383, 384, 394, 416, 423,
429, 430, 431, 519, 538, 565, 575, 576,
603, 604, 612, 677, 686, 687, 688, 705,
715, 717, 741, 742, 756, 835, 836, 839,
847, 849, 850, 851, 867, 868, 873, 874,
880, 894, 902, 905, 913, 928, 952, 970,
971, 973, 1001, 1006, 1007, 1025, 1047,
1060, 1074, 1094, 1104, 1130, 1133, 1155,
1178, 1196
dolist, 186, 222, 223, 266, 422, 466, 473,
475, 532, 559, 576, 597, 598, 685, 700,
804, 1001, 1165, 1168, 1169
dotimes, 183, 186, 226, 237, 238, 239, 423,
805

E

elt, 67, 122, 264, 323, 654, 885, 921, 922,
939, 940, 957, 1172, 1173
encode-universal-time, 1020, 1116
eq, 71, 74, 113, 134, 135, 180, 259, 310, 311,
312, 319, 334, 336, 338, 389, 394, 430,
438, 446, 490, 491, 492, 513, 518, 522,
523, 525, 528, 530, 531, 536, 553, 574,
580, 585, 588, 589, 596, 599, 613, 614,
627, 638, 640, 641, 645, 657, 658, 660,
664, 665, 687, 688, 725, 726, 863, 864,
881, 883, 884, 892, 893, 897, 902, 903,
913, 914, 915, 954, 957, 969, 987, 988,
1016, 1019, 1027, 1033, 1047, 1060, 1061,
1062, 1100, 1101, 1119, 1130, 1131, 1135,
1136, 1138, 1148, 1149, 1150, 1151, 1152,
1153, 1157, 1158, 1164, 1166, 1169, 1180,
1181, 1185, 1186, 1187, 1188, 1189
eq1, 72, 78, 114, 117, 119, 122, 135, 141,
142, 162, 174, 184, 194, 195, 205, 206,

207, 208, 210, 211, 221, 222, 223, 224,
 230, 231, 247, 491, 498, 517, 518, 519,
 524, 525, 548, 550, 603, 604, 754, 755,
 760, 761, 814, 835, 841, 843, 849, 850,
 851, 882, 888, 889, 891, 892, 939, 941,
 942, 978, 1001, 1006, 1007, 1027, 1101,
 1134, 1135, 1137

equal, 65, 72, 80, 97, 98, 99, 111, 112, 122,
 123, 124, 125, 126, 141, 145, 153, 154,
 168, 173, 174, 184, 186, 203, 211, 221,
 222, 223, 245, 246, 268, 271, 274, 276,
 297, 298, 319, 329, 330, 337, 359, 377,
 385, 386, 387, 388, 434, 438, 496, 497,
 498, 499, 503, 505, 506, 566, 567, 568,
 569, 570, 574, 579, 580, 584, 585, 597,
 598, 628, 629, 638, 639, 706, 707, 725,
 729, 730, 731, 739, 740, 741, 742, 756,
 764, 776, 777, 778, 781, 782, 807, 825,
 831, 832, 843, 862, 863, 864, 868, 869,
 877, 880, 881, 887, 888, 893, 916, 922,
 930, 934, 935, 939, 941, 965, 966, 967,
 968, 969, 974, 997, 999, 1009, 1010, 1013,
 1016, 1017, 1018, 1019, 1020, 1027, 1036,
 1048, 1097, 1098, 1099, 1134, 1159, 1164

equalp, 72, 111, 198, 199, 262, 263, 267,
 274, 377, 385, 437, 455, 456, 523, 552,
 553, 579, 611, 658, 666, 682, 693, 694,
 695, 696, 697, 698, 699, 704, 706, 707,
 722, 723, 733, 734, 735, 781, 904, 928,
 933, 934, 945, 959, 983, 984, 997, 1001,
 1006, 1007, 1027, 1047, 1061, 1062, 1063,
 1092, 1093, 1129, 1131, 1157, 1160, 1161,
 1162, 1176, 1177, 1184, 1203

eval, 73, 162, 197, 306, 332, 343, 357, 386,
 411, 489, 631, 637, 638, 654, 725, 726,
 753, 788, 789, 814, 874, 911, 980, 1155,
 1166, 1169

evenp, 68, 73, 76, 77, 83, 136, 137, 143, 145,
 146, 147, 162, 169, 170, 194, 195, 210,
 281, 282, 283, 285, 292, 293, 294, 334,
 335, 336, 340, 341, 342, 353, 360, 361,
 364, 365, 368, 379, 380, 435, 461, 489,
 503, 504, 513, 524, 525, 528, 577, 578,
 596, 607, 609, 616, 655, 689, 690, 744,
 802, 855, 856, 869, 870, 890, 910, 911,
 946, 947, 948, 949, 956, 985, 1040, 1041,
 1094, 1132, 1147, 1148, 1154, 1156, 1159

every, 76, 92, 131, 169, 220, 264, 268, 289,
 295, 296, 327, 334, 340, 342, 343, 344,
 362, 370, 371, 395, 408, 412, 428, 446,
 504, 541, 637, 638, 702, 706, 713, 738,
 758, 759, 802, 844, 894, 910, 911, 913,

975, 976, 1007, 1009, 1014, 1016, 1023,
 1108, 1114, 1132, 1133, 1145, 1146, 1182,
 1203

expt, 71, 79, 82, 84, 88, 89, 90, 93, 101, 119,
 120, 130, 131, 133, 134, 139, 140, 143,
 144, 147, 164, 169, 170, 185, 200, 201,
 215, 226, 234, 235, 236, 237, 238, 247,
 308, 317, 318, 322, 336, 435, 503, 504,
 623, 624, 709, 710, 743, 812, 816, 817,
 818, 824, 825, 826, 827, 832, 834, 963,
 964, 973, 989, 990, 1021, 1022, 1033,
 1074, 1075, 1101, 1109, 1110, 1118, 1159

F

fill, 258, 259, 384, 1130

find, 68, 147, 148, 175, 242, 250, 261, 270,
 273, 301, 350, 352, 358, 359, 377, 391,
 402, 425, 426, 446, 471, 482, 483, 492,
 502, 539, 659, 679, 731, 820, 875, 882,
 892, 894, 897, 999, 1036, 1114, 1129, 1163

find-if, 147, 148, 242, 270, 273, 301, 352,
 539, 897, 999

first, 59, 70, 139, 220, 230, 264, 289, 306,
 310, 311, 338, 339, 347, 375, 376, 377,
 394, 409, 499, 500, 528, 547, 548, 589,
 706, 710, 711, 714, 715, 784, 785, 808,
 871, 875, 905, 906, 913, 918, 922, 926,
 941, 942, 954, 970, 1008, 1017, 1018,
 1023, 1024, 1081, 1094, 1115, 1132, 1149,
 1155, 1163, 1192, 1193

float, 64, 73, 119, 120, 131, 160, 161, 211,
 212, 214, 309, 359, 457, 503, 584, 585,
 672, 673, 680, 681, 703, 756, 764, 767,
 810, 850, 851, 1012, 1106, 1194, 1195,
 1196, 1200

floor, 122, 142, 143, 144, 162, 184, 215,
 226, 232, 239, 244, 245, 287, 317, 318,
 455, 496, 497, 562, 563, 634, 667, 668,
 677, 678, 699, 877, 1002

format, 66, 83, 84, 86, 87, 88, 90, 91, 94, 96,
 101, 102, 103, 117, 118, 119, 122, 125,
 144, 145, 150, 156, 173, 175, 180, 182,
 184, 185, 186, 224, 234, 239, 240, 241,
 247, 260, 269, 286, 304, 349, 350, 382,
 395, 404, 405, 408, 428, 431, 495, 496,
 497, 501, 519, 581, 582, 756, 793, 794,
 795, 825, 830, 832, 833, 840, 843, 867,
 874, 893, 932, 970, 1012, 1097, 1098,
 1099, 1103, 1106, 1111, 1116, 1122

fresh-line, 247

funcall, 77, 331, 332, 454, 508, 524, 528,
 546, 564, 565, 601, 624, 651, 683, 684,
 685, 686, 687, 688, 752, 753, 766, 786,
 787, 890, 896, 901, 910, 911, 917, 918,
 937, 944, 945, 946, 947, 948, 949, 956,
 963, 973, 975, 976, 982, 983, 985, 998,
 1001, 1040, 1145, 1146, 1147, 1149, 1150,
 1162, 1169, 1174, 1175, 1194, 1195, 1196,
 1197, 1198
 function, 65, 104, 105, 106, 107, 168, 248,
 281, 372, 395, 412, 415, 427, 430, 784,
 803, 874, 918, 948, 1110, 1115, 1136,
 1163, 1169, 1170

G

gcd, 353, 354, 815, 816
 gethash, 186, 212, 223, 532, 741, 742, 756,
 804
 get-output-stream-string, 241, 242

I

identity, 65, 149, 181, 228, 229, 235, 246,
 338, 365, 420, 507, 582, 637, 638, 737,
 770, 832, 859, 866, 885, 897, 941, 955
 if, 67, 68, 77, 78, 79, 80, 81, 83, 90, 95, 97,
 98, 99, 100, 102, 103, 104, 105, 107, 109,
 110, 112, 113, 115, 116, 117, 119, 120,
 121, 122, 124, 125, 126, 127, 129, 131,
 132, 135, 136, 137, 138, 139, 141, 142,
 143, 144, 145, 146, 147, 148, 151, 153,
 154, 156, 157, 158, 159, 161, 162, 165,
 166, 167, 169, 170, 171, 172, 173, 174,
 175, 177, 178, 179, 181, 182, 184, 185,
 186, 188, 189, 190, 191, 192, 193, 194,
 195, 196, 197, 199, 201, 202, 203, 205,
 206, 207, 208, 209, 210, 213, 216, 217,
 218, 219, 220, 221, 222, 223, 225, 226,
 228, 229, 230, 231, 232, 237, 238, 241,
 242, 243, 244, 245, 247, 248, 249, 257,
 259, 262, 263, 264, 265, 266, 268, 269,
 270, 271, 272, 273, 275, 276, 277, 279,
 280, 281, 282, 283, 284, 285, 287, 289,
 291, 292, 293, 294, 295, 298, 301, 302,
 303, 304, 305, 306, 307, 308, 309, 310,
 312, 313, 314, 315, 316, 317, 318, 319,
 320, 322, 323, 324, 325, 327, 329, 330,
 331, 333, 334, 335, 336, 337, 339, 340,
 341, 342, 343, 348, 349, 350, 351, 352,
 353, 354, 355, 356, 357, 359, 360, 361,

362, 363, 364, 365, 366, 367, 368, 370,
 371, 372, 373, 374, 375, 377, 380, 381,
 385, 386, 387, 388, 389, 390, 392, 393,
 394, 395, 396, 397, 398, 399, 401, 402,
 405, 407, 408, 409, 410, 413, 414, 416,
 417, 418, 419, 420, 421, 422, 423, 424,
 425, 426, 430, 433, 434, 435, 437, 438,
 442, 443, 444, 445, 446, 448, 450, 451,
 452, 453, 454, 455, 456, 459, 460, 461,
 463, 464, 466, 467, 468, 470, 472, 473,
 474, 475, 477, 478, 479, 480, 482, 483,
 484, 485, 487, 488, 490, 491, 492, 495,
 496, 497, 498, 499, 500, 501, 503, 504,
 505, 506, 507, 509, 512, 513, 515, 516,
 517, 518, 520, 521, 524, 526, 527, 528,
 529, 530, 532, 533, 535, 538, 539, 540,
 541, 542, 543, 545, 546, 547, 548, 551,
 553, 554, 555, 556, 557, 558, 559, 560,
 561, 562, 563, 564, 566, 567, 568, 570,
 571, 572, 573, 574, 575, 576, 577, 578,
 580, 583, 584, 585, 586, 587, 588, 589,
 590, 591, 592, 593, 594, 595, 596, 597,
 598, 599, 600, 601, 602, 603, 606, 607,
 608, 609, 611, 612, 613, 614, 615, 616,
 617, 618, 619, 620, 621, 622, 623, 624,
 625, 626, 627, 629, 630, 631, 632, 635,
 638, 639, 640, 641, 642, 644, 645, 646,
 647, 648, 649, 650, 651, 652, 653, 654,
 655, 656, 657, 658, 659, 660, 664, 665,
 666, 668, 669, 670, 671, 672, 673, 674,
 675, 676, 677, 678, 679, 680, 682, 683,
 684, 685, 686, 687, 688, 689, 690, 691,
 692, 694, 695, 696, 697, 699, 701, 702,
 703, 706, 707, 709, 710, 711, 712, 713,
 714, 715, 716, 717, 718, 720, 721, 722,
 723, 724, 726, 727, 728, 729, 730, 732,
 738, 739, 740, 741, 742, 743, 744, 745,
 746, 747, 748, 749, 750, 752, 753, 754,
 756, 757, 758, 759, 760, 762, 763, 764,
 765, 766, 767, 768, 769, 772, 773, 774,
 775, 776, 777, 778, 779, 780, 781, 782,
 783, 784, 790, 791, 792, 793, 794, 798,
 800, 802, 803, 804, 806, 807, 808, 809,
 811, 812, 813, 814, 815, 816, 817, 818,
 819, 820, 822, 823, 824, 825, 826, 827,
 828, 829, 831, 833, 834, 839, 841, 842,
 843, 846, 847, 848, 852, 853, 854, 855,
 856, 857, 858, 859, 860, 861, 862, 863,
 864, 865, 866, 867, 868, 869, 872, 873,
 877, 878, 879, 880, 883, 884, 885, 886,
 887, 888, 889, 890, 891, 892, 893, 894,
 895, 896, 897, 898, 899, 900, 901, 902,

903, 904, 905, 910, 911, 912, 913, 914,
 915, 916, 917, 919, 920, 921, 922, 923,
 924, 925, 926, 927, 928, 929, 930, 931,
 932, 933, 934, 935, 936, 937, 938, 939,
 940, 941, 942, 943, 945, 946, 947, 950,
 951, 953, 954, 955, 957, 958, 959, 960,
 961, 962, 963, 965, 966, 967, 968, 969,
 971, 972, 974, 975, 976, 977, 978, 979,
 980, 982, 983, 984, 985, 988, 989, 991,
 994, 995, 996, 997, 998, 999, 1000, 1001,
 1002, 1003, 1004, 1006, 1007, 1008, 1009,
 1010, 1012, 1016, 1018, 1019, 1020, 1022,
 1023, 1024, 1026, 1027, 1028, 1030, 1031,
 1034, 1035, 1036, 1041, 1043, 1044, 1045,
 1046, 1047, 1051, 1052, 1053, 1057, 1058,
 1059, 1069, 1070, 1071, 1072, 1073, 1074,
 1075, 1076, 1077, 1078, 1079, 1080, 1081,
 1083, 1084, 1085, 1086, 1087, 1090, 1092,
 1093, 1094, 1095, 1096, 1098, 1099, 1100,
 1103, 1104, 1110, 1111, 1112, 1115, 1118,
 1120, 1121, 1123, 1124, 1125, 1128, 1129,
 1130, 1131, 1132, 1133, 1134, 1135, 1136,
 1137, 1139, 1140, 1142, 1143, 1144, 1145,
 1146, 1147, 1148, 1149, 1151, 1152, 1154,
 1155, 1156, 1157, 1158, 1159, 1160, 1161,
 1162, 1167, 1168, 1170, 1171, 1172, 1173,
 1174, 1175, 1176, 1177, 1180, 1181, 1185,
 1186, 1187, 1188, 1189, 1191, 1192, 1197,
 1198, 1202, 1203, 1204, 1240
 incf, 122, 186, 212, 222, 223, 239, 291, 378,
 409, 416, 417, 532, 612, 685, 686, 687,
 688, 694, 741, 742, 804, 905, 928, 1047,
 1196
 initial-value, 123, 124, 317, 465, 466,
 544, 571, 665
 in-package, 58, 976
 interp, 138, 269, 321, 590, 591, 615,
 751, 756, 757, 758, 860, 861, 862, 920,
 921, 924, 1033
 intern, 357, 358, 431, 438, 484, 505, 508,
 564, 565, 581, 582, 874, 1063, 1064, 1065,
 1066, 1137, 1202
 intersection, 76, 449, 501, 508, 549, 843,
 845, 873, 1041, 1044, 1045, 1055, 1056,
 1061, 1062, 1077, 1078, 1079, 1080, 1081,
 1091, 1092, 1093, 1119, 1177
 isqrt, 138, 185, 209, 239, 240, 241, 445,
 579, 823, 836, 847, 848, 871, 875, 876,
 893, 894, 905

L

labels, 90, 91, 95, 99, 101, 121, 125, 128,
 129, 132, 137, 138, 151, 183, 284, 308,
 314, 377, 458, 460, 471, 524, 533, 534,
 561, 567, 568, 569, 577, 592, 593, 636,
 675, 676, 720, 721, 723, 725, 728, 750,
 753, 754, 855, 856, 892, 947, 961, 962,
 971, 999, 1000, 1095, 1099, 1143, 1191
 lambda, 82, 94, 96, 97, 98, 100, 102, 103,
 123, 124, 125, 139, 145, 152, 153, 154,
 155, 159, 163, 174, 180, 184, 186, 196,
 199, 200, 202, 203, 206, 212, 217, 218,
 221, 222, 223, 227, 228, 229, 234, 235,
 236, 237, 238, 240, 241, 242, 245, 251,
 252, 253, 254, 255, 256, 257, 266, 268,
 270, 271, 272, 275, 276, 280, 284, 286,
 287, 291, 294, 299, 305, 306, 310, 312,
 317, 320, 323, 324, 325, 327, 329, 331,
 334, 340, 341, 348, 349, 350, 352, 353,
 357, 358, 361, 365, 389, 393, 400, 402,
 409, 414, 415, 416, 417, 422, 427, 428,
 429, 430, 431, 434, 435, 436, 438, 439,
 441, 443, 444, 446, 448, 453, 454, 455,
 456, 457, 458, 460, 465, 469, 470, 474,
 476, 478, 479, 482, 483, 486, 487, 492,
 495, 496, 503, 504, 506, 507, 508, 510,
 512, 513, 514, 515, 516, 518, 520, 528,
 529, 530, 532, 535, 537, 540, 542, 546,
 547, 549, 551, 554, 562, 564, 565, 567,
 574, 578, 579, 580, 581, 586, 587, 588,
 590, 591, 594, 597, 598, 601, 602, 606,
 612, 613, 614, 616, 617, 619, 621, 623,
 627, 628, 629, 630, 631, 632, 633, 634,
 637, 638, 639, 640, 641, 643, 644, 645,
 648, 649, 652, 653, 654, 655, 656, 658,
 659, 660, 662, 664, 665, 669, 670, 671,
 673, 674, 681, 682, 685, 690, 691, 692,
 693, 694, 696, 697, 700, 709, 718, 722,
 729, 731, 739, 740, 741, 742, 756, 759,
 762, 764, 765, 766, 767, 769, 770, 771,
 772, 774, 775, 777, 782, 788, 789, 794,
 798, 801, 804, 805, 806, 807, 809, 825,
 829, 843, 846, 857, 860, 861, 862, 863,
 868, 874, 881, 882, 886, 887, 905, 915,
 916, 918, 919, 925, 926, 927, 930, 931,
 934, 938, 942, 944, 945, 947, 950, 951,
 952, 956, 957, 958, 964, 968, 969, 972,
 974, 977, 979, 980, 984, 990, 991, 994,
 996, 999, 1000, 1004, 1008, 1011, 1014,
 1016, 1017, 1018, 1028, 1032, 1033, 1040,
 1041, 1042, 1044, 1046, 1051, 1058, 1062,

```

1063, 1064, 1067, 1068, 1069, 1071, 1076,
1086, 1088, 1092, 1093, 1132, 1135, 1136,
1137, 1139, 1145, 1146, 1148, 1153, 1154,
1159, 1161, 1162, 1164, 1166, 1169, 1171,
1177, 1180, 1181, 1185, 1186, 1187, 1191,
1194, 1195, 1196, 1202, 1203
last, 75, 126, 127, 147, 148, 157, 220, 250,
263, 264, 270, 272, 273, 274, 289, 300,
310, 311, 325, 347, 350, 352, 367, 368,
409, 413, 420, 428, 445, 449, 451, 452,
486, 503, 504, 507, 525, 541, 547, 548,
556, 574, 575, 615, 636, 659, 660, 665,
680, 706, 707, 712, 719, 720, 746, 754,
755, 756, 760, 761, 802, 803, 882, 883,
884, 885, 886, 887, 888, 921, 929, 954,
955, 957, 958, 959, 1072, 1081, 1084,
1132, 1159, 1163
lcm, 340
ldiff, 420
length, 67, 75, 79, 80, 88, 89, 90, 96, 97, 98,
99, 101, 103, 109, 110, 111, 112, 113, 114,
115, 117, 119, 122, 130, 131, 134, 135,
140, 152, 153, 154, 158, 159, 163, 165,
171, 173, 174, 175, 188, 189, 190, 204,
205, 206, 207, 208, 209, 211, 234, 235,
236, 237, 238, 239, 240, 241, 242, 243,
251, 252, 253, 254, 255, 256, 257, 260,
265, 271, 275, 276, 290, 301, 302, 303,
309, 311, 334, 335, 352, 359, 368, 372,
373, 378, 384, 391, 394, 399, 400, 409,
411, 412, 416, 417, 419, 421, 422, 423,
445, 454, 455, 457, 458, 469, 486, 490,
491, 503, 515, 516, 525, 541, 542, 543,
544, 545, 548, 549, 556, 562, 563, 571,
572, 577, 583, 584, 585, 586, 588, 594,
603, 604, 610, 613, 614, 615, 624, 625,
634, 648, 649, 655, 667, 668, 672, 673,
677, 678, 679, 680, 681, 683, 684, 685,
686, 687, 688, 689, 703, 706, 707, 708,
709, 721, 724, 729, 731, 732, 736, 737,
747, 748, 757, 763, 764, 767, 781, 796,
797, 798, 805, 810, 821, 822, 823, 825,
831, 850, 851, 858, 859, 884, 889, 897,
898, 899, 900, 901, 935, 946, 952, 978,
981, 1001, 1002, 1060, 1061, 1062, 1071,
1074, 1088, 1091, 1092, 1093, 1108, 1109,
1113, 1124, 1144, 1179, 1200
let, 96, 97, 98, 99, 148, 150, 152, 153, 154,
155, 183, 184, 193, 203, 204, 221, 222,
223, 238, 239, 241, 247, 249, 265, 271,
327, 347, 348, 349, 363, 376, 409, 447,
456, 458, 466, 468, 476, 511, 546, 547,
550, 594, 603, 626, 627, 631, 650, 677,
685, 714, 715, 730, 731, 752, 815, 844,
850, 851, 874, 878, 887, 949, 952, 956,
958, 965, 966, 967, 996, 1047, 1063, 1064,
1065, 1096, 1161, 1173, 1196
let*, 148, 150, 265, 376, 594, 714, 715, 1047
list, 64, 65, 66, 74, 76, 78, 81, 93, 96, 97,
98, 102, 108, 109, 110, 113, 114, 123, 124,
126, 127, 135, 141, 146, 147, 148, 149,
150, 152, 153, 154, 163, 165, 168, 174,
181, 182, 183, 184, 185, 186, 188, 189,
190, 191, 192, 193, 197, 198, 199, 200,
201, 203, 204, 205, 206, 207, 208, 211,
212, 213, 214, 220, 222, 223, 226, 227,
228, 229, 232, 234, 235, 236, 237, 238,
239, 240, 241, 242, 245, 246, 247, 249,
254, 258, 259, 260, 264, 273, 283, 284,
289, 299, 303, 304, 305, 306, 311, 321,
322, 347, 350, 357, 358, 376, 377, 389,
390, 391, 408, 409, 410, 411, 412, 414,
415, 416, 417, 418, 422, 423, 424, 425,
426, 427, 428, 431, 432, 434, 438, 443,
444, 445, 449, 450, 452, 453, 454, 455,
456, 457, 460, 461, 462, 463, 464, 465,
466, 467, 468, 469, 471, 472, 475, 479,
480, 482, 483, 484, 487, 488, 489, 490,
493, 494, 495, 503, 504, 505, 506, 507,
508, 509, 510, 511, 513, 517, 518, 519,
525, 529, 530, 532, 533, 534, 538, 539,
544, 545, 547, 548, 549, 551, 554, 556,
557, 558, 559, 561, 562, 564, 565, 566,
574, 577, 582, 583, 584, 585, 587, 588,
592, 595, 597, 598, 602, 603, 604, 606,
611, 612, 613, 614, 615, 617, 619, 620,
624, 628, 630, 631, 635, 636, 637, 638,
639, 640, 641, 642, 643, 644, 645, 646,
647, 648, 649, 650, 655, 656, 659, 660,
661, 663, 664, 665, 667, 669, 674, 675,
678, 679, 680, 685, 686, 687, 688, 694,
696, 699, 702, 705, 706, 707, 709, 712,
713, 714, 715, 716, 718, 719, 720, 725,
727, 728, 729, 730, 731, 733, 736, 738,
739, 740, 741, 744, 745, 746, 747, 748,
749, 750, 751, 752, 753, 754, 756, 762,
763, 764, 765, 766, 767, 771, 772, 773,
776, 780, 784, 785, 786, 787, 789, 790,
791, 792, 793, 794, 795, 796, 798, 802,
804, 805, 806, 807, 817, 828, 831, 832,
842, 843, 846, 850, 851, 853, 854, 856,
860, 861, 862, 863, 865, 866, 867, 873,
877, 879, 889, 917, 919, 920, 923, 931,
940, 946, 947, 952, 953, 954, 960, 968,

```

973, 978, 987, 990, 998, 1002, 1003, 1004,
 1007, 1008, 1010, 1012, 1020, 1029, 1030,
 1031, 1032, 1034, 1040, 1042, 1047, 1050,
 1055, 1063, 1064, 1065, 1066, 1067, 1070,
 1072, 1073, 1075, 1084, 1089, 1091, 1097,
 1098, 1099, 1102, 1117, 1118, 1123, 1124,
 1125, 1126, 1135, 1137, 1141, 1142, 1147,
 1149, 1150, 1158, 1159, 1164, 1165, 1166,
 1168, 1169, 1192, 1193, 1200
 list*, 417, 418, 1031
 listp, 71, 72, 205, 206, 268, 269, 271, 272,
 273, 275, 280, 293, 294, 313, 321, 322,
 324, 325, 341, 352, 353, 366, 370, 371,
 373, 374, 389, 390, 391, 394, 397, 411,
 412, 421, 422, 428, 442, 443, 447, 448,
 454, 459, 461, 462, 467, 471, 475, 476,
 477, 483, 484, 487, 489, 495, 496, 497,
 498, 517, 520, 521, 532, 535, 538, 539,
 545, 551, 554, 566, 567, 568, 577, 597,
 616, 619, 620, 626, 627, 632, 644, 645,
 649, 650, 652, 653, 667, 668, 670, 671,
 701, 702, 712, 713, 725, 726, 727, 728,
 738, 739, 740, 747, 749, 750, 751, 757,
 758, 759, 761, 762, 772, 776, 779, 798,
 803, 804, 808, 809, 882, 886, 887, 888,
 903, 914, 915, 917, 922, 924, 925, 930,
 935, 936, 961, 962, 963, 968, 979, 980,
 982, 983, 987, 988, 997, 1002, 1003, 1004,
 1005, 1006, 1009, 1010, 1015, 1016, 1018,
 1030, 1031, 1046, 1051, 1052, 1058, 1075,
 1079, 1080, 1083, 1084, 1085, 1086, 1123,
 1124, 1135, 1151, 1152, 1154, 1157, 1158,
 1171, 1172, 1173, 1184
 log, 1195, 1196
 loop, 65, 67, 82, 83, 88, 89, 90, 91, 93, 97,
 102, 106, 107, 108, 109, 110, 111, 112,
 115, 120, 126, 132, 134, 135, 138, 139,
 140, 142, 143, 145, 147, 148, 152, 153,
 158, 161, 162, 163, 166, 168, 169, 171,
 172, 174, 175, 177, 179, 180, 181, 182,
 185, 186, 189, 190, 191, 194, 196, 198,
 199, 200, 201, 202, 205, 208, 209, 210,
 211, 212, 213, 214, 215, 216, 217, 218,
 219, 220, 223, 226, 227, 228, 229, 231,
 232, 234, 235, 236, 237, 238, 239, 240,
 241, 242, 243, 244, 245, 246, 247, 249,
 251, 252, 253, 254, 255, 256, 257, 258,
 259, 260, 263, 266, 268, 272, 275, 280,
 282, 289, 291, 293, 294, 301, 302, 304,
 306, 307, 309, 311, 313, 314, 315, 316,
 319, 321, 322, 323, 330, 332, 333, 334,
 335, 339, 340, 341, 343, 344, 345, 346,
 350, 351, 352, 353, 355, 356, 359, 360,
 361, 365, 367, 369, 370, 373, 374, 377,
 378, 380, 381, 382, 383, 384, 385, 387,
 388, 389, 390, 391, 393, 394, 396, 397,
 398, 400, 401, 402, 403, 404, 405, 408,
 409, 410, 411, 418, 419, 422, 424, 425,
 426, 428, 433, 434, 436, 439, 442, 444,
 445, 446, 448, 449, 451, 457, 458, 459,
 460, 463, 464, 465, 467, 470, 474, 475,
 480, 481, 483, 484, 485, 487, 495, 497,
 498, 499, 500, 515, 519, 520, 521, 522,
 525, 529, 530, 532, 535, 537, 541, 542,
 543, 544, 545, 546, 548, 550, 551, 552,
 553, 554, 556, 557, 560, 561, 566, 567,
 568, 575, 576, 577, 578, 579, 582, 587,
 590, 597, 601, 605, 606, 607, 608, 610,
 611, 613, 614, 617, 619, 620, 621, 622,
 624, 625, 626, 627, 629, 630, 631, 632,
 633, 636, 639, 641, 642, 644, 645, 646,
 647, 649, 650, 651, 652, 653, 654, 655,
 656, 657, 658, 659, 662, 663, 664, 666,
 667, 668, 671, 673, 674, 678, 679, 681,
 683, 684, 685, 686, 687, 688, 690, 693,
 694, 695, 696, 700, 701, 702, 703, 704,
 705, 706, 708, 709, 710, 712, 713, 714,
 715, 717, 718, 719, 723, 724, 726, 727,
 729, 730, 731, 732, 733, 734, 735, 736,
 737, 738, 739, 740, 741, 742, 743, 744,
 745, 747, 749, 753, 756, 757, 758, 759,
 762, 763, 764, 765, 767, 768, 769, 770,
 771, 772, 773, 774, 775, 776, 777, 778,
 779, 780, 781, 783, 784, 785, 786, 787,
 788, 789, 790, 791, 794, 795, 797, 798,
 800, 801, 803, 805, 806, 807, 808, 809,
 811, 814, 816, 818, 820, 823, 824, 825,
 827, 834, 835, 836, 839, 840, 841, 844,
 846, 847, 848, 849, 850, 851, 852, 853,
 854, 855, 856, 857, 859, 860, 861, 862,
 863, 864, 867, 868, 869, 870, 871, 872,
 873, 874, 875, 876, 877, 878, 879, 880,
 881, 883, 884, 886, 893, 894, 896, 901,
 902, 904, 905, 911, 915, 916, 917, 920,
 921, 922, 924, 926, 928, 930, 931, 932,
 934, 935, 938, 940, 942, 943, 947, 951,
 952, 959, 969, 970, 971, 973, 974, 979,
 980, 981, 982, 983, 984, 985, 986, 990,
 992, 994, 996, 997, 998, 1000, 1005, 1006,
 1007, 1008, 1009, 1010, 1011, 1012, 1018,
 1021, 1024, 1032, 1035, 1036, 1037, 1038,
 1043, 1044, 1047, 1048, 1049, 1052, 1053,
 1054, 1055, 1056, 1058, 1061, 1062, 1063,
 1064, 1065, 1066, 1067, 1068, 1069, 1070,

1074, 1075, 1076, 1077, 1079, 1080, 1083,
1084, 1085, 1088, 1090, 1092, 1093, 1102,
1103, 1105, 1111, 1112, 1113, 1114, 1119,
1120, 1121, 1122, 1127, 1129, 1131, 1132,
1134, 1135, 1141, 1143, 1144, 1147, 1150,
1153, 1154, 1158, 1162, 1164, 1168, 1174,
1177, 1181, 1189, 1190, 1191, 1192, 1194,
1195, 1196, 1198, 1199

lower-case-p, 134, 135, 228, 229

M

macroexpand-1, 107

macrolet, 516

make-hash-table, 186, 212, 223, 532,
741, 742, 756, 804

make-list, 220, 679, 853, 1070, 1149, 1150

make-string, 241, 242, 409

make-string-output-stream, 241, 242

map, 65, 66, 108, 123, 124, 126, 141, 147,

149, 150, 181, 182, 203, 213, 228, 229,

234, 235, 236, 246, 306, 441, 442, 443,

504, 505, 506, 507, 529, 530, 549, 582,

602, 628, 737, 828, 829, 831, 866, 867,

1020, 1097, 1098, 1099

mapcon, 206, 207, 222, 223, 227, 228, 229,

268, 269, 272, 275, 276, 280, 281, 294,

306, 341, 342, 373, 416, 443, 448, 453,

483, 487, 495, 496, 498, 499, 520, 529,

530, 532, 535, 549, 554, 567, 597, 612,

613, 614, 619, 632, 644, 647, 648, 649,

652, 658, 671, 694, 696, 697, 740, 741,

766, 769, 770, 772, 773, 777, 778, 779,

782, 809, 886, 887, 922, 925, 947, 990,

1018, 1030, 1032, 1034, 1041, 1051, 1052,

1058, 1059, 1064, 1067, 1072, 1079, 1080,

1081, 1086, 1166, 1169, 1185, 1186, 1187

mapcar, 66, 81, 82, 91, 94, 95, 96, 97, 98,

102, 109, 110, 111, 112, 113, 114, 126,

139, 144, 145, 147, 148, 149, 150, 152,

153, 154, 155, 163, 164, 168, 169, 180,

181, 182, 188, 189, 190, 196, 199, 200,

203, 204, 205, 206, 207, 208, 213, 222,

234, 235, 236, 237, 238, 240, 241, 251,

252, 253, 254, 255, 256, 257, 258, 259,

266, 284, 286, 287, 288, 299, 303, 304,

305, 310, 311, 312, 318, 343, 347, 348,

349, 350, 355, 356, 357, 358, 365, 414,

415, 416, 424, 427, 428, 429, 430, 431,

434, 435, 436, 438, 439, 441, 443, 444,

446, 453, 454, 457, 460, 464, 465, 469,

470, 474, 476, 479, 482, 488, 503, 504,

505, 506, 507, 508, 509, 512, 515, 518,

530, 540, 541, 542, 546, 549, 550, 551,

556, 564, 574, 577, 578, 579, 580, 581,

582, 583, 587, 588, 589, 590, 594, 597,

598, 602, 605, 610, 613, 615, 616, 617,

618, 621, 623, 627, 628, 629, 630, 631,

633, 634, 637, 638, 639, 640, 641, 643,

644, 645, 648, 652, 654, 655, 659, 660,

661, 662, 664, 665, 667, 669, 670, 673,

674, 678, 679, 681, 682, 690, 692, 693,

694, 700, 705, 706, 718, 719, 731, 759,

762, 769, 770, 771, 774, 775, 785, 786,

787, 805, 806, 825, 828, 841, 842, 852,

854, 857, 863, 865, 866, 867, 893, 894,

911, 937, 938, 944, 945, 950, 951, 963,

964, 965, 968, 969, 972, 979, 980, 984,

990, 991, 1004, 1008, 1011, 1020, 1021,

1024, 1032, 1033, 1035, 1040, 1041, 1042,

1046, 1049, 1054, 1061, 1062, 1063, 1064,

1065, 1067, 1068, 1069, 1070, 1071, 1076,

1084, 1088, 1090, 1136, 1137, 1153, 1162,

1164, 1166, 1169, 1171, 1180, 1185, 1186,

1187, 1202

mapcon, 433, 574, 769, 1092, 1093

maphash, 186, 212, 223, 532, 741, 742, 756,
804

maplist, 327, 510, 537, 565, 574, 656, 801,
1032, 1063

max, 68, 96, 97, 98, 99, 165, 166, 172, 204,

205, 206, 207, 208, 209, 240, 241, 242,

243, 245, 271, 273, 292, 297, 298, 303,

307, 308, 310, 350, 365, 366, 367, 368,

369, 374, 375, 376, 377, 391, 392, 393,

394, 409, 511, 518, 528, 548, 549, 550,

551, 552, 558, 573, 574, 588, 589, 590,

595, 600, 615, 632, 633, 634, 638, 639,

640, 641, 647, 661, 662, 663, 664, 665,

667, 668, 682, 683, 684, 685, 686, 687,

688, 689, 705, 706, 708, 709, 713, 714,

715, 716, 718, 719, 727, 728, 729, 747,

749, 750, 788, 789, 790, 791, 792, 793,

794, 795, 835, 836, 840, 851, 852, 903,

1014, 1019, 1020, 1028, 1053, 1054, 1055,

1056, 1057, 1060, 1061, 1062, 1092, 1093,

1125, 1126

member, 74, 78, 91, 92, 99, 100, 102, 159,

198, 232, 266, 267, 276, 278, 297, 325,

326, 327, 328, 385, 388, 401, 402, 406,

436, 437, 440, 448, 461, 462, 472, 473,

476, 477, 490, 491, 493, 494, 495, 496,

501, 502, 508, 517, 518, 520, 521, 531,

534, 535, 567, 599, 600, 666, 675, 676,
 712, 716, 717, 732, 754, 755, 760, 761,
 831, 881, 886, 887, 889, 891, 892, 898,
 934, 939, 940, 944, 945, 963, 965, 966,
 969, 984, 1006, 1007, 1014, 1015, 1016,
 1028, 1035, 1036, 1037, 1039, 1042, 1043,
 1044, 1045, 1049, 1050, 1051, 1052, 1054,
 1057, 1058, 1059, 1062, 1063, 1075, 1078,
 1079, 1080, 1081, 1085, 1086, 1131, 1162,
 1166, 1169, 1174, 1175, 1180, 1181, 1182,
 1183, 1188, 1189, 1190, 1191
 merge, 76, 953, 1002, 1029, 1050, 1054,
 1055, 1089, 1091
 min, 68, 74, 116, 117, 130, 131, 152, 153,
 154, 155, 172, 307, 309, 312, 328, 332,
 333, 340, 341, 342, 354, 355, 357, 358,
 359, 368, 369, 374, 375, 394, 415, 416,
 511, 528, 538, 548, 549, 550, 551, 552,
 583, 588, 589, 590, 600, 615, 632, 633,
 634, 641, 669, 670, 671, 682, 705, 708,
 709, 710, 711, 763, 774, 775, 919, 930,
 931, 1014, 1047, 1125, 1126
 mismatch, 76, 1023, 1045, 1179
 mod, 122, 138, 139, 142, 143, 144, 155, 184,
 185, 215, 216, 217, 218, 220, 244, 245,
 287, 354, 361, 417, 440, 445, 507, 526,
 634, 814, 815, 823, 829, 836, 847, 848,
 875, 876, 878, 893, 894, 905, 1033, 1116,
 1203
 multiple-value-bind, 116, 124, 125,
 210, 577, 821
 multiple-value-setq, 239

N

nbutlast, 420, 882
 nconc, 93, 126, 127, 141, 158, 174, 182, 185,
 186, 194, 195, 201, 205, 206, 207, 208,
 215, 221, 222, 223, 226, 249, 256, 257,
 258, 259, 305, 313, 314, 355, 415, 416,
 417, 418, 424, 426, 432, 433, 444, 451,
 452, 468, 486, 495, 497, 498, 499, 500,
 514, 515, 516, 547, 549, 556, 557, 559,
 563, 582, 583, 589, 590, 592, 593, 599,
 600, 603, 604, 609, 613, 620, 621, 625,
 626, 633, 634, 651, 652, 656, 661, 663,
 664, 665, 669, 677, 696, 697, 703, 712,
 713, 720, 729, 730, 738, 740, 747, 748,
 751, 754, 755, 756, 760, 761, 764, 769,
 770, 790, 791, 792, 802, 803, 805, 806,
 809, 835, 843, 849, 850, 851, 862, 869,
 887, 889, 895, 898, 899, 900, 901, 911,
 947, 948, 958, 959, 961, 963, 1008, 1030,
 1031, 1032, 1034, 1042, 1043, 1045, 1050,
 1051, 1052, 1055, 1056, 1061, 1062, 1064,
 1065, 1066, 1067, 1068, 1069, 1070, 1072,
 1079, 1081, 1141, 1142, 1143, 1144, 1147,
 1149, 1150, 1151, 1167, 1170, 1182, 1190,
 1199, 1203
 nintersection, 769
 not, 58, 68, 77, 85, 97, 98, 99, 102, 103, 104,
 108, 117, 120, 125, 141, 142, 143, 145,
 153, 154, 155, 167, 168, 174, 175, 182,
 184, 186, 188, 203, 205, 206, 207, 208,
 217, 218, 220, 221, 222, 223, 227, 228,
 229, 230, 231, 233, 245, 250, 261, 271,
 274, 275, 276, 283, 285, 289, 296, 303,
 304, 305, 306, 309, 312, 324, 325, 326,
 327, 330, 334, 354, 359, 361, 368, 370,
 372, 373, 374, 375, 377, 378, 379, 387,
 388, 395, 408, 423, 432, 438, 446, 453,
 458, 459, 468, 504, 515, 516, 524, 528,
 534, 535, 551, 558, 564, 565, 576, 580,
 583, 585, 586, 587, 588, 600, 601, 602,
 606, 607, 612, 616, 617, 624, 625, 638,
 639, 640, 641, 646, 647, 648, 649, 652,
 664, 665, 705, 718, 729, 739, 740, 741,
 742, 745, 755, 761, 766, 769, 782, 783,
 794, 804, 805, 806, 824, 834, 842, 843,
 862, 863, 868, 870, 871, 873, 880, 894,
 911, 912, 913, 915, 921, 923, 924, 925,
 926, 929, 933, 934, 942, 946, 947, 959,
 969, 972, 973, 976, 984, 985, 991, 994,
 996, 999, 1001, 1003, 1004, 1006, 1007,
 1008, 1014, 1023, 1024, 1028, 1044, 1060,
 1077, 1078, 1081, 1087, 1089, 1094, 1104,
 1109, 1110, 1114, 1115, 1117, 1121, 1130,
 1133, 1135, 1136, 1148, 1154, 1162, 1165,
 1168, 1171, 1172, 1173, 1174, 1175, 1176,
 1177, 1178, 1180, 1185, 1186, 1187, 1188,
 1190, 1192, 1194, 1196, 1200, 1201, 1203,
 1240
 notany, 215, 216, 217, 218, 378
 nreconc, 591, 769
 nreverse, 110, 123, 124, 152, 179, 189,
 198, 199, 216, 218, 237, 239, 241, 242,
 327, 423, 473, 504, 514, 527, 571, 644,
 645, 651, 684, 685, 686, 687, 688, 691,
 692, 694, 696, 700, 707, 712, 717, 722,
 723, 726, 732, 733, 743, 747, 748, 764,
 765, 769, 847, 848, 867, 868, 872, 876,
 882, 952, 984, 985, 994, 995, 996, 999,
 1000, 1030, 1031, 1036, 1042, 1043, 1044,

1045, 1050, 1051, 1073, 1074, 1148, 1154,
 1174, 1175, 1176, 1190, 1197, 1198
 nset-difference, 225, 636, 769, 1047
 nset-exclusive-or, 769
 nsubst, 769
 nsubst-if, 769
 nsubst-if-not, 769
 nsubstitute, 1134, 1138
 nth, 67, 74, 263, 291, 305, 308, 323, 357,
 358, 430, 431, 436, 455, 456, 486, 549,
 550, 574, 594, 595, 668, 677, 707, 708,
 709, 805, 828, 878, 884, 885, 898, 899,
 900, 901, 902, 922, 923, 925, 932, 952,
 981, 1003, 1004, 1028, 1060, 1116, 1155,
 1156, 1159, 1163, 1171, 1172, 1180, 1188,
 1189, 1203
 nthcdr, 67, 75, 110, 171, 188, 189, 190, 251,
 252, 254, 255, 256, 257, 376, 378, 416,
 417, 418, 441, 544, 545, 583, 678, 697,
 714, 715, 732, 798, 885, 897, 898, 899,
 955, 986, 1072, 1203
 nth-value, 357, 358, 1116
 null, 72, 78, 80, 94, 95, 99, 102, 112, 113,
 123, 126, 135, 141, 184, 198, 199, 202,
 203, 204, 205, 206, 207, 208, 244, 262,
 265, 266, 267, 268, 269, 270, 271, 272,
 273, 274, 275, 276, 277, 278, 279, 280,
 281, 282, 283, 284, 285, 287, 288, 290,
 291, 292, 293, 295, 296, 297, 298, 300,
 301, 302, 303, 304, 305, 306, 309, 310,
 311, 312, 313, 315, 316, 318, 319, 320,
 321, 322, 324, 325, 326, 327, 328, 330,
 331, 332, 333, 335, 336, 338, 339, 340,
 342, 343, 351, 354, 357, 358, 359, 360,
 361, 362, 363, 364, 365, 366, 367, 368,
 371, 372, 373, 375, 376, 377, 379, 380,
 381, 385, 386, 387, 388, 389, 390, 391,
 392, 393, 394, 396, 398, 399, 401, 406,
 408, 410, 411, 413, 415, 417, 420, 421,
 425, 426, 427, 432, 433, 434, 437, 438,
 440, 441, 442, 443, 444, 447, 448, 449,
 450, 451, 452, 453, 454, 455, 458, 459,
 460, 461, 462, 463, 464, 466, 467, 469,
 470, 471, 472, 473, 474, 475, 477, 478,
 479, 481, 482, 483, 485, 486, 487, 488,
 489, 490, 491, 492, 493, 494, 495, 496,
 497, 498, 499, 501, 502, 505, 506, 508,
 509, 510, 511, 512, 513, 514, 517, 518,
 520, 521, 522, 523, 524, 525, 526, 528,
 530, 531, 532, 533, 534, 535, 536, 537,
 538, 539, 543, 546, 547, 548, 550, 551,
 554, 555, 558, 559, 560, 561, 562, 564,
 566, 567, 568, 569, 570, 571, 572, 573,
 574, 577, 579, 580, 581, 582, 583, 584,
 585, 586, 587, 588, 589, 596, 598, 599,
 601, 602, 605, 606, 607, 609, 610, 612,
 617, 618, 619, 620, 622, 623, 624, 625,
 626, 629, 631, 632, 635, 636, 639, 640,
 641, 642, 643, 644, 645, 646, 647, 648,
 649, 650, 651, 652, 653, 654, 655, 656,
 657, 659, 660, 661, 662, 663, 665, 666,
 667, 669, 670, 671, 672, 674, 675, 676,
 680, 681, 685, 689, 690, 691, 692, 695,
 696, 698, 699, 700, 703, 710, 711, 712,
 713, 714, 715, 716, 720, 722, 723, 724,
 725, 726, 727, 728, 732, 737, 738, 739,
 743, 746, 747, 748, 749, 750, 751, 753,
 754, 757, 758, 761, 763, 764, 765, 768,
 772, 774, 775, 776, 777, 778, 781, 782,
 784, 785, 786, 787, 789, 790, 791, 792,
 794, 795, 796, 797, 800, 805, 806, 807,
 808, 813, 831, 855, 857, 858, 864, 880,
 881, 882, 883, 884, 885, 886, 887, 888,
 889, 890, 891, 892, 893, 895, 896, 897,
 901, 903, 904, 910, 911, 912, 913, 914,
 915, 916, 917, 919, 921, 922, 923, 924,
 926, 927, 928, 930, 931, 932, 933, 938,
 939, 941, 942, 943, 944, 945, 946, 947,
 948, 949, 950, 951, 953, 954, 955, 956,
 957, 959, 960, 961, 962, 963, 964, 965,
 967, 968, 969, 971, 972, 973, 974, 975,
 976, 978, 981, 982, 983, 984, 985, 987,
 988, 989, 992, 993, 994, 995, 996, 998,
 999, 1000, 1002, 1005, 1008, 1009, 1014,
 1015, 1016, 1017, 1018, 1019, 1021, 1022,
 1024, 1025, 1026, 1028, 1029, 1030, 1031,
 1033, 1034, 1035, 1036, 1037, 1038, 1042,
 1044, 1045, 1048, 1049, 1050, 1051, 1053,
 1054, 1055, 1057, 1058, 1059, 1060, 1062,
 1063, 1066, 1067, 1069, 1071, 1072, 1073,
 1074, 1075, 1077, 1078, 1080, 1081, 1082,
 1083, 1085, 1086, 1089, 1090, 1091, 1094,
 1096, 1097, 1098, 1109, 1129, 1130, 1134,
 1136, 1137, 1138, 1139, 1140, 1141, 1142,
 1144, 1145, 1146, 1147, 1148, 1149, 1150,
 1151, 1154, 1157, 1160, 1162, 1164, 1166,
 1167, 1169, 1172, 1173, 1174, 1175, 1176,
 1180, 1182, 1183, 1184, 1186, 1187, 1188,
 1189, 1190, 1191, 1197, 1198, 1201, 1202
 numberp, 72, 120, 147, 148, 205, 206, 207,
 208, 263, 264, 270, 275, 276, 289, 290,
 302, 303, 304, 305, 306, 311, 318, 319,
 324, 325, 338, 342, 343, 344, 352, 353,

369, 370, 371, 374, 387, 389, 397, 399,
400, 419, 427, 428, 434, 475, 478, 481,
496, 497, 501, 512, 538, 543, 553, 561,
562, 576, 577, 578, 596, 597, 598, 602,
639, 640, 646, 647, 648, 649, 650, 702,
706, 712, 747, 758, 759, 761, 762, 763,
768, 769, 782, 783, 790, 791, 792, 793,
794, 901, 911, 912, 917, 918, 951, 956,
975, 976, 984, 1005, 1006, 1071, 1072,
1085, 1135, 1136, 1138, 1139, 1154

nunion, 769

O

oddp, 67, 73, 76, 110, 169, 170, 195, 226,
265, 270, 282, 301, 328, 329, 330, 331,
334, 335, 336, 364, 365, 368, 460, 461,
524, 525, 526, 543, 556, 601, 621, 651,
652, 677, 678, 758, 759, 763, 803, 919,
930, 931, 982, 983, 1041, 1118, 1162,
1173, 1174, 1175, 1176, 1197, 1198
or, 63, 65, 67, 76, 77, 82, 83, 88, 89, 90, 91,
93, 97, 98, 99, 102, 106, 107, 108, 109,
111, 112, 115, 120, 123, 124, 125, 126,
128, 131, 132, 134, 135, 138, 139, 140,
142, 143, 145, 147, 148, 149, 150, 151,
152, 153, 154, 155, 158, 162, 163, 166,
167, 168, 169, 171, 172, 174, 175, 177,
179, 180, 181, 182, 184, 185, 186, 189,
190, 191, 192, 193, 194, 196, 198, 199,
200, 201, 202, 205, 208, 209, 210, 211,
213, 214, 215, 216, 217, 218, 219, 220,
221, 222, 223, 226, 227, 228, 229, 230,
231, 232, 233, 234, 235, 236, 237, 239,
240, 241, 242, 243, 244, 245, 246, 247,
249, 251, 252, 253, 254, 255, 256, 257,
258, 259, 260, 263, 265, 266, 267, 268,
269, 272, 275, 280, 281, 282, 289, 291,
294, 295, 296, 301, 302, 304, 306, 307,
309, 311, 312, 313, 314, 315, 316, 319,
321, 322, 323, 326, 327, 330, 331, 332,
333, 334, 335, 339, 340, 341, 343, 344,
345, 346, 350, 352, 353, 354, 355, 356,
359, 360, 361, 362, 364, 365, 367, 369,
370, 371, 372, 373, 374, 377, 380, 381,
382, 383, 384, 385, 387, 388, 389, 390,
391, 393, 394, 396, 397, 398, 399, 400,
401, 402, 403, 404, 405, 407, 409, 410,
411, 418, 419, 422, 424, 425, 426, 428,
429, 433, 434, 436, 438, 439, 440, 442,
444, 445, 446, 448, 449, 451, 457, 458,
459, 460, 463, 464, 465, 467, 470, 471,

474, 475, 480, 481, 483, 484, 485, 487,
489, 490, 491, 492, 495, 497, 498, 499,
500, 515, 519, 520, 521, 522, 525, 526,
529, 530, 532, 534, 535, 536, 537, 538,
541, 542, 543, 544, 545, 546, 548, 550,
551, 552, 553, 554, 556, 557, 560, 561,
562, 566, 567, 568, 575, 576, 577, 578,
579, 582, 584, 585, 587, 589, 590, 595,
596, 597, 600, 601, 605, 606, 607, 608,
610, 611, 613, 614, 617, 619, 620, 621,
622, 624, 625, 626, 627, 629, 630, 631,
632, 633, 636, 639, 641, 642, 644, 645,
646, 647, 649, 650, 651, 652, 653, 654,
655, 656, 657, 658, 659, 662, 663, 664,
666, 667, 668, 671, 673, 674, 678, 679,
681, 685, 690, 693, 694, 695, 696, 700,
701, 702, 703, 704, 705, 706, 708, 709,
710, 712, 713, 714, 715, 717, 718, 719,
723, 724, 725, 726, 727, 729, 730, 731,
732, 733, 734, 735, 736, 737, 738, 739,
740, 741, 742, 743, 744, 745, 747, 749,
753, 756, 757, 758, 759, 762, 763, 764,
765, 767, 768, 769, 770, 771, 772, 773,
774, 775, 776, 777, 778, 779, 780, 781,
783, 784, 785, 786, 787, 788, 789, 790,
791, 794, 795, 796, 797, 798, 800, 801,
803, 805, 806, 807, 808, 809, 810, 811,
816, 820, 823, 824, 825, 827, 829, 831,
832, 833, 834, 835, 836, 839, 840, 841,
843, 845, 847, 848, 849, 850, 851, 852,
853, 854, 857, 859, 860, 861, 862, 863,
864, 868, 869, 870, 871, 872, 873, 874,
875, 876, 877, 878, 880, 881, 883, 884,
886, 889, 893, 894, 896, 901, 902, 904,
905, 911, 913, 915, 916, 917, 918, 919,
920, 921, 922, 923, 924, 925, 926, 928,
929, 930, 931, 932, 933, 934, 935, 937,
938, 940, 941, 942, 943, 947, 951, 952,
954, 959, 969, 970, 971, 973, 974, 976,
979, 980, 981, 982, 983, 984, 985, 986,
990, 992, 994, 996, 997, 998, 1000, 1001,
1005, 1006, 1007, 1008, 1009, 1010, 1011,
1012, 1013, 1015, 1016, 1017, 1018, 1021,
1022, 1024, 1025, 1026, 1027, 1032, 1035,
1036, 1041, 1042, 1043, 1044, 1046, 1047,
1048, 1049, 1050, 1051, 1052, 1053, 1054,
1055, 1056, 1058, 1059, 1061, 1062, 1063,
1064, 1065, 1066, 1067, 1068, 1069, 1070,
1073, 1075, 1076, 1077, 1079, 1080, 1081,
1083, 1084, 1085, 1087, 1088, 1090, 1092,
1093, 1102, 1103, 1105, 1108, 1109, 1110,
1111, 1113, 1114, 1115, 1117, 1118, 1119,

1120, 1121, 1122, 1124, 1125, 1127, 1129,
1130, 1131, 1132, 1134, 1135, 1141, 1143,
1144, 1147, 1153, 1154, 1155, 1157, 1158,
1160, 1162, 1164, 1165, 1168, 1174, 1175,
1176, 1177, 1180, 1181, 1182, 1183, 1185,
1186, 1187, 1188, 1189, 1190, 1191, 1192,
1194, 1195, 1196, 1197, 1198, 1199, 1210

P

parse-integer, 64, 109, 110, 117, 119,
122, 125, 143, 150, 173, 180, 181, 182,
188, 189, 190, 197, 234, 286, 318, 347,
348, 349, 350, 504, 505, 507, 578, 627,
628, 629, 821, 822, 828, 859, 860, 861,
862, 866, 867, 908
plusp, 68, 77, 105, 121, 137, 167, 178, 179,
194, 196, 219, 242, 243, 247, 259, 265,
277, 289, 290, 309, 372, 373, 374, 409,
481, 485, 486, 488, 489, 503, 504, 512,
514, 528, 539, 540, 541, 577, 578, 583,
585, 586, 587, 588, 623, 624, 625, 644,
645, 651, 669, 670, 671, 724, 736, 737,
745, 752, 753, 764, 765, 766, 767, 768,
769, 778, 779, 853, 856, 877, 879, 901,
902, 956, 964, 971, 976, 977, 995, 996,
997, 998, 1004, 1008, 1012, 1033, 1111,
1112, 1160
pop, 378
position, 67, 98, 99, 101, 102, 103, 154,
159, 225, 249, 307, 310, 312, 436, 539,
546, 549, 551, 552, 574, 595, 600, 634,
667, 668, 697, 886, 887, 897, 902, 904,
922, 1028, 1072, 1163, 1166, 1169, 1180,
1188, 1189
position-if, 67, 98, 99, 154, 159, 539,
1072
position-if-not, 98, 99, 154
prin1, 150, 183, 236
prin1-to-string, 150, 236
princ, 172, 247, 790, 839, 969, 970
print, 97, 98, 99, 114, 126, 127, 153, 154,
155, 161, 162, 186, 192, 193, 231, 251,
252, 253, 254, 255, 256, 257, 258, 259,
814, 833, 851, 898, 899, 971, 1119, 1156
progl, 327
progn, 194, 594, 595, 708, 709, 754, 760,
790, 793, 794, 795, 1081, 1172, 1173
push, 123, 124, 178, 179, 186, 212, 222, 223,
239, 241, 242, 266, 327, 384, 423, 425,
426, 427, 434, 473, 493, 494, 496, 513,

514, 532, 559, 575, 576, 597, 598, 642,
684, 685, 700, 717, 741, 742, 756, 804,
835, 952, 1030, 1031, 1165, 1169
pushnew, 199, 494, 530, 717

R

random, 83, 192, 193, 194, 226, 350, 351,
455, 456, 544, 545, 551, 552, 669, 683,
684, 685, 686, 687, 688, 846, 854, 855,
856, 872, 873, 1037, 1038, 1101, 1103,
1122, 1123
rationalp, 119, 120
read, 63, 66, 91, 94, 95, 96, 97, 98, 111, 112,
113, 114, 117, 119, 120, 122, 125, 126,
140, 141, 142, 144, 145, 147, 148, 152,
153, 154, 161, 162, 164, 172, 173, 174,
184, 186, 187, 188, 194, 195, 197, 203,
204, 205, 206, 207, 208, 209, 210, 211,
212, 213, 221, 222, 223, 224, 230, 231,
245, 250, 286, 287, 299, 303, 344, 345,
346, 403, 404, 405, 406, 438, 467, 468,
506, 519, 521, 522, 523, 530, 536, 549,
552, 553, 581, 583, 602, 603, 604, 616,
704, 733, 734, 735, 756, 814, 825, 827,
839, 840, 841, 843, 845, 849, 850, 851,
862, 863, 874, 893, 894, 895, 913, 932,
969, 990, 991, 992, 1006, 1007, 1012,
1087, 1106, 1114, 1156, 1159, 1208, 1210
read-from-string, 63, 66, 91, 94, 95, 96,
97, 98, 111, 112, 113, 114, 120, 125, 126,
140, 142, 144, 145, 147, 148, 152, 153,
154, 161, 162, 164, 174, 184, 186, 194,
195, 203, 204, 205, 206, 207, 208, 209,
213, 221, 222, 223, 224, 245, 286, 287,
303, 404, 438, 506, 549, 581, 583, 602,
603, 604, 814, 825, 839, 843, 845, 849,
850, 851, 862, 863, 874, 894, 932, 969,
970, 971, 1007, 1012, 1106
read-line, 114, 117, 119, 122, 141, 142,
161, 162, 164, 173, 174, 184, 186, 194,
195, 205, 206, 207, 208, 210, 211, 221,
222, 223, 224, 230, 231, 404, 519, 583,
603, 604, 814, 840, 841, 843, 849, 850,
851, 895, 932, 1006, 1007, 1012
realp, 289, 501, 503, 504, 751, 756, 1118
reduce, 68, 82, 93, 108, 123, 124, 139, 145,
147, 170, 172, 205, 206, 207, 208, 234,
235, 236, 237, 238, 240, 241, 262, 264,
271, 275, 276, 278, 284, 285, 286, 288,
289, 290, 291, 294, 302, 305, 306, 309,

310, 312, 313, 314, 315, 317, 319, 328,
 330, 331, 339, 340, 342, 347, 348, 349,
 350, 352, 353, 354, 355, 356, 357, 359,
 361, 368, 369, 372, 373, 389, 393, 394,
 400, 402, 403, 404, 405, 409, 416, 418,
 443, 449, 465, 466, 469, 472, 503, 508,
 510, 511, 537, 544, 552, 558, 571, 586,
 588, 589, 595, 597, 598, 603, 604, 613,
 624, 625, 632, 633, 634, 637, 638, 639,
 640, 641, 648, 649, 652, 654, 655, 656,
 664, 665, 667, 668, 669, 670, 671, 672,
 673, 679, 680, 681, 682, 702, 703, 705,
 706, 709, 710, 711, 714, 731, 753, 758,
 759, 763, 764, 766, 767, 769, 774, 775,
 782, 783, 784, 785, 786, 787, 788, 789,
 794, 795, 798, 801, 818, 821, 822, 823,
 825, 835, 840, 891, 908, 911, 912, 919,
 927, 930, 931, 976, 983, 1008, 1010, 1019,
 1020, 1021, 1023, 1024, 1028, 1055, 1061,
 1062, 1067, 1068, 1069, 1088, 1093, 1102,
 1105, 1106, 1112, 1122, 1129, 1136, 1137,
 1191, 1192, 1200, 1203
 rem, 110, 115, 116, 120, 121, 151, 155, 157,
 158, 165, 166, 170, 171, 172, 181, 185,
 188, 190, 196, 198, 199, 209, 210, 215,
 216, 218, 232, 237, 238, 239, 240, 241,
 243, 244, 336, 337, 340, 379, 380, 419,
 561, 563, 564, 579, 586, 587, 594, 629,
 630, 667, 668, 694, 757, 758, 814, 822,
 823, 824, 828, 851, 852, 864, 865, 869,
 871, 877, 898, 899, 900, 901, 905, 924,
 944, 945, 973, 974, 989, 990, 1007, 1074,
 1102, 1103, 1104, 1105, 1113, 1114, 1119,
 1120, 1121
 remove, 68, 77, 95, 96, 97, 100, 112, 113,
 114, 125, 126, 153, 185, 199, 202, 203,
 205, 206, 207, 208, 211, 213, 214, 217,
 218, 224, 228, 229, 230, 231, 243, 245,
 271, 272, 275, 276, 278, 283, 285, 303,
 304, 309, 320, 330, 347, 348, 349, 352,
 359, 361, 368, 370, 371, 372, 373, 387,
 388, 424, 425, 437, 438, 444, 446, 453,
 454, 455, 456, 459, 461, 462, 467, 482,
 483, 484, 493, 494, 495, 496, 497, 501,
 502, 509, 511, 512, 515, 516, 521, 522,
 523, 524, 551, 558, 561, 562, 564, 567,
 568, 576, 579, 580, 583, 586, 587, 588,
 592, 593, 597, 598, 599, 601, 602, 606,
 607, 613, 614, 624, 625, 629, 630, 631,
 638, 639, 640, 641, 643, 644, 645, 646,
 647, 648, 649, 652, 658, 664, 665, 666,
 679, 690, 691, 693, 694, 703, 714, 716,
 717, 718, 719, 729, 731, 733, 739, 740,
 741, 742, 745, 753, 764, 766, 771, 772,
 773, 776, 777, 778, 782, 783, 794, 798,
 805, 806, 807, 825, 843, 860, 861, 863,
 864, 865, 866, 867, 868, 869, 911, 912,
 919, 930, 931, 934, 935, 942, 943, 944,
 946, 947, 956, 961, 962, 963, 965, 969,
 972, 974, 976, 977, 981, 983, 984, 985,
 990, 991, 994, 996, 999, 1000, 1001, 1003,
 1004, 1008, 1041, 1043, 1044, 1046, 1048,
 1052, 1055, 1056, 1058, 1059, 1069, 1070,
 1074, 1075, 1077, 1078, 1081, 1092, 1093,
 1110, 1136, 1144, 1148, 1154, 1156, 1173,
 1176, 1177, 1181, 1182, 1185, 1186, 1187,
 1190, 1202, 1203
 remove-duplicates, 68, 100, 114, 185,
 199, 202, 203, 211, 213, 214, 245, 278,
 352, 359, 424, 425, 437, 446, 483, 484,
 493, 496, 497, 502, 521, 567, 568, 597,
 598, 613, 614, 630, 631, 638, 639, 643,
 644, 645, 658, 679, 690, 691, 693, 694,
 703, 714, 716, 717, 718, 719, 733, 739,
 740, 741, 764, 771, 772, 773, 776, 777,
 778, 806, 807, 843, 863, 864, 865, 866,
 867, 868, 869, 942, 943, 944, 965, 969,
 977, 981, 983, 984, 1043, 1052, 1055,
 1056, 1058, 1059, 1190
 remove-if, 68, 77, 97, 153, 199, 202, 203,
 205, 206, 207, 208, 217, 218, 228, 229,
 230, 231, 271, 272, 275, 276, 283, 285,
 303, 304, 309, 320, 330, 361, 368, 370,
 371, 372, 373, 387, 438, 446, 453, 454,
 459, 461, 467, 495, 515, 516, 524, 551,
 558, 561, 562, 564, 576, 583, 586, 587,
 588, 601, 602, 606, 607, 624, 625, 629,
 630, 638, 639, 640, 641, 646, 647, 648,
 649, 652, 658, 664, 665, 691, 703, 718,
 729, 739, 740, 741, 742, 745, 753, 766,
 782, 783, 794, 798, 860, 861, 868, 911,
 912, 919, 930, 931, 934, 942, 946, 947,
 961, 962, 963, 972, 974, 976, 977, 985,
 991, 994, 996, 999, 1000, 1003, 1004,
 1008, 1041, 1044, 1046, 1074, 1077, 1078,
 1081, 1092, 1093, 1136, 1148, 1154, 1156,
 1173, 1176, 1177, 1181, 1185, 1186, 1187,
 1202, 1203
 remove-if-not, 68, 97, 153, 203, 205,
 206, 207, 208, 217, 218, 228, 229, 230,
 231, 271, 272, 275, 276, 283, 285, 303,
 304, 309, 330, 361, 368, 370, 372, 373,
 387, 446, 453, 515, 516, 524, 551, 558,
 564, 576, 583, 586, 587, 588, 601, 602,

606, 607, 624, 625, 638, 639, 640, 641, 646, 647, 648, 649, 652, 664, 665, 718, 729, 739, 740, 741, 742, 745, 766, 782, 783, 794, 868, 911, 912, 934, 942, 946, 947, 972, 976, 985, 991, 994, 996, 999, 1003, 1004, 1008, 1044, 1074, 1077, 1078, 1081, 1136, 1148, 1154, 1173, 1176, 1177, 1203

replace, 56, 122, 146, 175, 815, 1096, 1097, 1113, 1115, 1137, 1138, 1180, 1181, 1184, 1185, 1188, 1189

rest, 59, 70, 166, 177, 200, 201, 230, 262, 284, 287, 309, 339, 354, 383, 409, 423, 457, 507, 556, 615, 636, 637, 756, 816, 891, 989, 990, 1004, 1062, 1088, 1106, 1117, 1125, 1190, 1191, 1192, 1202, 1203

return, 83, 106, 110, 126, 132, 139, 140, 152, 162, 166, 169, 171, 185, 186, 189, 201, 216, 218, 223, 237, 238, 239, 301, 302, 311, 314, 315, 335, 359, 377, 378, 382, 384, 409, 410, 460, 499, 525, 541, 548, 575, 576, 621, 625, 626, 646, 651, 655, 656, 659, 663, 673, 686, 687, 688, 715, 717, 730, 745, 756, 763, 765, 778, 779, 780, 783, 785, 787, 788, 789, 814, 844, 846, 867, 893, 901, 902, 904, 905, 917, 928, 934, 986, 1012, 1048, 1074, 1089, 1112, 1178, 1196

revappend, 75, 255, 256, 257

reverse, 58, 68, 75, 81, 103, 111, 112, 116, 117, 124, 125, 126, 127, 141, 143, 144, 145, 159, 178, 179, 183, 185, 203, 210, 249, 251, 252, 253, 254, 255, 256, 257, 258, 259, 274, 305, 311, 318, 337, 377, 378, 408, 420, 421, 425, 426, 431, 432, 433, 434, 437, 449, 450, 451, 460, 461, 463, 464, 465, 466, 471, 472, 473, 475, 482, 483, 485, 490, 491, 492, 503, 514, 524, 527, 544, 547, 548, 558, 559, 565, 575, 576, 577, 578, 580, 581, 582, 583, 584, 585, 592, 593, 596, 598, 599, 600, 601, 602, 616, 617, 625, 626, 629, 630, 635, 642, 646, 651, 657, 659, 660, 664, 666, 673, 674, 676, 677, 678, 681, 700, 701, 721, 725, 728, 729, 731, 732, 744, 746, 747, 750, 751, 773, 774, 775, 776, 777, 778, 781, 782, 798, 800, 801, 804, 805, 835, 841, 850, 851, 858, 859, 860, 861, 862, 877, 885, 901, 922, 932, 944, 945, 946, 965, 966, 967, 968, 987, 988, 992, 993, 994, 1008, 1066, 1072, 1090, 1131, 1136, 1162, 1179

rotateref, 486, 549, 594, 595, 668, 707, 1125, 1155, 1156, 1159

row-major-aref, 740, 741, 920, 931, 1159

rplacd, 754, 760, 1081

S

search, 66, 68, 102, 103, 115, 146, 813, 815, 841, 858, 859, 1096, 1109, 1113

second, 157, 220, 338, 375, 376, 492, 550, 583, 584, 618, 619, 631, 650, 705, 706, 714, 715, 785, 804, 923, 926, 1017, 1018, 1023, 1024, 1138

set-difference, 74, 76, 168, 440, 448, 487, 769, 862, 863, 1035, 1036, 1042, 1046, 1054, 1078, 1079, 1086, 1094, 1173, 1176, 1191, 1192

set-exclusive-or, 76, 769, 872, 1041, 1042, 1043, 1044, 1050, 1051, 1052

setf, 110, 141, 142, 152, 174, 189, 194, 195, 205, 206, 207, 208, 209, 210, 211, 216, 218, 226, 230, 231, 237, 238, 239, 291, 300, 314, 315, 344, 345, 346, 377, 382, 394, 409, 410, 427, 430, 467, 536, 550, 552, 575, 576, 591, 592, 593, 594, 616, 677, 685, 686, 687, 688, 694, 700, 704, 708, 709, 715, 733, 734, 736, 769, 770, 771, 780, 805, 835, 836, 841, 843, 844, 849, 850, 851, 867, 874, 878, 949, 971, 991, 992, 1001, 1006, 1007, 1047, 1074, 1088, 1111, 1112, 1159, 1172, 1173, 1192

setq, 161, 162, 183, 185, 221, 222, 223, 239, 291, 299, 300, 416, 423, 424, 430, 466, 475, 498, 522, 523, 531, 603, 604, 814, 949, 1182, 1183

seventh, 78, 264, 289

signum, 248

some, 76, 92, 131, 167, 168, 210, 230, 233, 247, 270, 321, 328, 358, 359, 372, 374, 380, 387, 391, 395, 416, 428, 449, 468, 538, 539, 561, 562, 579, 580, 712, 713, 738, 739, 803, 833, 845, 871, 873, 875, 881, 897, 901, 917, 918, 920, 929, 933, 1014, 1087, 1110, 1115, 1163, 1190, 1191

sort, 66, 77, 96, 97, 98, 101, 102, 111, 116, 117, 152, 153, 154, 163, 168, 186, 198, 199, 203, 222, 223, 230, 231, 266, 271, 299, 300, 347, 348, 349, 350, 357, 358, 376, 395, 419, 427, 443, 484, 492, 493, 494, 504, 505, 508, 509, 511, 512, 517, 527, 530, 533, 534, 538, 540, 549, 621,

645, 647, 648, 652, 653, 654, 658, 677,
 679, 703, 714, 715, 731, 732, 763, 776,
 777, 778, 799, 839, 952, 1001, 1002, 1028,
 1029, 1041, 1042, 1056, 1061, 1062, 1063,
 1064, 1065, 1066, 1071, 1073, 1077, 1078,
 1089, 1115, 1138
 sqrt, 130, 131, 220, 439, 517, 699, 700, 847,
 848, 893, 1021, 1109, 1110
 stable-sort, 1028
 string/=: 123, 124
 string<, 76, 223, 358, 419, 427, 484, 494,
 1064, 1065
 string=: 65, 90, 91, 337, 502, 764, 835,
 1071, 1091, 1097
 string>, 1071
 string-capitalize, 103, 117, 479
 string-downcase, 103, 221, 222, 438,
 564, 565
 string-equal, 65, 173, 597, 598, 739, 740,
 741, 864, 1036, 1098, 1099
 string-greaterp, 222
 string-lessp, 222, 358
 stringp, 391, 399, 400, 408, 410, 496, 497,
 512, 513, 638, 639, 739, 740, 741, 742,
 756, 1124, 1125
 string-right-trim, 94, 95, 126, 127,
 144, 145, 163, 175, 581, 582
 string-trim, 120, 126
 string-upcase, 103, 476
 subseq, 67, 75, 94, 98, 99, 101, 102, 103,
 110, 113, 114, 115, 134, 135, 154, 158,
 159, 168, 171, 173, 188, 189, 190, 225,
 245, 249, 311, 319, 355, 378, 409, 417,
 418, 422, 441, 456, 457, 538, 545, 563,
 594, 600, 677, 678, 697, 708, 709, 720,
 721, 731, 732, 748, 771, 797, 810, 813,
 815, 821, 822, 823, 858, 859, 898, 899,
 900, 901, 908, 912, 913, 940, 946, 955,
 981, 986, 1002, 1045, 1071, 1072, 1096,
 1109, 1129, 1145, 1146, 1165, 1166, 1167,
 1169, 1203
 subsetp, 443, 444, 579, 580, 1013, 1046
 subst, 396, 769, 807
 substitute, 68, 98, 154, 159, 470, 474,
 528, 540, 603, 670, 723, 763, 954, 1105,
 1106, 1134, 1148, 1149, 1162
 substitute-if, 98, 154, 159, 470, 474,
 540, 603, 670, 723, 763
 substitute-if-not, 98, 154
 svref, 1047

symbol-function, 874
 symbol-name, 61, 541, 542
 symbolp, 72, 269, 303, 351, 369, 370, 371,
 398, 399, 400, 496, 497, 510, 541, 542,
 543, 577, 626, 627, 646, 647, 756, 757,
 773, 930, 931, 1124, 1125

T

terpri, 756
 third, 220, 264, 289, 375, 376, 416, 417,
 418, 419, 561, 698, 699, 714, 715, 785,
 799, 926, 1011, 1113, 1114, 1192, 1193
 time, 105, 106, 131, 132, 138, 139, 157, 166,
 167, 168, 177, 178, 179, 180, 182, 188,
 189, 220, 234, 235, 236, 237, 238, 239,
 250, 350, 351, 369, 381, 382, 383, 384,
 399, 401, 408, 409, 410, 423, 424, 449,
 499, 500, 518, 527, 533, 534, 544, 545,
 551, 552, 581, 582, 683, 684, 685, 686,
 687, 688, 689, 699, 729, 747, 801, 807,
 808, 824, 832, 833, 836, 847, 871, 905,
 929, 942, 943, 944, 1013, 1020, 1028,
 1037, 1039, 1081, 1095, 1096, 1100, 1110,
 1115, 1116, 1130, 1132, 1194
 trace, 104, 105, 106, 107
 truncate, 110, 115, 116, 120, 121, 124,
 125, 151, 152, 157, 158, 165, 166, 170,
 171, 181, 188, 189, 190, 196, 198, 199,
 210, 216, 218, 237, 238, 243, 244, 311,
 319, 320, 336, 337, 577, 583, 594, 810,
 821, 822, 823, 824, 828, 864, 865, 867,
 869, 1074, 1116, 1120, 1121
 typecase, 207, 389, 397, 400, 498, 740,
 756, 762, 772, 777, 886, 1052, 1059, 1079,
 1080, 1086, 1158

U

union, 75, 440, 1029, 1037, 1038, 1039
 unless, 126, 129, 155, 172, 175, 178, 193,
 210, 223, 311, 326, 327, 359, 378, 419,
 445, 473, 561, 567, 625, 650, 655, 658,
 678, 717, 724, 730, 743, 748, 753, 794,
 795, 797, 805, 817, 823, 869, 877, 915,
 926, 928, 945, 959, 969, 974, 982, 983,
 990, 994, 997, 1000, 1007, 1010, 1036,
 1043, 1047, 1052, 1054, 1074, 1075, 1079,
 1083, 1105, 1144, 1154, 1174, 1190, 1191
 untrace, 105, 106, 107

V

values, 83, 110, 152, 189, 194, 195, 201,
 205, 206, 207, 208, 209, 216, 218, 226,
 237, 275, 276, 334, 335, 460, 495, 496,
 497, 525, 541, 548, 551, 555, 574, 576,
 584, 585, 586, 588, 603, 613, 614, 632,
 633, 634, 646, 647, 649, 651, 655, 659,
 668, 685, 706, 742, 745, 752, 753, 778,
 779, 787, 788, 789, 817, 852, 855, 856,
 858, 859, 867, 1074, 1112, 1155, 1182
 vector-push-extend, 384

W

when, 71, 77, 85, 91, 93, 94, 98, 99, 100, 111,
 112, 115, 120, 126, 127, 129, 131, 134,
 135, 145, 147, 154, 161, 162, 166, 167,
 169, 170, 172, 177, 178, 179, 180, 182,
 186, 188, 189, 190, 192, 194, 196, 197,
 198, 199, 201, 202, 208, 209, 213, 215,
 217, 219, 221, 222, 225, 226, 227, 228,
 229, 232, 233, 234, 235, 236, 237, 238,
 239, 240, 241, 242, 243, 244, 245, 249,
 251, 253, 254, 255, 256, 257, 258, 259,
 263, 265, 267, 268, 269, 273, 281, 282,
 289, 290, 299, 301, 302, 304, 306, 307,
 311, 315, 316, 319, 321, 322, 323, 326,
 327, 328, 330, 340, 341, 342, 343, 344,
 345, 346, 347, 348, 349, 352, 359, 360,
 361, 362, 365, 368, 369, 374, 380, 381,
 383, 385, 386, 387, 388, 389, 390, 391,
 393, 396, 398, 401, 402, 409, 421, 423,
 425, 428, 429, 433, 436, 437, 438, 439,
 442, 444, 445, 447, 449, 450, 451, 452,
 453, 454, 455, 456, 457, 459, 462, 463,
 464, 467, 468, 470, 475, 476, 478, 480,
 481, 484, 488, 489, 495, 510, 511, 512,
 515, 519, 521, 522, 524, 525, 529, 530,
 536, 537, 539, 540, 541, 543, 544, 546,
 547, 549, 550, 552, 553, 558, 559, 560,
 561, 563, 564, 565, 566, 571, 572, 573,
 574, 575, 577, 579, 580, 587, 589, 591,
 592, 593, 594, 595, 596, 601, 605, 606,
 607, 608, 609, 610, 611, 612, 615, 616,
 617, 618, 619, 621, 622, 623, 624, 625,
 626, 635, 636, 637, 641, 642, 645, 646,
 648, 649, 654, 655, 656, 657, 662, 666,
 667, 669, 671, 672, 674, 675, 677, 678,
 681, 682, 683, 685, 687, 688, 689, 690,
 692, 693, 694, 695, 696, 697, 699, 703,
 704, 705, 706, 707, 708, 709, 711, 713,

714, 716, 718, 719, 720, 721, 722, 723,
 724, 725, 726, 727, 729, 730, 731, 733,
 734, 735, 736, 737, 738, 739, 740, 741,
 742, 744, 746, 747, 748, 749, 753, 758,
 759, 761, 764, 766, 767, 768, 769, 773,
 775, 780, 781, 782, 783, 784, 786, 787,
 788, 789, 790, 791, 793, 794, 795, 797,
 798, 800, 803, 806, 807, 808, 814, 822,
 824, 829, 831, 833, 835, 836, 847, 848,
 849, 851, 852, 853, 855, 856, 857, 860,
 861, 862, 863, 864, 865, 866, 867, 868,
 870, 871, 872, 875, 877, 879, 882, 883,
 884, 886, 887, 888, 890, 891, 892, 893,
 894, 896, 901, 902, 904, 905, 910, 917,
 918, 919, 921, 923, 926, 928, 929, 930,
 931, 933, 934, 935, 937, 938, 939, 942,
 943, 946, 947, 948, 950, 951, 952, 956,
 959, 960, 963, 964, 973, 974, 975, 976,
 977, 978, 979, 980, 981, 982, 984, 985,
 986, 988, 989, 990, 991, 992, 993, 994,
 995, 996, 997, 998, 1001, 1003, 1004,
 1005, 1006, 1007, 1008, 1009, 1011, 1013,
 1016, 1017, 1022, 1023, 1025, 1029, 1031,
 1032, 1034, 1035, 1042, 1044, 1046, 1047,
 1048, 1053, 1056, 1060, 1061, 1062, 1063,
 1068, 1075, 1076, 1077, 1078, 1082, 1083,
 1084, 1087, 1088, 1089, 1090, 1093, 1094,
 1098, 1099, 1101, 1102, 1103, 1105, 1110,
 1111, 1112, 1113, 1116, 1119, 1120, 1121,
 1122, 1127, 1130, 1131, 1133, 1134, 1135,
 1139, 1140, 1141, 1142, 1143, 1144, 1145,
 1146, 1147, 1148, 1149, 1150, 1152, 1153,
 1154, 1157, 1158, 1161, 1163, 1164, 1165,
 1166, 1168, 1169, 1170, 1174, 1175, 1176,
 1177, 1181, 1184, 1186, 1187, 1188, 1189,
 1191, 1197, 1198, 1199, 1200, 1202
 with-open-file, 114, 117, 118, 119, 122,
 141, 142, 164, 174, 184, 186, 194, 195,
 205, 206, 207, 208, 210, 211, 212, 221,
 222, 223, 224, 230, 231, 519, 603, 604,
 841, 843, 849, 850, 851, 893, 895, 1006,
 1007, 1103
 with-output-to-string, 383, 1113
 write, 65, 88, 89, 90, 104, 109, 110, 117,
 118, 119, 120, 125, 140, 143, 148, 149,
 150, 161, 163, 164, 170, 173, 175, 181,
 188, 189, 190, 197, 213, 215, 217, 234,
 235, 236, 241, 242, 250, 260, 286, 287,
 309, 318, 319, 337, 347, 348, 349, 372,
 383, 395, 399, 400, 403, 438, 504, 505,
 506, 507, 542, 543, 564, 565, 578, 602,
 627, 628, 629, 804, 821, 822, 827, 828,

832, 835, 847, 859, 860, 861, 862, 863,
864, 865, 866, 868, 874, 877, 880, 908,
918, 920, 932, 933, 970, 1027, 1087, 1110,
1113, 1115, 1130, 1137, 1155, 1171, 1194,
1206

write-char, 241, 242

write-string, 383, 1113

write-to-string, 65, 88, 89, 90, 109,
110, 117, 118, 119, 120, 125, 140, 143,
148, 149, 150, 163, 170, 173, 175, 181,
188, 189, 190, 197, 213, 215, 217, 234,
235, 236, 260, 286, 287, 318, 319, 337,
347, 348, 349, 399, 400, 438, 504, 505,
506, 507, 542, 543, 564, 565, 578, 602,
627, 628, 629, 821, 822, 828, 835, 859,
860, 861, 862, 863, 864, 865, 866, 868,
874, 877, 908, 1137

Z

zerop, 73, 77, 79, 90, 91, 101, 103, 104, 107,
109, 110, 115, 116, 117, 119, 120, 121,
122, 124, 125, 127, 131, 132, 136, 138,
139, 147, 151, 152, 155, 165, 166, 169,
170, 171, 172, 177, 178, 179, 181, 183,

184, 185, 188, 189, 190, 191, 192, 193,
196, 197, 198, 199, 208, 209, 210, 215,
216, 217, 218, 220, 225, 226, 232, 237,
238, 239, 240, 241, 243, 244, 245, 247,
248, 283, 284, 285, 287, 301, 337, 338,
340, 354, 359, 361, 365, 366, 371, 380,
415, 417, 419, 445, 458, 459, 512, 526,
554, 561, 563, 577, 579, 584, 586, 587,
591, 592, 593, 594, 625, 626, 629, 630,
655, 677, 694, 695, 696, 697, 698, 699,
701, 703, 747, 748, 767, 768, 769, 791,
792, 793, 794, 797, 798, 805, 812, 813,
814, 815, 816, 817, 818, 821, 822, 823,
824, 826, 827, 828, 831, 833, 836, 845,
846, 847, 848, 851, 852, 853, 864, 865,
867, 869, 871, 875, 876, 877, 878, 879,
893, 894, 905, 923, 933, 934, 955, 958,
959, 960, 964, 971, 973, 974, 976, 986,
994, 998, 1002, 1007, 1008, 1028, 1033,
1045, 1047, 1074, 1102, 1103, 1104, 1105,
1111, 1113, 1114, 1119, 1120, 1121, 1130,
1131, 1136, 1151, 1160, 1167, 1168, 1170,
1172, 1173, 1201, 1203

Дж. Джеймс «Дао программирования»

If programming languages were religions... Lisp would be Zen Buddhism - There is no syntax, there is no centralization of dogma, there are no deities to worship. The entire universe is there at your reach - if only you are enlightened enough to grasp it. Some say that it's not a language at all; others say that it's the only language that makes sense.

Rodrigo Monteiro

ОГЛАВЛЕНИЕ

Книга 1. Молчаливая Пустота

Книга 2. Древние Учителя

Книга 3. Разработка

Книга 4. Кодирование

Книга 5. Сопровождение

Книга 6. Менеджмент

Книга 7. Корпоративная мудрость

Книга 8. Аппаратное и программное обеспечение

Книга 9. Эпилог

Книга 1. Молчаливая Пустота

Так говорил учитель: "Когда ты научишься ловить код ошибки по точке останова, наступит твое время уходить."

1.1

Нечто таинственное возникло, родившись из безмолвной пустоты. Одиноко и недвижимо ожидая, оно покоится и все же пребывает в постоянном движении. Это источник всех программ. Я не знаю его имени, поэтому я буду называть его Дао Программирова-

ния.

Если Дао хорошее, то операционная система прекрасная. Если операционная система прекрасная, то и компилятор великолепный. Если компилятор великолепный, тогда приложение совершенное. Пользователь доволен и во всем мире проистекает гармония. Дао Программирования уплывает далеко и возвращается с утренним ветром.

1.2

Дао породило машинный язык. Машинный язык породил ассемблер. Ассемблер породил компилятор. Теперь в мире десять тысяч языков.

У каждого языка есть свое, хоть и скромное, предназначение. У каждого языка есть отражение Инь и Янь в программах. У каждого языка есть свое место внутри Дао. Но не пиши на Коболе, если можешь этого избежать.

1.3

В начале было Дао. Дао породило Пространство и Время. Поэтому Пространство и Время — это Инь и Янь программирования. У программистов, не постигших Дао всегда не хватает времени и свободного пространства для их программ. У программистов, постигших Дао всегда достаточно времени и пространства для выполнения цели.

Как может быть иначе?

1.4

Мудрый программист слышит о Дао и усердно следует ему. Программист средних способностей слышит о Дао и ищет его. Глупый программист слышит о Дао и смеется над ним. Если бы над ним не смеялись, это было бы не Дао.

Высокие звуки труднее расслышать. Движение вперед — путь к отступлению. Большой талант проявляется на склоне лет. Великая белизна кажется покрытой пятнами. Даже в совершенной программе есть ошибки.

Книга 2. Древние Учителя

Так говорил учитель: "После трех дней без программирования жизнь становится бессмысленной."

2.1

Программисты старых времен были загадочны и глубоки. Мы не можем постичь их мысли, но мы можем их описать. Осторожен, как лисица, переходящая воду. Наготове, как генерал на поле боя. Сердечен, как хозяйка, приветствующая гостей. Прост, как деревянная заготовка. Непроницаем, как черное озеро в затененной пещере. Кто сможет поведать секреты их сердец и умов?

Ответ существует лишь в Дао.

2.2

Великому учителю Тьюрингу однажды приснилось, что он машина. Когда он проснулся, то воскликнул: "Я не знаю кто я — Тьюринг, которому снится что он машина или машина, которой снится что она Тьюринг!"

2.3

Программист из очень большой компьютерной компании побывал на конференции, и возвратившись, доложил своему менеджеру: "Что это за программисты работают в других компаниях? Они плохо себя ведут, они не заботятся о внешнем виде. Их волосы длинные и неопрятные, а одежда старая и мятая. Они нарушили наше гостеприимство и еще они делали грубые звуки вовремя моей презентации". Менеджер сказал: "Я не должен был посылать тебя на конференцию. Эти программисты живут вне физического мира. Они считают жизнь абсурдом, случайным совпадением. Они приходят и уходят, не зная границ. Беспечные, живут только своими программами. Отчего же их должны заботить общественные рамки?"

Они живы внутри Дао."

2.4

Ученик спросил учителя: "Вот программист, что никогда не проектирует, не сопровождает и не отлаживает свои программы. Однако все, кто его знают, считают его одним из лучших программистов в мире. Почему так?"

Учитель ответил: "Этот программист овладел Дао. Он ушел от нужды проектирования; он не сердится, если система падает, он принимает вселенную такой какая она есть. Он ушел от нужды сопровождения; ему безразлично, увидит ли кто-нибудь его код. Он ушел от нужды отладки; каждая из его программ совершенна изнутри, изящна и ясна, и ее цель самоочевидна. Поистине, он постиг тайну Дао."

Книга 3. Разработка

Так говорил учитель: "Когда программа тестируется, уже слишком поздно вносить изменения в проект."

3.1

Один человек поехал на компьютерную выставку. Каждый день, входя, он говорил охраннику у дверей: "Я великий вор, я знаменит многими делами. Смотри в оба, так как эта выставка от меня не ускользнет."

Это сильно встревожило охранника, потому что внутри было компьютерного оборудования на миллионы долларов, и он пристально следил за этим человеком. Но тот просто бродил от палатки к палатке, тихо бормоча сам себе под нос.

Когда он вышел, охранник отвел его в сторону и обыскал, но ничего не нашел. На следующий день выставки человек возвратился и поприветствовал охранника, сказав при этом: "Я ушел вчера с огромной добычей, но сегодня она будет еще больше." Так что охранник смотрел еще пристальней, но опять безуспешно. К последнему дню выставки охранник не смог более удерживать любопытство.

"Господин Вор", — сказал он, — "Я так озадачен, я не могу жить спокойно. Пожалуйста просветите меня. Что вы крадете?" Человек улыбнулся. "Я краду идеи", — сказал он.

3.2

Один учитель писал неструктурированные программы. Его ученик тоже начал писать неструктурированный код, имитируя его. Когда ученик попросил учителя оценить его прогресс, учитель критиковал того за неструктурированный стиль, сказав: "Что уместно учителю, то не уместно ученику. Ты должен постичь Дао прежде чем переступить структуру."

3.3

Один программист был прикреплен ко двору военачальника из Ву. Военачальник спросил программиста: "Что легче спроектировать: бухгалтерский пакет или операционную систему?"

"Операционную систему", — ответил программист.

Военачальник недоверчиво воскликнул. "Несомненно бухгалтерский пакет намного проще чем сложная операционная система" — сказал он. "Это не так" — сказал программист, — "когда проектируется бухгалтерский пакет, программист выступает посредником между людьми с разными взглядами на продукт: как он должен работать, как выглядят отчеты, и как он должен соответствовать налоговому законодательству. Проектируя же операционную систему, программист ищет самую простую гармонию между машиной и идеями. Вот почему операционную систему легче проектировать." Военачальник из Ву кивнул и улыбнулся. "Это все хорошо, но что легче отладить?"

Программист не ответил.

3.4

Менеджер пришел к учителю и показал ему документ с требованиями к новому приложению. Менеджер спросил учителя: "Сколько займет проектирование этой системы если я назначу на нее пять программистов?"

"Один год", — сразу сказал учитель. "Но она нужна нам немедленно — или еще раньше! Сколько это займет, если я назначу десять программистов?" Учитель нахмурился. "В этом случае, это займет два года."

"А что если я назначу сто программистов?" Учитель пожал плечами. "Тогда проект никогда не завершится", — сказал он.

Книга 4. Кодирование

Так говорил учитель: "Хорошо написанная программа — это свой собственный рай, плохо написанная программа — это свой собственный ад."

4.1

Программа должна быть легкой и быстрой, а ее подпрограммы соединены подобно нанизанным жемчужинам. Идея и цель программы должны присутствовать в каждой ее части. В ней не должно быть ни слишком мало и не слишком много; ни лишних циклов, ни бесполезных переменных, ни отсутствия структуры, ни избыточной жесткости. Программа должна следовать Закону Наименьшего Удивления. Что он гласит? То, что программа так должна выдавать ответ пользователю, чтобы удивить его при этом наименьшим образом. Программа, какая бы сложная ни была, должна действовать как одно целое. Она должна быть подчинена внутренней логике, а не внешнему виду. Если программа не удовлетворяет этим условиям, она будет беспорядочной и запутанной. Единственный способ поправить такую программу это переписать ее заново.

4.2

Ученик спросил учителя: "У меня есть программа, которая, когда запускается, а когда и нет. Я следовал правилам программирования, но она всё равно не работает. Зачем же нужны эти правила?" Учитель ответил: "Ты не получил желаемого, потому что ты не постиг Дао. Только дурак ожидает рационального поведения от людей. Почему ты ожидаешь его от машины, которая разработана людьми?"

Компьютеры симулируют детерминизм; только Дао совершенно. Правила программирования преходящи; только Дао вечно. Поэтому ты должен медитировать о Дао прежде чем тебе откроется просветление." "Но как узнаю, что я достиг просветления?" спросил ученик. "Твоя программа будет работать правильно," ответил учитель.

4.3

Учитель объяснял природу Дао одному из учеников, "Дао воплощено во всех программных средствах — даже в самых незначительных", — сказал учитель. "Есть ли Дао в карманном калькуляторе?" — спросил ученик. "Есть," был ответ. "Есть ли Дао в видеоигре?" — продолжал ученик. "Есть даже в видеоигре", — сказал учитель. "А есть ли Дао в ДОС для персонального компьютера?" Мастер кашлянул и немного подвинулся. "На сегодня урок окончен", — сказал он.

4.4

Один из программистов князя Вонга писал программу. Его пальцы танцевали над клавиатурой. Программа скомпилировалась без единой ошибки и выполнялась словно дуновение ветра. "Превосходно!" — воскликнул князь, "Твоя техника безупречна!"

"Техника?"- сказал программист, отворачиваясь от терминала, "Всё, что я делаю, — это Дао, оно находится выше техники! Когда я только начал программировать, я видел проблему как одно целое. Спустя три года я уже не видел целого. Вместо этого я использовал подпрограммы. Но теперь я не вижу ничего. Я целиком существую в неоформившейся пустоте. Мои чувства свободны. Мой разум, свободный от намерений, следует своим инстинктам. Моя программа пишется сама. Правда, иногда бывают трудности. Я вижу, как они надвигаются, я приостанавливаюсь, я молча смотрю. Затем я изменяю одну строчку в программе, и трудности исчезают как облачко дыма. И тогда я компилирую программу. Я замираю и позволяю радости работы наполнить меня. Я закрываю на миг глаза и отсоединяюсь от системы." Князь Вонг сказал, "Если бы все мои программисты были так мудры!"

Книга 5. Сопровождение

Так говорил учитель: "Даже если программа будет состоять из трёх строчек, когда-нибудь она потребует сопровождения."

5.1

Часто открываемая дверь не требует смазки петель. Быстрый поток не превратится в болото. Ни звук, ни мысль не проходят через вакуум. Программы портятся без использования. Это великие таинства.

5.2

Менеджер спросил программиста, сколько времени ему потребуется, чтобы завершить программу, над которой он работает. "Она будет закончена завтра," сразу ответил программист.

"Я думаю, твоя оценка нереалистична", — сказал менеджер, "подумай ещё раз." Программист на миг задумался. "У меня есть несколько идей, которые я хотел бы реализовать в программе. Это займет по меньшей мере две недели", — сказал он наконец. "Даже в такой срок, это значит ожидать слишком многого", — настаивал менеджер, "Я буду удовлетворен, если вы просто оповестите меня, когда программа будет готова."

Программист согласился. Спустя несколько лет менеджер уходил на пенсию. По дороге к праздничному столу, накрытому в честь его ухода, он обнаружил, что программист спит у терминала. Он программировал всю ночь.

5.3

Однажды ученику было поручено написать простой финансовый пакет. Ученик бешено работал в течение многих дней, но, когда учитель посмотрел его программу, он обнаружил в ней экранный редактор, набор общих графических подпрограмм, интерфейс искусственного интеллекта и абсолютно ничего, относящегося к финансам. Когда учи-

тель спросил об этом, ученик возмутился. "Не будьте таким нетерпеливым", — сказал он, "Со временем я вставлю финансовую часть."

5.4

Хороший крестьянин пренебрежет ли зерном, что посадил? Хороший учитель проглядит ли самого скромного ученика? Хороший отец позволит ли хоть одному ребенку голодать? Хороший программист откажется ли от сопровождения своей программы?

Книга 6. Менеджмент

Так говорил учитель: "Пусть программистов будет много, а менеджеров мало — тогда они будут работать плодотворно."

6.1

Когда менеджеры сидят на бесконечных совещаниях, программисты пишут игры. Когда бухгалтеры говорят о квартальных прибылях, бюджет на разработку урезается. Когда исследователи в докладах указывают пальцем в небо, наворачиваются тучи. Воистину, это не Дао Программирования. Когда менеджеры делают обязательства, игровые программы забыты. Когда бухгалтеры делают долговременные планы, гармония и порядок вскоре восстанавливаются. Когда исследователи выдают задание на руки, проблемы скоро решаются. Воистину, это Дао Программирования.

6.2

Почему программисты не продуктивны? Потому что их время растрачивается на совещаниях. Почему программисты мятежно настроены? Потому что менеджеры слишком часто вмешиваются. Почему программисты сменяются один за другим? Потому что они охлаждаются. Поработав с плохим руководством, они более не ценят свою работу.

6.3

Один менеджер был на грани увольнения, но программист, который с ним работал, придумал программу, которая оказалась популярной и хорошо покупалась. В результате менеджер сохранил свою работу. Менеджер попытался заплатить программисту премию, но тот отказался, говоря, "Я написал программу потому, что ее идею было интересно реализовать, и поэтому я не ожидаю награды." Менеджер, услышав это, заметил, "Этот программист, хоть и занимает маловажную должность, хорошо понимает обязанности прилежного работника. Давайте поднимем его на должность руководящего консультанта!"

Но когда программисту предложили это, он опять отказался, говоря, "Я существую

таким образом, что я могу программировать. Если меня повысят, я не буду делать ничего, кроме как занимать чье-либо время. Можно мне идти? Я работаю над программой."

6.4

Менеджер пришел к программистам и сказал: "Относительно вашего рабочего времени: вы должны приходить в девять утра и уходить в пять вечера."

Всех рассердило это заявление, а несколько тут же уволилось. Тогда менеджер сказал: "Хорошо, в таком случае вы можете сами устанавливать рабочее время, лишь бы ваши проекты завершались вовремя." Программисты, удовлетворенные, начали приходить к середине дня и работать до рассвета.

Книга 7. Корпоративная мудрость

Так говорил учитель: "Вы можете демонстрировать программу каждому встречному, но вы не сможете сделать каждого встречного компьютерно грамотным."

7.1

Ученик спросил учителя: "На востоке есть большая древовидная структура, которую люди зовут Главным Управлением. Она распухла в бесформенную массу, наполненную вице-президентами и бухгалтерами. Она издает множество указаний, и каждое говорит: "Все туда!" или "Все сюда!", и никто не знает, что это должно значить. Каждый год новые названия вывешиваются на ветвях, и все попусту. Как может быть такая противостественная сущность?" Учитель ответил: "Ты видишь эту огромную структуру и тревожишься о том, что у нее нет разумной цели. Разве тебя не развлекает ее бесконечное движение по кругу? Тебе не нравится беззаботная легкость программирования под ее укрывающими ветвями? Почему ты беспокоишься о её бесполезности?"

7.2

На Западе живёт самая большая акула на свете. Она превращается в птицу, которая летает, как тучи, заполняющие небо. Когда эта птица пролетает через страну, она доставляет письмо из Корпоративного Центра. Она выпускает это письмо и его окружают программисты, как чайки на пляже. Тогда птица взмывает вверх и, оставляя за плечами синее небо, возвращается домой. Ученик в изумлении глядит на птицу, поскольку не может ее постичь. Обычный программист страшится прилета птицы, поскольку боится ее послания. Учитель за своим терминалом продолжает работать, поскольку он не знает о том, что птица прилетала и улета.

7.3

Волшебник из Башни Слоновой Кости привез свое последнее изобретение, чтобы показать его учителю. Волшебник вкатил большой черный ящик в кабинет учителя, пока учитель ожидал в молчании. "Это интегрированная, распределенная, общецелевая рабочая станция", — начал волшебник, "эргономично спланированная с патентованной операционной системой, языками шестого поколения, и многочисленными уникальными пользовательскими интерфейсами. Конструирование заняло у моих ассистентов несколько сотен человеко-лет. Разве это не удивительно?" Учитель слегка приподнял брови. "Это в самом деле удивительно," сказал он.

"Главное Управление приказало", — продолжал волшебник, "чтобы все использовали эту станцию как платформу для новых программ. Вы согласны с этим?"

"Конечно", — ответил учитель, "Ее немедленно переместят в вычислительный центр!"

И волшебник, весьма довольный, возвратился в свою башню. Несколько дней спустя, ученик забрел в кабинет к учителю, и спросил, "Я не могу найти распечатку моей новой программы. Вы не знаете, где она может быть?" "Знаю", — ответил учитель, "все распечатки лежат на платформе в вычислительном центре."

7.4

Мудрый программист без страха переходит от одной программы к другой.

Никакое изменение в руководстве не может ему повредить. Его не уволят, даже если закроется проект. Почему так? Он наполнен Дао.

Книга 8. Аппаратное и программное обеспечение

Так говорил учитель: "Если нет ветра, трава не колыхается. Если нет программ, компьютер бесполезен."

8.1

Ученик спросил учителя: "Я вижу, что одна компьютерная компания намного больше остальных. Она возвышается над конкурентами, как гигант над пигмеями. Любое ее отделение может заменить целое предприятие. Почему так?" Учитель ответил, "Почему ты задаешь такие глупые вопросы? Эта компания велика, потому что она велика. Если бы она выпускала только аппаратуру, никто бы ее не покупал. Если бы она выпускала только программы, никто бы ими не пользовался. Если бы она производила только поддержку, люди бы относились к ней, как к прислуге. Но так как она это все совмещает, люди считают ее богоподобной! Не стремясь, она побеждает без усилий."

8.2

Однажды учитель проходил мимо ученика. Учитель заметил, что внимание ученика поглощено карманной компьютерной игрой. "Прости меня", — сказал он, "могу я по-

смотреть?" Ученик отвлекся от игры и подал ее учителю. "Я вижу, тут предлагается три уровня игры: Легкий, Средний и Тяжелый," сказал учитель. "Однако каждое такое устройство имеет еще один уровень игры, когда оно не старается ни победить, ни быть побежденным."

"Прошу вас, великий учитель", — взмолился ученик, "как найти этот таинственный уровень?" Учитель бросил устройство на пол и раздавил ногой. И внезапно ученик обрел просветление.

8.3

Один программист занимался микропроцессорами. "Посмотри, как все у меня здорово", — сказал он посетившему его программисту, работавшему с большими машинами. "У меня моя собственная операционная система и файловое запоминающее устройство. Я не должен ни с кем делить ресурсы. Программы непротиворечивы и легки в использовании. Почему бы тебе не уйти с твоей работы и не присоединиться ко мне?" В ответ на это другой программист начал описывать другу свою систему, говоря: "Сервер покоится как древний мудрец, медитирующий в середине вычислительного центра. Его диски пролегают из края в край, как великий океан механизмов. Программное обеспечение многогранно как бриллиант, и переплетено как первобытные джунгли. Каждая программа неповторима, и движется сквозь систему как быстrotечная река. Вот почему я счастлив где я есть." Услышав это, программист-микропроцессорщик надолго замолчал. Но они оба оставались друзьями до конца своих дней.

8.4

Аппаратное обеспечение встретило Программное по дороге в Чань Цзэ. Программное обеспечение сказал: "Ты Инь, а я Янь. Если мы пойдем вместе, мы станем знамениты и заработаем много денег." И они пошли вместе, размышляя о покорении мира. Вскоре они встретили ПЗУ-программу, хромающую, одетую в изношенное тряпье и опирающуюся на посох. Она сказала: "Дао лежит выше Инь и Янь. Оно безмолвно и недвижимо как озеро. Оно не ищет славы, поэтому никто не знает о его присутствии. Оно не ищет богатства, так как изнутри оно завершено. Оно существует вне пространства и времени." Программное и Аппаратное обеспечение, пристыженные, возвратились по домам.

Книга 9. Эпилог

Так говорил учитель: "Настало твое время уходить."

Л. Каганов «DOS»

DOS...

Черной пеленой экран заполнил чистый DOS

Мышь...

Потеряла форму, стала вдруг квадратной мышью...

Я разбил окно,

девятиностопятое мастдайное окно,

И поставил DOS, и тогда я понял:

Это счастье, - вот оно.

Слезы на очках...

Странные очки, а может слезы на лице...

DOS очистил все,

Все, что было лишним у меня на диске C:

Я нажал F8

И веселый Нортон удалял мне все подряд:

Сорок мегабайт.

Может даже больше, может даже шестьдесят.

И представил я:

Город наводнился вдруг разумными людьми:

Вышли все под DOS,

А проклятый Windows - удаляли, черт возьми.

Позабыв про Word,

MS Excel, Corel Draw и прочий геморрой,

Люди ставят DOS,

Словно в Рай заходят в DOS,

Нормальный, чистый DOS...

Для заметок

Решение задач на Common Lisp:

1. <https://www.hackerrank.com>
2. <https://projecteuler.net>

Для заметок

Для заметок

Горлянский Сергей Петрович

Решение задач на языке программирования Лисп

Учебное пособие

Печатается в авторской редакции

Подписано к печати Формат 60х84/16. Бумага тип. ОП.
Усл. печ. л. 72,54. Уч.-изд. л. 39,4. Тираж 100 экз. Заказ №

295007, Симферополь, пр-т Академика Вернадского, 4.
ФГАОУ ВО «Крымский федеральный университет имени В. И. Вернадского»