

# МИР ЛИСПА. Т.1: ВВЕДЕНИЕ В ЯЗЫК ЛИСП И ФУНКЦИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ

Двухтомник финских специалистов, содержащий введение в язык Лисп, методы и системы программирования. Этот язык широко известен и применяется в задачах символьной обработки информации, обработки естественных языков, искусственного интеллекта, экспертных систем, систем логического программирования. Изложение языка и примеры основаны на последней версии, которая станет стандартом языка. В книге приведены конкретные задачи с ответами и решениями. В 1-м томе даны основные понятия языка Лисп и введение в функциональное программирование.

Для программистов разной квалификации, для всех, использующих язык Лисп.

## Содержание

Предисловие редактора перевода	5
<b>ВВЕДЕНИЕ</b>	<b>11</b>
Скачок в развитии вычислительной техники	11
Исследовательские программы искусственного интеллекта	12
Национальные программы по исследованию языков	13
Появление Лиспа в Финляндии	13
Лисп - основа искусственного интеллекта	14
Учебник Лиспа на финском языке	14
Язык Лисп и функциональное программирование	15
Методы и системы программирования	16
На кого рассчитана книга	17
Терминология	17
Иконология	18
От дерева к мысли и от мысли к дереву	19
Благодарности	20
<b>1 ВВЕДЕНИЕ В МИР ЛИСПА</b>	<b>21</b>
<b>1.1 СИМВОЛЬНАЯ ОБРАБОТКА И ИСКУССТВЕННЫЙ ИНТЕЛЛЕКТ</b>	<b>22</b>
Искусственный интеллект и технология знаний	23
Исторические предубеждения	23
Символьное или численное вычисление	24
Эвристическое или алгоритмическое решение задачи	25
Искусственный интеллект - сфера исследования многих наук	27
Знание дела и умение как товар	27
Учебники по искусенному интеллекту	28
Упражнения	29
<b>1.2 ПРИМЕНЕНИЯ ИСКУССТВЕННОГО ИНТЕЛЛЕКТА</b>	<b>30</b>
Многообразие форм искусственного интеллекта	30
Обработка естественного языка	31
Экспертные системы	33
Символьные и алгебраические вычисления	35

Доказательства и логическое программирование	36
Программирование игр	37
Моделирование	37
Обработка сигналов и распознавание образов	38
Машинное зрение и обработка изображений	39
Робототехника и автоматизация производства	39
Машинное проектирование	40
Языки и средства программирования искусственного интеллекта ч.	40
Повышение производительности программирования	41
Автоматическое программирование и обучение	41
Литература	42
Упражнения	45
<b>1.3 ЛИСП - ЯЗЫК ПРОГРАММИРОВАНИЯ ИСКУССТВЕННОГО ИНТЕЛЛЕКТА</b>	<b>46</b>
Символьная обработка	47
Лисп опередил свое время	48
Однаковая форма данных и программы	49
Хранение данных, не зависящее от места	50
Автоматическое и динамическое управление памятью	51
Функциональный образ мышления	51
Возможность различных методов программирования	52
Пошаговое программирование	52
Интерпретирующий или компилирующий режимы выполнения	53
Лисп - бестиповый язык программирования	53
Единый системный и прикладной язык программирования	54
Интегрированная среда программирования	55
Широко распространенные заблуждения и предрассудки	56
Простой и эффективный язык	57
Учебная литература по Лиспу	58
Упражнения	59
<b>2 ОСНОВЫ ЯЗЫКА ЛИСП</b>	<b>60</b>
<b>2.1 СИМВОЛЫ И СПИСКИ</b>	<b>61</b>
Символы используются для представления других объектов	61
Символы в языке Коммон Лисп	62
Числа являются константами	63
Логические значения Т и NIL	64
Константы и переменные	64
Атомы - Символы + Числа	64
Построение списков из атомов и подсписков	65
Пустой список = NIL	65
Список как средство представления знаний	66
Значение способа записи	67
Различная интерпретация списков	67
Упражнения	68

<b>2.2 ПОНЯТИЕ ФУНКЦИИ</b>	69
Функция - отображение между множествами	69
Тип аргументов и функций	70
Определение и вызов функции	71
Единообразная префиксная нотация	71
Аналогия между Лиспом и естественным языком	73
Диалог с интерпретатором Лиспа	73
Иерархия вызовов	74
QUOTE блокирует вычисление выражения	75
Упражнения	77
<b>2.3 БАЗОВЫЕ ФУНКЦИИ</b>	78
Основные функции обработки списков	79
Функция CAR возвращает в качестве значения головную часть списка	80
Функция CDR возвращает в качестве значения хвостовую часть списка	81
Функция CONS включает новый элемент в начало списка	83
Связь между функциями CAR, CDR и CONS	84
Предикат проверяет наличие некоторого свойства	85
Предикат ATOM проверяет, является ли аргумент атомом	85
EQ проверяет тождественность двух символов	86
EQUAL сравнивает числа одинаковых типов	88
Предикат * сравнивает числа различных типов	88
EQUALP проверяет идентичность записей	89
EQUALP проверяет наиболее общее логическое равенство	90
Другие примитивы	90
NULL проверяет на пустой список	91
Вложенные вызовы CAR и CDR можно записывать в сокращенном виде	91
LIST создает список из элементов	93
Упражнения	94
<b>2.4 ИМЯ И ЗНАЧЕНИЕ СИМВОЛА</b>	95
Значением константы является сама константа	95
Символ может обозначать произвольное выражение	96
SET вычисляет имя и связывает его	96
SETQ связывает имя, не вычисляя его	97
SETF - обобщенная функция присваивания	98
Побочный эффект псевдофункции	99
Вызов интерпретатора EVAL вычисляет значение значения	100
Основной цикл: READ-EVAL-PRINT	102
Упражнения	103
<b>2.5 ОПРЕДЕЛЕНИЕ ФУНКЦИЙ</b>	104
Лямбда-выражение изображает параметризованные вычисления	104
Лямбда-вызов соответствует вызову функции	106
Вычисление лямбда-вызова, или лямбда-преобразование	106

Объединение лямбда-вызовов	107
Лямбда-выражение - функция без имени	108
DEFUN дает имя описанию функции	108
SYMBOL-FUNCTION выдает определение функции	110
Задание параметров в лямбда-списке	111
Изображение функций в справочных руководствах	114
Функции вычисляют все аргументы	115
Многозначные функции	115
Определение функции в различных диалектах Лиспа	115
При вычислении NLAMBDA аргументы не вычисляются	117
Упражнения	117
<b>2.6 ПЕРЕДАЧА ПАРАМЕТРОВ И ОБЛАСТЬ ИХ ДЕЙСТВИЯ</b>	<b>119</b>
В Лиспе используется передача параметров по значению	119
Статические переменные локальны	120
Свободные переменные меняют свое значение	121
Динамическая и статическая область действия	121
Одно имя может обозначать разные переменные	123
Упражнения	125
<b>2.7 ВЫЧИСЛЕНИЕ В ЛИСПЕ</b>	<b>127</b>
Программа состоит из форм и функций	127
Управляющие структуры Лиспа являются формами	128
LET создает локальную связь	129
Последовательные вычисления: PROG1, PROG2 и PROGN	131
Разветвление вычислений: условное предложение COND	132
Другие условные предложения: IF, WHEN, UNLESS и CASE	136
Циклические вычисления: предложение DO	137
Предложения PROG, GO и RETURN	139
Другие циклические структуры	142
Повторение через итерацию или рекурсию	142
Формы динамического прекращения вычислений: CATCH и THROW	145
Упражнения	146
<b>2.8 ВНУТРЕННЕЕ ПРЕДСТАВЛЕНИЕ СПИСКОВ</b>	<b>148</b>
Лисповская память состоит из списочных ячеек	149
Значение представляется указателем	149
CAR и CDR выбирают поле указателя	151
CONS создает ячейку и возвращает на нее указатель	151
У списков могут быть общие части	152
Логическое и физическое равенство не одно и то же	154
Точечная пара соответствует списочной ячейке	155
Варианты точечной и списочной записей	157
Управление памятью и сборка мусора	159
Вычисления, изменяющие и не изменяющие структуру	160
RPLACA и RPLACD изменяют содержимое полей	161
Изменение структуры может ускорить вычисления	163

Упражнения	166
<b>2.9 СВОЙСТВА СИМВОЛА</b>	<b>168</b>
У символа могут быть свойства	168
У свойств есть имя и значение	169
Системные и определяемые свойства	169
Чтение свойства	169
Присваивание свойства	170
Удаление свойства	171
Свойства глобальны	171
Упражнения	172
<b>2.10 ВВОД И ВЫВОД</b>	<b>174</b>
Ввод и вывод входят в диалог	174
READ читает и возвращает выражение	175
Программа ввода выделяет формы	176
Макросы чтения изменяют синтаксис Лиспа	177
Символы хранятся в списке объектов	179
Пакеты или пространства имен	180
PRINT переводит строку, выводит значение и пробел	180
PRIN1 и PRINC выводят без перевода строки	182
TERPRI переводит строку	183
FORMAT выводит в соответствии с образцом	184
Использование файлов	187
LOAD загружает определения	190
Упражнения	191
<b>3 ФУНКЦИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ</b>	<b>193</b>
<b>3.1 ОСНОВЫ РЕКУРСИИ</b>	<b>194</b>
Лисп - это язык функционального программирования	194
Процедурное и функциональное программирование	195
Рекурсивный - значит использующий самого себя	196
Рекурсия всегда содержит терминальную ветвь	197
Рекурсия может проявляться во многих формах	198
Списки строятся рекурсивно	200
Лисп основан на рекурсивном подходе	201
Теория рекурсивных функций	201
Литература	204
<b>3.2 ПРОСТАЯ РЕКУРСИЯ</b>	<b>205</b>
Простая рекурсия соответствует циклу	205
MEMBER проверяет, принадлежит ли элемент списку	208
Каждый шаг рекурсии упрощает задачу	209
Порядок следования ветвей в условном предложении существенней	211
Ошибка в условиях может привести к бесконечным вычислениям	213
APPEND объединяет два списка	214
REMOVE удаляет элемент из списка	216
SUBSTITUTE заменяет все вхождения элемента	217

REVERSE обращает список	218
Использование вспомогательных параметров	220
Упражнения	221
<b>3.3 ДРУГИЕ ФОРМЫ РЕКУРСИИ</b>	<b>224</b>
Параллельное ветвление рекурсии	225
Взаимная рекурсия	228
Программирование вложенных циклов	229
Рекурсия более высокого порядка	232
Литература	235
Упражнения	235
<b>3.4 ФУНКЦИИ БОЛЕЕ ВЫСОКОГО ПОРЯДКА</b>	<b>239</b>
Функционал имеет функциональный аргумент	239
Функциональное значение функции	241
Способы композиции функций	242
Функции более высокого порядка	243
Литература	244
<b>3.5 ПРИМЕНЯЮЩИЕ ФУНКЦИОНАЛЫ</b>	<b>245</b>
APPLY применяет функцию к списку аргументов	246
FUNCALL вызывает функцию с аргументами	246
Упражнения	248
<b>3.6 ОТОБРАЖАЮЩИЕ ФУНКЦИОНАЛЫ</b>	<b>249</b>
Отображающие функции повторяют применение функции	249
MAPCAR повторяет вычисление функции на элементах списка	250
MAPLIST повторяет вычисление на хвостовых частях списка	252
MAPCAN и MAPCON объединяют результаты	252
MAPC и MAPL теряют результаты	254
Композиция функционалов	255
Итоговая таблица отображающих функций	256
Упражнения	257
<b>3.7 ЗАМЫКАНИЯ</b>	<b>259</b>
FUNCTION блокирует вычисление функции	259
Замыкание - это функция и контекст ее определения	260
Связи свободных переменных замыкаются.	261
Замыкания позволяют осуществлять частичное вычисление Г	263
Генератор порождает последовательные значения	264
Контекст вычисления функционального аргумента	265
Литература	269
Упражнения	269
<b>3.8 АБСТРАКТНЫЙ ПОДХОД</b>	<b>270</b>
Обобщение функций, имеющих одинаковый вид	271
Параметризованное определение функций	275
Рекурсивные функции с функциональным значением	279
Автоаппликация и авторепликация	280
Порядок и тип функций	283

Проблемы абстрактного подхода	285
Литература	286
Упражнения	287
<b>3.9 МАКРОСЫ</b>	<b>288</b>
Макрос строит выражение и вычисляет его значение	288
Макрос не вычисляет аргументы	290
Макрос вычисляется дважды	290
Контекст вычисления макроса	291
Пример отличия макроса от функции	292
Рекурсивные макросы и продолжающиеся вычисления	294
Тестирование макросов	295
Лямбда-список и ключевые слова макроса	296
Обратная блокировка разрешает промежуточные вычисления	298
Образец удобно использовать для определения макросов	300
Макросы с побочным эффектом	301
Определение новых синтаксических форм	304
Определение типов данных с помощью макросов	305
Литература	306
Упражнения	307
<b>4 ТИПЫ ДАННЫХ</b>	<b>308</b>
<b>4.1 ПОНЯТИЯ</b>	<b>309</b>
Явное и неявное определение	309
Абстракция данных	310
Составные типы и процедуры доступа	312
В Лиспе тип связан со значением, а не с именем	312
Проверка и преобразование типов	314
Иерархия типов	316
Определение новых типов	316
<b>4.2 ЧИСЛА</b>	<b>319</b>
Лисп умеет работать с числами	319
Целые числа	323
Дробные числа	323
Числа с плавающей запятой	324
Комплексные числа	325
<b>4.3 СИМВОЛЫ</b>	<b>326</b>
Системные свойства символа	326
Специальные знаки в символах	327
Обращение с внешним видом символа	328
GENTEMP создает новый символ	330
<b>4.4 СПИСКИ</b>	<b>331</b>
Ассоциативный список связывает данные с ключами	332
PAFR LIS строит список пар	332
ASSOC ищет пару, соответствующую ключу	333
ACONS добавляет новую пару в начало списка	334

PUTASSOC изменяет а-список	335
<b>4.5 СТРОКИ</b>	<b>337</b>
Знаки и строки	337
Преобразования строк	338
Работа со строками	339
Наследуемые функции	340
<b>4.6 ПОСЛЕДОВАТЕЛЬНОСТИ</b>	<b>341</b>
Последовательности являются списками или векторами	341
Основные действия с последовательностями	342
Мощные функционалы	344
Упорядочивание последовательности	347
<b>4.7 МАССИВЫ</b>	<b>349</b>
Типы массивов	350
Работа с массивами	350
Хэш-массив ассоциирует данные	351
Хэш-массивы обеспечивают быстродействие	351
<b>4.8 СТРУКТУРЫ</b>	<b>353</b>
Структуры представляют собой логическое целое	353
Определение структурного типа и создание структуры	354
Функции создания, доступа и присваивания	354
Действия не зависят от способа реализации	356
<b>5 РЕШЕНИЯ</b>	<b>358</b>
<b>ПРИЛОЖЕНИЕ 1. СВОДКА КОММОН ЛИСПА</b>	<b>384</b>
<b>ПРИЛОЖЕНИЕ 2. УКАЗАТЕЛЬ СИМВОЛОВ КОММОН ЛИСПА</b>	<b>417</b>
<b>ПРИЛОЖЕНИЕ 3. УКАЗАТЕЛЬ ИМЕН И СОКРАЩЕНИЙ</b>	<b>431</b>
<b>ПРИЛОЖЕНИЕ 4. ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ</b>	<b>434</b>

## УКАЗАТЕЛЬ ИМЕН И СОКРАЩЕНИЙ

В этом приложении собраны встречающиеся в тексте имена и сокращения. Одновременно оно может служить указателем авторов по перечням литературы. С его помощью можно найти литературу, которая вследствие разбиения по темам приведена в разных разделах. Символы и зарезервированные слова Коммон Лиспа образуют отдельный указатель (приложение 2).

Авсоний 22	Гёте И. 337
Ада 194	Дюамель Ж. 270
Аккерман 202, 232	Дюма А. мл. 239
Алгол-68 310	Зеталисп 16, 40
АНАЛИТИК 35	Интерлисп 17, 40, 55, 116
Ауэрбах Б. 309	Калевала 19
Байрон 249	Клини 116
Бейсик 40, 194	Кобол 40, 194
Бирс А. 194	Коммон Лисп 16, 54, 55, 319
Блейк У. 319	Конфуций 245
Бэкон Р. 69	Крабб Дж. 174
Витгенштейн Л. 46, 288	Лавров С. 58
Вольтер 349	Лец Е. 127, 148, 205

- Лисп 40, 48, 194, 284  
Лого 40  
Маккарти Дж. 82  
Маклисп 55, 116, 183, 214, 319  
Мир 35  
Ницше Ф. 104  
Паронен С. 119  
Паскаль 40, 54, 194, 312, 353  
Паскаль Б. 331  
ПЛ/1353  
Пролог 40, 55, 194, 284  
пятое поколение 24  
СБИС 40  
Си 40, 194  
Силагадзе Г. 58  
Симула 37  
Смолтолк 40, 284  
Твен М. 326, 353  
Тьюринг 201  
Фибоначчи 202  
Фортран 40, 48, 54, 194, 313, 319, 320  
Франц Лисп 111, 262  
Фуллер Т. 224  
Харрис С. 11  
Черч 104  
Эмерсон Р. 341  
д'Юрфе О. 259  
Abrahams P. 235  
Allen J. 286  
Alpac 23  
Alwey 12  
Apl 194  
ART 41  
ATN41  
Bagley S. 59  
Banerji R. 28  
Barr A. 28  
Berwick R. 42  
Bibel W. 43  
Boden M. 44  
Boyer R. 286  
Brady J. 204, 244, 286  
Brady M. 42, 44  
Bramer D. 43  
Bramer M. 43  
Brown B. 306  
Brown R. 29  
Bundy A. 28  
Burge W. 286  
CAD 35, 40  
CAI35  
CAM 35  
Campbell J.A. 43  
CAR 82  
CASNET/GLAUCOMA 23  
CAT 35  
Cattel G. 58  
CDR82  
Chang C 43  
Charniak E. 28, 42  
Clancey W. 42  
Clark K. 43  
Clocksin W. 43  
Cohen B. 28  
Cohen H. 45  
Cohen P. 45  
Curry H. 269  
Danicic I. 58  
Darlington J. 286  
DARPA 12  
DCG 41  
DENDRAL 23  
Dreyfus H. 44  
Edwards D. 235  
Eisenstadt M. 29  
Emycin 41  
ESPRIT 12  
EUREKA 12  
Expert 41  
Feigenbaum E. 28  
Flavor 16  
Foerster H. 204, 286  
FP194  
Franz Lisp 111, 262  
Freys R. 269  
Friedman D. 58  
Friedman P. 269  
FRL41

- Gloess P. 58  
Goerz G. 287, 306  
Gorz G. 59  
Hall E. 44  
Hamann C.-M. 58  
Hanson A. 44  
Hart T. 235  
Hasemer T. 58  
Hayes-Roth F. 42  
Henderson P. 58,235,244, 269, 286  
Hofstadter D. 45, 204, 286  
Horn B. 306  
Huomo T. 44  
Hyvonen E. 42, 44  
IBM 605 82  
ICOT 12  
Interlisp 17, 40, 55, 116  
Johnston T. 44  
KEE41  
KL-TWO 41  
Knowledge Engineering Ky 15  
Kowalski R. 43  
Krc 194  
Krutch J. 29  
Kulikowski C 43  
Landin P. 269  
Lee R. 43  
Lenat D. 42  
Levin M. 235  
Logo 194  
Loveland D. 43  
MacLisp 55,116, 183, 214, 319  
MACSYMA 23, 35  
Makelin M. 44  
Marr D. 44  
Maurer W.D. 58  
MCC12  
McCarthy J. 58, 235  
McCorduck P. 45  
McDermott D. 28  
MelHsh C 43  
Michie D. 42  
Milner R. 244  
Minsky M. 204  
MIT 14  
MITI 12  
Moon D. 306  
Moore J. 286  
Morris J. 269  
MYCIN 23  
Neumann J. 286  
Nial 194  
Nii P. 45  
NIL 285  
Nilsson N. 29  
NITEC 15  
Nokia Informaatiojarjestelmat 20  
Norman A. 58  
O'Shea T. 29  
OPS541  
Perlis D. 287  
Prendergast K. 44  
Queinnec C 58, 306  
Rank Xerox 20  
Raphael B. 29, 45  
REDUCE 35  
Reisbeck C 42  
Ribbens D. 59  
Rich E. 29  
Riivari J. 44  
Riseman H. 44  
Rogers H. 204  
Rozsa P. 204  
S-1 319  
SAC 35  
Schank R. 42  
Schraeger J. 59  
Scott D. 287  
SECD-машина 268  
Seppanen J. 59, 235  
Shortliffe E. 42  
Siklossy L. 59  
SITRA 13  
Steele G. 59, 306  
STeP-84 13  
Stoyan H. 59, 287, 306  
Stratchey C 306  
Tarnlund S.-U. 43

TEKES 13  
Tennant H. 42  
Texas Instruments 20  
Touretzky D. 59  
Tractqn K. 59  
Turner D. 269  
VAX-11/780 275  
Waite M. 306  
Walker D. 42  
Waterman D. 42, 43  
Wegner P. 204

Weinreb D. 306  
Weismann C 59  
Weiss S. 43  
Weizenbaum J. 45  
WilenskyR. 59,306  
WilksY. 42  
Winograd T. 42  
Winston P. 29, 43, 44, 59, 306  
Wise D. 269  
Zetaisp 16, 40

## ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

Указатель составлен из встречающихся в тексте понятий и терминов, а также из специальных терминов языка Лисп. Имена и сокращения собраны в свой указатель (приложение 3) так же, как символы Ком мои Лиспа (приложение 2).

Абстракция вычислений 270  
- данных (data abstraction) 310  
- отображения 270  
Автофункция (auto-function) 243  
Аксиомы типа 310  
Анализ дискурса (discourse) 32  
- синтаксический 72  
Анализатор (parser) 177  
Аргумент 69,106  
- функциональный (functional argument) 239  
Атом (atom) 64  
Барьер сложности (complexity barrier) 286  
Блокировка вычислений (quote) 75  
- обратная (back quote) 233  
- функциональная (function quote) 259  
Вектор (vector) 350  
Верификация программ (program proving verification) 37  
Включение или интернирование (intern) 180, 329  
Выборка (selection) 312  
Вызов макроса (macro call) 128, 290  
- функции (call) 71, 240  
Выражение по умолчанию (init-form) 113  
- символьное (s-expression) 66  
Вычисление (evaluation) 73

- отложенное (lazy/suspended evaluation) 244, 265  
- параллельное 265  
- символьное и алгебраическое (symbolic and algebraic corn-put ing SAQ 35  
- частичное (partial evaluation) 244, 263  
- численное (numeric computing) 24  
Вычислимость (computability) 201  
- алгоритмическая (effective computability) 202  
Генератор (generator) 264  
- случайных чисел (random number generator) 322  
Голова списка (head) 80  
Гражданин полноправный (first class citizen) 111  
Графика компьютерная (computer graphics) 39  
Действия универсальные (generic) 311  
Дерево бинарное (binary tree) 225  
Доказательство теорем (theorem proving) 36  
Замыкание (lexical closure) 261, 315  
Запись (structure/record) 311, 353  
- (нотация) префиксная 71  
Знак (character) 337

- Знания (knowledge) 33
  - активные 28
- Значение (value) 69, 312, 326
  - глобальное (global) 122
  - логическое (boolean) 309
- Зрение машинное (machine vision) 39
- Игра, ведение (game playing) 37
  - программирование 37
- Иерархия понятий (conceptual hierarchy) 316
- Имя печатное (print name pna-me) 150, 326, 327
- Индекс (index) 350
- Интеллект искусственный (artificial intelligence machine intelligence) 23, 27
- Интерпретатор 100, 127, 174, 245, 289
- Интерпретация (interpretation) 34
  - изображений (scene analysis) 39
  - режим (interpretation) 73
- Интерфейс пользователя (human interface) 32
- Источник (source) ввода 189
- Код управляющий (directive) 185
- Компилятор кремниевый (silicon compiler) 40
- Константа (constant) 64, 95
- Конструктор (constructor) 312
- Контекст вычислительный (evaluation environment) 121, 291
- Лингвистика (linguistics) 32
  - математическая (computational linguistics) 32
- Лямбда-вызов 106
- Лямбда-выражение 105, 270
- Лямбда-исчисление (lambda calculus) 104
- Лямбда-преобразование (lambda conversion) 107
- Лямбда-список (lambda list) 105
- Макрознак (macro character) 177
- Макрос (macro) 289
  - вызов 128, 290
  - расширение (expansion) или
- раскрытие 290
- структуроразрушающий (destructive) 301
- трансляция (translation) 290
- чтения (read macro) 177
- Массив (array) 86, 349
  - объектов (obarray) 179
  - специализированный (specialized array) 350
  - универсальный (general arrays) 350
- Машина вывода (inference engine) 33
- Метка перехода (tag) 140
- Механизм возвратов (backtracking) 201
- Множество всех подмножеств множества (power set) 274
- значений (range codomain) 69
- определения (domain) 69
- Моделирование (modelling) 37
  - когнитивное (cognitive modelling) 38
  - работы (simulation) 37
- Морфология (morphology) 32
- Мусор (garbage) 159
- Мусорщик (garbage collector) 51, 160
- Мутатор (mutator) 312
- Наблюдение (monitoring) 34
- Наука когнитивная (cognitive science) 38
- Нотация списочная (list notation) 157
  - точечная (dot notation) 156, 157
- Область действия (scope) 121
- Обновление (update mutation) 312
- Обработка естественного языка (natural language processing) 31
  - знаний (knowledge processing) 22
  - изображений (image processing) 39
  - рисунков (picture processing) 39
  - сигналов (signal processing) 38
  - символьная (symbol manipulation) 24
  - символьной информации (symbolic processing/computing) 24
- Образ графический (icon) 187
- Образец (template skeleton) 300
- Обучение (instruction tutoring) 35

- языку (language learning) 32
- Объект (object) 311
  - данных (data object) 309
  - функциональный (functional object) 241
- Окружение вычислительное (evaluation environment) 121, 291
- Определение неявное (implicit) 309
  - функции (definition) 71
  - явное (explicit) 309
- Отмена блокировки замещающая 299
  - - присоединяющая 299
- Отображение (mapping) 69
- Пакет(package) 180
- Пара точечная (dotted pair) 84, 156
- Параметр ключевой (key) 112
  - необязательный (optional) 112
  - передача по значению (call by value) 119
    - - - ссылке (call by reference) 119
    - фактический (actual parameter) 106
    - формальный (formal parameter) 105
- Перевод машинный (machine translation) 32
- Переменная (variable) 64
  - внешняя (external) 180
  - внутренняя (internal) 180
  - вспомогательная (auxiliary) 112
  - глобальная специальная (global special/dynamic variable) 64
  - динамическая или специальная (dynamic/special variable) 121, 293
  - лексическая или статическая (lexical/static variable) 120
  - программная (program variable) 141
  - свободная (free variable) 121
- Печать структурная (prettyprint-ter) 67
- Планирование действий (planning) 34
- По умолчанию (default) 112
- Подсписок 65
- Поиск (search) 201
  - ошибки (diagnosis) 34
- Поле 149, 354
- Получатель (sink) вывода 189
- Последовательность (sequence) 257, 337, 340, 341
- Построение (construction) 312
  - прототипов быстрое (rapid prototyping) 53
- Поток (stream) 176, 187, 265
- Прагматика (pragmatics) 32
- Предикат (predicate) 79, 85
- Предложение (clause) 128
- Приглашение (prompt) 74
- Применение функции (apply) 71
- Присваивание (set) 96
- Прогнозирование (prediction) 34
- Программирование автоматическое (automatic programming program synthesis) 41
  - исследовательское (exploratory programming) 53
  - логическое (logic programming) 36, 52
  - объектно-ориентированное (object oriented) programming 38, 52, 244
  - пошаговое (incremental programming) 52
  - продукционное (rule-based programming) 52
  - процедурное (procedural programming) 195
  - ситуационное (event-based programming) 52
  - управляемое данными (data driven programming) 50, 244
  - функциональное (functional programming) 51, 196
- Проектирование (design) 35
  - машинное (computer aided design computer aided engineering) 39
- Производство гибкое (flexible manufacturing) 40
- Пространство имен (name space) 180
- Процедура доступа (access operator)

- accessor) 312
  - чтения (lisp reader) 176
  - Пустой список 65
  - Размер (size) 349
  - Размерность (dimensionality) 349
  - Распознавание образов (pattern recognition) 38
    - речи (speech recognition) 32
  - Рассуждения на уровне здравого смысла (common sence reasoning) 37
  - Режим EVALQUOTE 102
  - Рекурсия более высокого порядка 225, 232
    - взаимная (mutual) 224, 228
    - параллельная 224, 225
    - по аргументам 205
    - - значению 205
    - простая (simple) 205
  - Решение алгоритмическое (algorithmic) 25
    - эвристическое (heuristic) 25, 37
  - Робототехника (robotics) 39
  - Самоизменение (self-modification) 282
  - Самосознание (self-consciousness) 282
  - Свойство (property) 168
  - Связывание (bind) 96
    - параметров (spreading) 107
    - позднее (late binding) 54
  - Связь (binding) 96
    - время действия (extent) 122
  - Селектор (selector) 312
  - Семантика (semantics) 32, 101
  - Символ 61, 326
  - Синтаксис (syntax) 32
  - Синтез программ (program synthesis) 37
    - речи (speech synthesis) 32
  - Система поддержки принятия решений (decision support system) 33
    - самообучающаяся (learning system) 41
- экспертная (expert system) 23, 33
  - Слова ключевые (lambda-list keyword) 111
  - Сопоставление с образцом (pattern matching) 39
  - Список (list) 65
    - ассоциативный (association list a-list) 311, 331, 332
    - объектов (object list oblist) 179
    - нап 311, 331, 332
    - свободной памяти (free list) 160
    - свойств (property list p-list) 150, 168, 311, 326
    - - свободный (disembodied property list) 172
  - Строка (string) 309, 337
    - управляющая (control string) 185
  - Структура (structure/record) 311, 353
    - символьная 24, 61
  - Таблица чтения (read table) 177
  - Дело функции (body) 105
  - Теория рекурсивных функций 201
    - функций более высокого порядка 283
  - Технология знаний (knowledge engineering) 23
  - Тип (type) 70, 312
    - данных (data type) 309
    - - абстрактный (abstract data type) 310
    - составной (compound) 309, 312
  - Точка неподвижная (fixed point) 282
  - Транслирование по частям (incremental compiling) 53
  - Трассировка (trace) 206
  - Указатель (pointer) 149
  - Управление производством (control) 35
  - Уровень командный (top level) 102
  - Ускоритель вычислений с плавающей точкой (floating point accelerator) 319
  - Фонология (phonology) 31
  - Форма (form) 127

- самоопределенная (self-evaluating) 127
  - специальная (special form) 128
- Формулы рекуррентные (recurrence formula) 202
- Фунарг-проблема (funarg problem) 265, 267
- Функционал (functional) 194, 239
- применяющий или аппликативный (applicative functional) 245
- Функция 69, 240
- автоаппликативная (self-applicative auto-applicative) 243, 280
  - авторепликативная (self-replicative auto-replicative) 243, 280
  - более высокого порядка (higher order) 243
  - вызов (call) 71, 240
  - доступа (access function) 305
  - значение 69, 240
  - многозначная (multiple valued function) 115
  - общерекурсивная (general recursive) 203
  - определение (definition) 71
  - отображающая (mapping function) 249, 344
  - порядок (order) 284
  - применение (apply) 71
  - примитивно рекурсивная (primitive recursive) 202
  - рекурсивная (recursive) 142, 196
  - с функциональным значением (function valued) 241
  - структуроразрушающая (destructive) 161
  - тип 283
  - универсальная (universal function) 101, 245
- Хвост списка (tail) 80, 82
- Хэш-массив (hash array) 351
- Хэш-функция 352
- Цикл вложенный (nested) 229
- Черта вертикальная (bar) 62, 179, 328
- обратная косая (backslash) 62, 179, 327
- Число (number) 63, 319
- вещественное (real) 309
  - дробное (ratio) 323
  - комплексное (complex) 315, 323, 325
  - рациональное (rational) 323
  - с плавающей запятой (float) 323
  - целое (integer) 309
- Шрифт (font) 186
- Экземпляр (instance) 264, 309
- Элемент (element) 65
- Эффект побочный (side effect) 99, 301
- Язык аппликативный 194
- бестиповый (typeless) 54, 284, 313
  - встроенный (embedded language) 289
    - декларативный (declarative) 194
    - естественный (natural language) 23
    - императивный (imperative) 194
    - логический (logic programming) 194
    - операторный (imperative) 194
    - процедурный (procedural) 194
    - функциональный (functional) 194
- Ячейка памяти (storage location) 98
- списочная (memory cell list cell cons cell cons) 149
- MAP-функция 249, 344
- m-нотация (meta-notation) 115
- NLAMBDA-выражение 117
- PROG-механизм (prog feature) 139
- PROGN неявный (implicit progn feature) 132
- s-выражение (s-expression) 66

## Предисловие редактора перевода

С выходом в свет настоящей книги наша скучная литература по Лиспу (хотя можно было бы указать на [1,2] и некоторые другие источники) получает не просто описание одной из самых современных и наиболее популярных версий языка, Коммон Лисп, но и тщательно продуманное руководство по использованию этого языка, содержащее множество важных вспомогательных соображений.

Коммон Лисп, или общеупотребительный Лисп, явился результатом целенаправленной разработки такой версии языка, которая, при сохранении специфических для Лиспа черт, позволяющих с первого взгляда узнать программу, написанную на Лиспе, стал с точки зрения вычислительной эффективности вполне конкурентным с другими, более традиционными языками программирования. Коммон Лисп стал стандартом, пригодным для использования и на персональных вычислительных машинах, и на мощных мини-ЭВМ, и на специализированных Лисп-машинах, предлагая программисту услуги, объем которых расширяется с ростом класса машины, при этом обеспечивая разумную совместимость результирующих программ.

Потребность в унификации Лиспа возникла давно. Дело в том, что теоретический "чистый" Лисп – это очень ограниченный набор функций, позволяющий создавать и видоизменять произвольные списки, состоящие из элементов подсписков, путем добавления нового элемента (или подсписка) в начало списка или путем взятия головной части списка или соответствующего остатка. Используя предусмотренный в "чистом" Лиспе оператор условного перехода, в принципе, уже можно программировать на этом языке, но язык Лисп получил широкое распространение благодаря тщательно разработанной и богатой библиотеке уже не столь элементарных операций. Поэтому с момента создания этого языка Дж. Маккарти в 1962 г. число вариантов, или версий, этого языка стало исключительно большим.

Необходимость создания некоторой стандартной версии ощущалась и в университетах, где создавались очень большие программы на этом языке, которые объединялись в еще более крупные системы посредством сетей передачи данных типа ARPANET, а затем в промышленных и военных фирмах. В последних в качестве стандартов приняты два языка программирования. Для вычислительных задач это язык Ада, а для невычислительных задач и задач искусственного интеллекта, в частности, — Коммон Лисп. Здесь следует заметить, что и в японском проекте ЭВМ пятого поколения в качестве базового языка наряду с Прологом был выбран также и Лисп.

Трудно однозначно сказать, почему предпочтение было отдано именно Коммон Лиспу, хотя на роль стандарта в течение многих лет претендовал Интерлисп [3] — коллективный труд многих опытных программистов — сторонников этого языка. Может решающим фактором оказалась вычислительная эффективность Коммон Лиспа, может быть настойчивость его создателей и сопутствующая им удача, а может быть солидная поддержка, которая была оказана этой версии Лабораторией искусственного интеллекта Массачусетского технологического института (США), в недрах которой зародилась и была создана так называемая Лисп-машина с Коммон Лиспом в качестве системного языка программирования.

Большинству программистов не надо представлять такой известный язык программирования, каким является Лисп, тем более, что некоторые из них уже прочно связали с ним свою деятельность.

Дж. Саммит, известный специалист по языкам программирования, как-то сказала, что все языки программирования можно грубо разбить на два класса. В одном находится Лисп, в другом — все остальные языки программирования.

В Лиспе, например, процедуры могут служить в качестве "данных", подставляемых на место аргументов. В результате они могут вернуться в качестве значений функций, запоминаемых снова в качестве данных.

Первоначально Лисп был задуман как теоретическое средство для рекурсивных построений, а сегодня он превратился в мощное средство, обеспечивающее программиста разнообразной поддержкой, позволяющей ему быстро строить прототипы весьма и весьма серьезных систем.

Профессор Массачусетского технологического института Дж.Самман заметил, что математическая ясность и предельная четкость Лиспа – это еще не все. Главное – Лисп позволяет сформулировать и запомнить "идиомы", столь характерные для проектов по искусственноому интеллекту.

Создание и отработка таких замкнутых подсистем, решающих важные отдельные подзадачи, и позволили, на наш взгляд, в дальнейшем строить крупные, организованные на локальных принципах системы, обладающие целесообразным поведением, рожденным совместным функционированием отдельных целесообразных подсистем [4].

Сегодня не приходится убеждать в достоинствах языка Лисп. Об этом красноречиво говорят огромные объемы программ, созданных в области искусственного интеллекта и экспертных систем. На Лиспе сегодня пишут и программы, не относящиеся прямо к области искусственного интеллекта. Например, набор редакторов GNU Emacs для различных языков программирования, используемых на ЭВМ семейства VAX, написан на Лиспе. На этом же языке написано все математическое обеспечение системы инженерного проектирования AUTOCAD. С появлением Лисп-машин язык Лисп стали использовать и для вычислительных задач. В работе [5], посвященной описанию способа применения рекурсии при вычислении значения функции непрерывного аргумента, отмечен также пример моделирования на Лиспе движения робота с учетом механических свойств используемых материалов.

Но широкое распространение Лиспа самым существенным образом зависит от наличия адекватных руководств по этому языку. Парадоксально, что "авторское" описание языка [6] было настолько неудач-

ным, что оттолкнуло от него многих потенциальных пользователей на несколько лет.

Новичков обычно пугает необходимость использования в этом языке большого числа скобок. Здесь можно порекомендовать только одно – попробуйте подойти к терминалу и составить программу на Лиспе. Вы убедитесь, что принятая в этом языке скобочная префиксная запись является одной из наиболее привлекательных черт этого языка, превращающих программирование в удовольствие. А за правильным соотношением числа скобок в современных версиях Лиспа следит сама машина! Благодаря скобкам тексты на Лиспе сравнительно легко читаются, облегчается понимание смысла программ.

Но чтобы оценить эти и многие другие возможности потребовался прекрасный учебник Вейсмана [7], к сожалению, своевременно не переведенный на русский язык. Именно с этой книги началась пора серьезного увлечения Лиспом на Западе, где многие термины, родившиеся в этом языке, вошли в сленг программистов.

Сходная, на наш взгляд, судьба постигла и авторское описание Коммон Лиспа [8], которое является столь полным, столь подробным и столь аргументированным, что требуется какое-то дополнительное толкование, чтобы пользоваться этим руководством на практике. Объясняется это тем, что это описание ориентировано прежде всего на специалистов, хорошо знакомых с миром Лиспа. Сама по себе эта книга может снова создать впечатление, что Лисп – это что-то очень сложное и специальное.

Рекомендуемый советскому читателю перевод книги финских ученых Хювёнена и Сеппянена, давно и плодотворно занимающихся Лиспом, – одно из немногих руководств по Коммон Лиспу, которое является простым и понятным даже для новичка. И хотя это руководство не заменяет книгу [8] полностью, предлагаемый перевод дает описание всех основных конструкций Коммон Лиспа, а также большое число примеров, позволяющих выработать навык в пользовании этим языком.

То, что книга написана финскими учеными, наложила свой отпечаток в выборе некоторых примеров, в части которых и в русском переводе читатель увидит "странные" буквы финского алфавита и некоторые рассуждения о том, как их поддерживать в Коммон Лиспе, изначально рассчитанном на английский язык. Эти примеры сохранены не только из уважения к авторам, которые, кстати, давно проявляют профессиональный интерес к средствам общения с ЭВМ на различных языках, на которых пишут народы мира. Другая причина состоит в том, что и в русском языке встречаются "необычные" буквы, поддержка которых в Коммон Лиспе также порождает определенные трудности. Впрочем мы готовы здесь поделиться опытом, накопленным в нашей практике использования языка Лисп.

В целом мы с удовольствием рекомендуем читателю эту своевременную и нужную книгу. Перевод книги выполнен специалистом по языку Лисп из Эстонии, А. Рейтсакасом. Вместе с переводчиком мы старались обнаружить и устранить отдельные ограхи оригинала, не пытаясь добавить при этом новых. Вряд ли нам это полностью удалось, поэтому мы будем очень рады получить от читателей соответствующие замечания и предложения. Эти замечания будут использованы в случае последующих переизданий книги.

### Литература

1. Лавров С.С., Силагадзе Г.С. *Автоматическая обработка данных. Язык лисп и его реализация.* - М.: Наука, 1978.
2. Уинстон П. *Искусственный интеллект.* -М.: Мир, 1980, с.303-512.
3. Teitelman W. *INTERLISP Reference Manual.* Xerox Palo Alto Research Center, 1974.
4. Стефанюк В.Л. *Анализ целесообразности локально-организованных систем через потоки вероятности.* В сб.: *Модели в системах обработки данных.* -М.: Наука, 1989, с.33-45.

5. Стефанюк В.Л. *Рекурсивное оценивание арифметических функций в системах ЛИСП*, "Программирование", №5, 1981, с.92-94.
6. McCarthy J. (with Abrahams, Edwards, Hart, Levin) *LISP 1.5 Programmer's Manual*. -MIT Press, Cambridge, Mass, 1962.
7. Weissman C. *LISP 1.5 Primer*, Dickenson Publishing Company, Belmont, Calif, 1967.
8. Steele G.L. *Common Lisp – the language*, Massachusetts: Digital Press, 1986.



*Опасность не в том, что ма-  
шина начинает уподобляться  
человеку, а в том, что человек  
превращается в подобие ма-  
шины.*

*Сидней Харрис*

## **ВВЕДЕНИЕ**

- С скачком в развитии вычислительной техники
- Исследовательские программы искусственного интеллекта
- Национальные программы по исследованию языков
- Появление Лиспа в Финляндии
- Лисп – основа искусственного интеллекта
- Учебник Лиспа на финском языке
- Язык Лисп и функциональное программирование
- Методы и системы программирования
- На кого рассчитана книга
- Терминология
- Иконология
- От дерева к мысли и от мысли к дереву
- Благодарности

### **С скачком в развитии вычислительной техники**



Цель данной книги – дать читателю представление о языке программирования Лисп, символьной обработке и о мире искусственного интеллекта. Вычислительная техника нового поколения, базирующаяся на этих идеях, открывает дорогу новому технологическому и промышленному скачку. С помощью искусственного интеллекта, использования естественных языков и экспертных систем вычислительная техника в состоянии справиться с задачами, решение которых раньше считалось прерогативой человека.

Вычислительные машины и автоматы позволяют механизировать решение как задач рутинного характера, так и интеллектуальных проблем. Развитие вычислительной техники происходит скачкообразно. С скачком, наблюдаемым в данный момент, качественный и более глубокий по своему характеру, чем все предыдущие. Системы обработки данных вторгаются в области, традиционно считавшиеся сферой деятельности человека, где для решения различных проблем требовались компетентность и знания, имеющиеся у человека.

### **Исследовательские программы искусственного интеллекта**

Стало возможным применение на практике результатов лабораторных исследований искусственного интеллекта. Искусственный интеллект и технология знаний начинают играть первостепенную роль в промышленности и экономике. В 80-е годы для изучения интеллектуальных систем, их развития и применения во всем мире были объявлены различные национальные и международные программы исследований.

#### **ICOT**

新世代コンピュータ

Японцы собираются первыми представить к 1990 г. демонстрационные образцы интеллектуальных вычислительных машин, которые будут решать задачи на уровне компетентности "близком к человеческому", понимать естественный язык и т. д. Для достижения поставленной цели основан исследовательский центр ICOT и под руководством Министерства внешней торговли и промышленности (MITI) начата разработка широко обсуждавшегося в течение последних 10 лет проекта "Вычислительных машин пятого поколения".

США ответили на вызов созданием совместного исследовательского центра производителей больших вычислительных машин MCC и государственной программой исследований агентства DARPA Министерства обороны США. В Европе страны Общего рынка под руководством Франции начали исследования по программам ESPRIT и EUREKA, и, кроме этого, во многих

европейских странах разработаны свои национальные программы (английская программа Alwey и др.).

### **Национальные программы по исследованию языков**

С программами развития вычислительной техники связаны начатые в различных странах широкие национальные и международные проекты исследования языков, на которых говорят люди. Их цель – сделать возможным использование естественных или близких к ним языков в различных сферах применения вычислительных машин, вплоть до понимания машиной фраз естественного языка и осуществления автоматического перевода с языка на язык. Во многих странах языковые проекты рассматриваются как важнейшие с национальной точки зрения направления в области развития современной компьютерной культуры. Конечной целью различных проектов является разработка интеллектуальной технологии, с помощью которой человеческие знания и способности станут новым источником технической и промышленной мощи, экономическим и культурным достоянием нации.

### **Появление Лиспа в Финляндии**



В 80-е годы в Финляндии исследования в области искусственного интеллекта стали объектом повышенного интереса. В 1983 г. в Техническом университете был выполнен обзор проводимых в Финляндии исследований в этой области, и через год были организованы первые Дни исследования искусственного интеллекта (STeP-84) в Отаниеми. В следующем году были

начаты исследовательские программы по искусственно-му интеллекту как в Государственном техническом исследовательском центре, так и в Центре по разработке технологий (TEKES). В исследованиях в области финского языка участвуют проекты, финансируемые как со стороны Финской академии, так и со стороны

**SITRA.** В университетах и высших учебных заведениях расширено изучение и исследование этой ранее мало изучаемой области. В промышленности также начата переподготовка и объявлены новые научно-конструкторские разработки, относящиеся к области искусственного интеллекта.

### Лисп – основа искусственного интеллекта

Лисп – важнейший язык, используемый в символьной обработке и в исследованиях по искусственному интеллекту. Эти работы, начатые в США в МИТ уже в середине 50-х годов, проводились преимущественно на языке Лисп. Большая часть имеющихся на рынке программ символьной обработки, систем искусственного интеллекта и программ работы с естественным языком написаны на Лиспе. Многие методы, используемые в области искусственного интеллекта, основаны на особых свойствах языка Лисп. Лисп составляет основу для обучения методам искусственного интеллекта, исследованиям и практическому применению этой области, иными словами, Лисп вводит в мир символьной обработки и искусственного интеллекта.

### Учебник Лиспа на финском языке

В последнее время за рубежом вышло много литературы, посвященной Лиспу, в основном, естественно, на английском языке, но также и на других языках. Долгое время распространению Лиспа и обучению технике искусственного интеллекта препятствовало отсутствие в Финляндии учебника на финском языке. На самом деле первый учебник по языку Лисп, финансированный OtaData, появился в начале 70-х годов, но он так и остался единственным, уже давно распродан и устарел по содержанию<sup>1)</sup>.

<sup>1)</sup> Читатель заметит, что в СССР сложилась аналогичная обстановка. – Прим. ред.



Надеемся, что выход данного двухтомника существенно улучшает положение. При написании книги мы попытались учесть всю опубликованную на данный момент литературу по Лиспу, особенно учебную и весь накопленный опыт по Лиспу и искусственноому интеллекту. В этой книге отражены также наш личный опыт программирования на Лиспе, сведения, содержащиеся в учебной литературе и в других изданиях, касающихся Лиспа, а также на прочитанных нами в разных местах курсах по Лиспу. В основу книги положены лекции, прочитанные в 1981 г. на отделении общего языкознания Хельсинского университета, и серия из 11 статей, опубликованных в журнале "Процессори" (Процессор) в 1982-1983 годах, а также материалы некоторых других наших публикаций. Рабочая версия книги и первое издание, финансированное Knowledge Engineering Ky, были использованы в качестве учебника в учебном центре вычислительной техники фирмы Nokia (NITEC) и на курсах Лиспа Artificial Intelligence Systems Oy.

### Язык Лисп и функциональное программирование

Для ознакомления с миром Лиспа в первой части книги исследуются важнейшие сферы применения символьной обработки и искусственного интеллекта и отличия языка Лисп от других языков программирования. Вслед за этим изложен сам язык Лисп, связанные с ним понятия, а также языковые конструкции и механизмы. Основные методы программирования на Лиспе – функциональное и рекурсивное программирование – рассматриваются более детально и систематично с подразделением типов функций и форм рекурсии на отдельные виды, что является новшеством по сравнению с предшествующими публикациями по Лиспу.

## **COMMON**

## **LISP**

Первая часть книги может использоваться для изучения как языка Лисп, так и функционального программирования.

Основой изложения нами был выбран диалект Коммон Лисп, ставший "де-факто" промышленным стандартом языка Лисп. В книге представлены все

важнейшие языковые формы и свойства конструкций Коммон Лиспа, а также типы функций и данных. Материал изложен не в виде справочного руководства, а в логически последовательной и поэтому пригодной для обучения форме. Даются пояснения для рассматриваемых в книге понятий и методов, которые в справочных руководствах обычно не освещаются. В то же время ограничено количество трудночитаемой системной технической информации. Чтобы можно было использовать книгу и как справочное руководство, в конце ее приведено краткое описание всего Коммон Лиспа и словарь символов.

Однако изложение не ограничивается рамками Коммон Лиспа. По мере надобности приводятся сведения о важнейших свойствах и особенностях других систем и расширениях стандарта. Например, объекты и объектно-ориентированное программирование, которое сейчас (1985) еще не входит в состав Коммон Лиспа, представлены здесь в том виде, как они реализованы в системе Flavor среды Зеталисп.

### Методы и системы программирования

Во второй части книги наряду с функциональным программированием изложены основные методы, применяемые в решении задач искусственного интеллекта. Среди них операторное и процедурное программирование, программирование, управляемое данными, сопоставление с образцом, ситуационное и продукционное программирование, а также объектно-ориентированное и логическое программирование. Текст дополнен примерами практического программирования.

В части, касающейся среды программирования, рассмотрены содержащиеся в Коммон Лиспе средства и окружение. Стандарт Коммон Лиспа<sup>1)</sup> рассматривает вопросы, связанные со средой программирования лишь вскользь. Однако предоставление разви-



<sup>1)</sup> См. в литературе книгу Steele "Common Lisp". – Прим. ред.

тых интегрированных сред программирования являются одним из важнейших преимуществ Лиспа перед традиционными языками и системами программирования. Дав понятие об этих возможностях, мы кратко представим также средства сред программирования Интерлиспа и Зеталиспа.

"Мир Лиспа" – не только просто учебник по языку и программированию, он рассматривает Лисп-культуру, методы искусственного интеллекта и их развитие в более широком плане. В конце каждого посвященного методам программирования раздела приводится обзор возникновения связанных с обсуждаемым методом понятий и идей и их связей с другими областями вычислительной техники. В отдельной главе дается представление о развитии языка, начиная с ранней истории до современных систем, в том числе и о Лисп-машинах. Рассматриваются также и предлагаемое японцами пятое поколение вычислительных машин и другие новые направления развития архитектуры вычислительных машин.

### **На кого рассчитана книга**

Книга рассчитана на учащихся и исследователей как в области технических, естественных, так и гуманитарных специальностей, а также для специалистов по вычислительной технике и для занятых в других близких к ней сферах. Надеемся, что эта книга будет иметь широкую сферу применения и использоваться как в качестве учебника в университетах и высших школах, так и в качестве материала курсов повышения квалификации специалистов, работающих в промышленности, и для самостоятельного обучения языку Лисп, а также будет полезна специалистам и всем интересующимся проблематикой искусственного интеллекта.

### **Терминология**

В вычислительной технике время от времени рождаются новые понятия и термины, для которых приходится искать или создавать аналоги на других языках, в

частности на финском языке. Обычно в вычислительной технике проблемы именования наиболее сложны в быстро развивающихся областях, где еще не устоялась терминология на языке-оригинале. Особенно много



новых понятий требуется и создается в программировании и технике искусственного интеллекта, где сам по себе вопрос в основном сводится к определению и формированию понятий.

Мы попытались найти для английских терминов как можно более точные и практические соответствия на финском языке. В какой степени это удалось, судить читателям.

Оригинальный термин дан в скобках в момент его определения или разъяснения. Термины на русском<sup>1)</sup> и английском языках собраны в приложении. Выбранные соответствия, конечно, не являются наилучшими. Поэтому мы с благодарностью примем все предложения читателей, касающиеся терминологии, а также другие замечания и предложения по содержанию книги и изложению материала.

### Иконология

Для иллюстрации и лучшего восприятия текста мы использовали взятые из различных источников и



украшенные юмором символическую графику и афоризмы. В тексте и на картинках часто повторяющимся мотивом является дерево, которое символизирует как логические и физические конструкции Лиспа, так и воплощаемые в них деревья данных и модели мира. Дерево мира, дерево данных и дерево жизни – центральные символы

картины мира, знаний, жизни, счастья и благополучия

<sup>1)</sup>

В оригинале, конечно, на финском, но переводчик и редактор данной книги пошли еще дальше и попытались предложить соответствующую терминологию на русском языке. – Прим. ред.

во многих странах и культурах. И в "Калевала" есть большой дуб:

Если кто там поднял ветку,  
Тот нашел навеки счастье;  
Кто принес себе верхушку,  
Стал навеки чародеем;  
Калевала 2:191-194<sup>1)</sup>

Дерево всегда широко использовалось в технике и культуре. Из дерева делались различные предметы обихода, в том числе и вспомогательные средства, и инструменты для работы, с помощью которых снова изготавливались новые предметы, машины и более совершенные рабочие инструменты. В трудные времена древесные отходы добавлялись и в хлеб. На изготовленной из дерева бумаге частично базируется и современная письменная культура, в том числе и эта книга. Так и в Лиспсе символные деревья являются как инструментом и средством программирования, так и обрабатываемым программами материалом, из которого строятся различные применения, создаются системы и более развитые инструменты.

### **От дерева к мысли и от мысли к дереву**

На дереве в его различных формах основывается также финская хозяйственная жизнь. Однако все более важным товаром становятся наши знания и компетентность, новые возможности применения и продажи которых в виде экспертных систем открывают методы искусственного интеллекта. Говорят, что центр тяжести промышленности перемещается "от дерева к мысли". В этой книге представлены те основы, с помощью которых человеческие знания и компетентность могут быть перенесены в (лисповское) дерево в этом вновь возникающем, дружественном окружающей среде "деревообрабатывающем производстве".

---

<sup>1)</sup> Перевод Л. Бельского.

## Благодарности

В процессе подготовки и написания книги нам много раз в различной форме и в связи с различными обстоятельствами оказывали помощь Орри Эрлинг, Хейки Хонкио, Ниссе Хасберг, Еийя Ярвинен, Кари Каллио, Матти Карьялайнен, Фред Карлссон, Урпо Карьялайнен, Антеро Катайнен, Суви Кеттула, Ора Лассила, Кари Маннерсало, Клаус Оэш, Пекка Пиринен, Реййо Сулонен, Маркку Сюрьянен, Перtti Тапола, Пекка Толонен, Матти Виртанен и Мартти Юлестало. Мы хотим выразить признательность за помощь в работе Юхани Карвинену и Аннели Карпинен из издательства "Кирьяюхтюмя", а также Ханну Ромппанену из "Гуммеруксена".

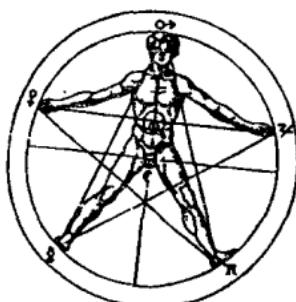
Подготовка и редактирование книги стали возможными благодаря содействию, оказанному Техническим университетом и Финской академией наук. Digital



Equipment Corporation Oy, Knowledge Engineering Ky, Nokia Informaatiojärjestelmät (Информационные системы Нокия), Rank Xerox, Texas Instruments и Министерство образования предоставили машинное время, устройства и соответствующие материалы к ним. Благодарим Технический университет и Министерство образования за оказанную нам финансовую поддержку.

**Отаниеми**

**Авторы**



*Микрокосмос человека.  
Магический знак.*

# **1 ВВЕДЕНИЕ В МИР ЛИСПА**

## **1.1 СИМВОЛЬНАЯ ОБРАБОТКА И ИСКУССТВЕННЫЙ ИНТЕЛЛЕКТ**

## **1.2 ПРИМЕНЕНИЯ ИСКУССТВЕННОГО ИНТЕЛЛЕКТА**

## **1.3 ЛИСП – ЯЗЫК ПРОГРАММИРОВАНИЯ ИСКУССТВЕННОГО ИНТЕЛЛЕКТА**

*В этой главе мы вступим в мир символьной обработки, искусственного интеллекта и Лиспа. Покажем, чем символьная обработка и программирование в области искусственного интеллекта отличаются от числовых вычислений и традиционной обработки данных. Познакомимся с основными сферами применения искусственного интеллекта и технологии знаний, в частности с такими, как обработка естественного языка, экспертные системы, вычисления в символьном виде и логика, машинное зрение, робототехника, автоматическое программирование и т. д. После этого рассмотрим, почему обычные языки программирования не годятся для программирования задач искусственного интеллекта, а используются языки, подобные Лиспу, специально созданные для работы с символами и конструкциями. Кроме того, мы увидим, почему Лисп подходит для программирования задач искусственного интеллекта, почему он превратился в основной язык, используемый в этой области, почему он стал широко применяться только в 80-е годы и почему будущее Лиспа еще впереди, хотя он создан в 50-е годы.*

*Начни – и полдела сделано.  
Начни еще раз – и сделано  
уже все.*

*Авсоний*

## 1.1 СИМВОЛЬНАЯ ОБРАБОТКА И ИСКУССТВЕННЫЙ ИНТЕЛЛЕКТ

- Искусственный интеллект и технология знаний
- Исторические предубеждения
- Символьное или численное вычисление
- Эвристическое или алгоритмическое решение задачи
- Искусственный интеллект – сфера исследования многих наук
- Знание дела и умение как товар
- Учебники по искусенному интеллекту
- Упражнения

В 50-е годы вычислительные машины использовались в основном для численных вычислений в научных



исследованиях. В 60-е годы их научились применять для выполнения рутинной деятельности в экономике и управлении, начали говорить о работе с "данными" и об автоматической обработке данных. В 70-е годы вычислительные машины стали использоваться в новых областях, и обработка данных проникала во все новые сферы. Вычислительные машины начали использоваться для управления и как вспомогательное средство в управлении деятельностью целых организаций. Ключевыми словами десятилетия стали системы данных и информационные системы, управление данными, передача данных и т. д.

В 80-е годы возникает новый этап. Начались разговоры об обработке знаний (knowledge processing) и искусственном интеллекте, пятом поколении вычислительных машин и новой волне вычислительной техники. Ключевыми словами десятилетия становятся

экспертные системы (*expert systems*), естественный язык (*natural language*) и речь, интеллектуальные программы и применения, искусственный интеллект.

### Искусственный интеллект и технология знаний

*Искусственный интеллект* (*artificial intelligence, machine intelligence*) – это область исследований, находящаяся на стыке наук, специалисты, работающие в этой области, пытаются понять, какое поведение,



считается разумным (анализ), и создать работающие модели этого поведения (синтез). Исследователи ставят задачу с помощью новых теорий и моделей научиться понимать принципы и механизмы интеллектуальной деятельности. Практической целью является создание мето-

дов и техники, необходимой для программирования "разумности" и ее передачи вычислительным машинам, а через них всевозможным системам и средствам. Инженерные методы и навыки в области искусственного интеллекта стали называть *технологией знаний* (*knowledge engineering*).

### Исторические предубеждения

Исследования в области искусственного интеллекта начались уже в середине 50-х годов. На начальной стадии развития на эту область возлагались, пожалуй, слишком большие надежды. Исследования были направлены на разработку механизмов решения общих проблем произвольного характера. Из-за слишком самонадеянных целей возникли разочарование и предубеждения по отношению к самой области и кенным в ней исследователям. Например, в США официально были признаны бесперспективными исследования в области машинного перевода (доклад Alpac).

С созданием в конце 60-х и в 70-х годах первых экспертных систем (*DENDRAL, MACSYMA, CASNET/GLAUCOMA, MYCIN* и т. д.) все же было замечено, что сосредоточиваясь на решении проблемы в некото-



рой узкой области и пытаясь решить ее с помощью знаний из этой специальной области, а не с помощью общих механизмов принятия решений, можно достичь лучших результатов.

В 80-е годы искусственный интеллект испытал второе рождение, были широко осознаны его большие потенциальные возможности как в исследованиях, так и в развитии производства. В рамках новой технологии появились первые коммерческие программные продукты. В начале десятилетия в различных странах были начаты крупнейшие в истории обработки данных национальные и международные исследовательские проекты, нацеленные на интеллектуальные вычислительные машины "пятого поколения".

### Символьное или численное вычисление

Программы искусственного интеллекта отличаются от традиционных программ тем, что в них в первую очередь обрабатываются данные, представленные в символьном, а не в числовом виде. Исследователи в этой области заметили уже в 50-е годы, что присущие



*Стол для вычислений. Средневековая гравюра.*

человеческому мышлению и языку или реальному миру вещи, объекты, ситуации, случаи и другие понятия нельзя естественным образом отобразить в виде чисел и массивов. В программах искусственного интеллекта нужна возможность работать с данными и знаниями, представленными в символьном виде или в виде символьных структур. В отличие от традиционного численного вычисления (numerical computing) в

программировании искусственного интеллекта говорится об обработке символьной информации (symbolic processing/computing) или о символьной обработке (symbol manipulation).

В обработке символьной информации важна не только форма рассматриваемых знаний, но и их

*содержание и значение.* Если в численных вычислениях внимание уделяется *количественной стороне* и численным значениям данных, то в символьной обработке важна *качественная сторона* данных, т.е. особенности их строения и их функциональные особенности. Для численных вычислений характерно большое количество простых действий (вычисление, присваивание, вывод) над большим числом простых элементов (биты, числа, строки и т.д.). А в программах искусственного интеллекта обрабатываются особенно большие и сложные структуры данных и описываются сложные действия, но количество вводимых и выводимых данных в большинстве случаев невелико, например предложение на естественном языке, описание деятельности и т.д.

### **Эвристическое или алгоритмическое решение задачи**

Другое отличие программирования искусственного интеллекта от традиционного состоит в том, что в программах искусственного интеллекта проблема решается часто *эвристически* (*heuristic*), в то же время традиционное программирование основывается главным образом на *алгоритмическом* (*algorithmic*) способе.

При алгоритмическом решении задачи в различных ситуациях, требующих принятия решения, как правило, достаточно информации, чтобы сделать верный выбор. Решение будет найдено всегда, когда оно вообще возможно.



Но когда имеющиеся в наличии данные не достаточны или не внушают доверия или когда другие способы не пригодны для принятия решения, приходится прибегать к эвристическим методам. Они обычно основываются на связанных с задачей специальных знаниях, простейших правилах, интуитивных критериях, базирующихся на предыдущем опыте и на других ненадежных методах вплоть до угадывания. Если выводы, сделанные на основе имеющихся знаний, признаются ошибочными, то они аннулируются, полученные результаты анализируются

## Способ решения

### Алгоритмический Эвристический

	Алгоритмический	Эвристический
Численное	Научно-технические вычисления	Моделирование, обработка сигналов
Символьное	Экономико-хозяйственная обработка данных	Искусственный интеллект, технология знаний

### Представление данных

*Рис. 1.1.1 Представление данных и способы решения проблем с точки зрения традиционной обработки данных и искусственного интеллекта.*

и решение задачи продолжается каким-нибудь другим способом. Эвристические методы не всегда приводят к цели, даже когда решение существует, или они могут привести к неверному решению. В этом отношении программы искусственного интеллекта подобны человеку.

Для многих задач искусственного интеллекта и технологии знаний алгоритмические решения неизвестны. К эвристикам обращаются в том случае, когда специалисты не в состоянии представить точные данные или когда их трактовки отличаются или даже противоречат друг другу. К тому же существуют задачи, для которых можно показать, что они не имеют алгоритмического решения.

Эвристики используют кроме случаев, связанных с характером задачи, также и в связи с ограничениями, налагаемыми вычислительной техникой. Средства и методы решения задачи могут быть известны, но их



*Неопределенность.  
Исландский рунический знак.*

использование потребовало бы большого объема работы. Эвристики используют тогда, когда время или какой-нибудь другой параметр алгоритмической процедуры неприемлемо велик.

В программировании задач искусственного интеллекта используются все алгоритмические методы, которые только возможны, и таким образом пытаются найти алгоритмическое решение проблемы. Однако такая попытка не должна стать препятствием на пути решения. Если даже не пытаться что-либо сделать, то ничего и не получится – это старая истина. Во многих случаях на основе эвристических решений потихоньку изобретаются алгоритмы.

Естественно, и эвристические программы в своей реализации базируются на алгоритмах.

### **Искусственный интеллект – сфера исследования многих наук**

Искусственный интеллект рожден в области вычислительной техники, но как наука он находится на пересечении информатики, языкоznания, психологии и философии. Математика, логика и многие теоретические направления информатики тесно связаны с исследованиями в области искусственного интеллекта. Технология знаний – это работающий искусственный интеллект. В сферах применения искусственного интеллекта используются также и конкретные специальные знания из соответствующей области, например из области естественных наук, медицины, юриспруденции, экономики и т. п.

### **Знание дела и умение как товар**

Вычислительная техника традиционно приспособлена прежде всего для автоматизации рутинной деятельности и увеличения эффективности, например в управлении процессами, данными, обработке текстов, вычислениях и т. д. Эта область деятельности предлагает специальные вспомогательные средства, с помощью которых человек может решить определенные задачи.



*Рис. 1.1.2 Искусственный интеллект объединяет различные области науки.*

Технология знаний непосредственно применяется к решению сложных проблем, направленных на увеличение производительности и качества умственного труда, на поднятие профессиональных навыков работника. Продуктом технологии знаний являются активные знания и навыки работы со знаниями, которые отличаются от пассивных книжных знаний тем, что их использование (в оптимальном варианте) не предполагает чтения и освоения всех необходимых для решения проблемы знаний.

Искусственный интеллект и технологию знаний нужно рассматривать как расширение и обобщение традиционной вычислительной техники, а не как альтернативу ей.

#### **Учебники по искусственному интеллекту**

1. Banerji R. *Artifical Intelligence. A Theoretical Approach*. North-Holland, New York, 1980.
2. Barr A., Cohen P., Feigenbaum E. (Eds.) *The Handbook of Artifical Intelligence*, Vol. 1–3. Pitman, London, 1981–1982.
3. Bundy A. *Artificial Intelligence. An Introductory Course*. Edinburgh University Press, 1978.

4. Charniak E., McDermott D. *Introduction to Artificial Intelligence*. Addison-Wesley, Reading, Massachusetts, 1984.
  5. Krutch J. *Experiments in Artificial Intelligence for Small Computers*. Howard W. Sams et Co., Indiana, 1981.
- 

Nilsson N. *Principles of Artificial Intelligence*. Tioga, Palo Alto, California, 1980. [Имеется перевод: Нильсон Н. Принципы искусственного интеллекта. -М.: Радио и связь, 1985.]
7. O'Shea T., Eisenstadt M. (Eds.) *Artificial Intelligence. Tools, Techniques, and Application*. Harper et Row, New York, 1984.
  8. Raphael B. *The Thinking Computer*. Freeman, San Francisco, 1976. [Имеется перевод: Рафаэл Б. Думающий компьютер. -М.: Мир, 1979.]
  9. Rich E. *Artificial Intelligence*. McGraw-Hill, New York, 1983.
  10. Winston P. *Artificial Intelligence*. Addison-Wesley, Reading, Massachusetts, 1983. [Имеется перевод: Уинстон П. Искусственный интеллект. -М.: Мир, 1980.]
  11. Winston P., Brown R. (Eds.) *Artificial Intelligence: An MIT Perspective*. Vol. 1-2, MIT Press, Cambridge, Massachusetts, 1979.

### Упражнения



Зачем, чтобы сделать вычислительные машины более дружественными и разумными, необходимы символьная обработка и эвристические процедуры?

2. Оцените, каким образом искусственный интеллект и технология знаний изменят область обработки данных?

*Искусственный интеллект за-  
менил мне мою природную  
глупость.*

*Автор неизвестен*

## **1.2 ПРИМЕНЕНИЯ ИСКУССТВЕННОГО ИНТЕЛЛЕКТА**

- **Многообразие форм искусственного интеллекта**
- **Обработка естественного языка**
- **Экспертные системы**
- **Символьные и алгебраические вычисления**
- **Доказательства и логическое программирование**
- **Программирование игр**
- **Моделирование**
- **Обработка сигналов и распознавание образов**
- **Машинное зрение и обработка изображений**
- **Робототехника и автоматизация производства**
- **Машинное проектирование**
- **Языки и средства программирования искусствен-  
ного интеллекта**
- **Повышение производительности программирования**
- **Автоматическое программирование и обучение**
- **Литература**
- **Упражнения**

Потенциальные возможности применения искусственного интеллекта и технологии знаний кажутся почти безграничными. Эта технология используется, когда необходимо передать вычислительной машине какие-нибудь способности к рассуждению, разумность, мудрость, творческие или другие свойственные человеку способности и когда нужно создать для них действующую теорию или модель.

**Многообразие форм искусственного интеллекта**  
В исследованиях по искусственному интеллекту, осуществляемому в различных областях, используется

много общих приемов программирования, способов представления данных, механизмов рассуждения и других методов. Например, эвристическое решение задач и символные вычисления характеризуют почти все исследования и методы программирования в области искусственного интеллекта. Однако никакой общей теории интеллектуальности или общего метода решения проблем в этой области найдено не было. Как область науки искусственный интеллект и технология знаний представляют собой еще весьма разрозненный набор попыток разработать модель "разумности" и разумные машинные системы.



Исследования по искусственному интеллекту часто классифицируются, исходя из области их применения, а не на основе различных теорий и школ. В каждой из этих областей на протяжении десятков лет разрабатывались свои методы программирования, формализмы; каждой из них присущи свои традиции, которые могут заметно отличаться от традиций соседней области исследования. Далее мы вкратце рассмотрим символную обработку и наиболее важные области исследования и применения искусственного интеллекта, давая обзор разнообразных постановок проблем и широких возможностей применения в каждой области.

### Обработка естественного языка

Обработка естественного языка (*natural language processing*) позволяет вести диалог между машиной и человеком на обычном или близком ему языке (например, на финском или английском). Это необходимо, поскольку с дальнейшим развитием автоматизации все большее число людей должны будут иметь дело со все более часто встречающимися системами обработки данных и их применение. Лучше пусть машины научатся человеческому языку, чем люди будут учить такие технические языки, как машинные языки, языки программирования и т. п.

Основными областями в обработке естественных языков являются фонология (*phonology*), изучающая

фонемы и их различные признаки, морфология (*morphology*), изучающая изменения слов, синтаксис (*syntax*), изучающий строение предложений, семантика (*semantics*), изучающая значение различных частей и структур языка, прагматика (*pragmatics*), изучающая использование языка и анализ дискурса (*discourse*), изучающий большие целостные фрагменты и языковые ситуации, как, например, диалог. Исследования в этих областях относятся к *лингвистике* (*linguistics*), или к общему языкознанию, и к *математической лингвистике* (*computational linguistics*).

Областью применения обработки естественного языка являются:

- системы, синтезирующие (*speech synthesis*) и распознающие речь (*speech recognition*);
- распознаватели ошибок написания, программы, разбивающие на слоги, распознаватели форм слова и другие применения, связанные с морфологией;
- интерфейсы пользователя (*human interface*) информационных, экспертных и других систем обработки данных;
- машинный перевод (*machine translation*);
- автоматическая интерпретация или генерация документов;
- обучение языку (*language learning*).

Начиная с некоторого уровня сложности, естественный язык необходим большинству систем обработки данных. Этой области предсказывается большое будущее.



Многообещающие прогнозы подтверждают хорошее согласование аппаратной части и разрабатываемого программного обеспечения, особенно на нижних уровнях обработки языка.

Разработка вычислительной техники, работающей на финском языке, является в нашей

стране национальной задачей первоочередной важности.

### Экспертные системы

Экспертная система (*expert system*) – это система обработки данных, основанная на знаниях (*knowledge*)



*Caduceus* – жезл, дающий жизнь, магический знак и эмблема некоторых экспертных систем.

и экспертных оценках в некоторой специальной области, и которая в состоянии с помощью специальной программы, принимающей решения, или машины вывода (*inference engine*) решать проблемы, для которых нужны, как считается, способности человека. Экспертные системы развились как *системы поддержки принятия решений* (*decision support system*). Этот более общий термин используют в тех случаях, когда желают подчеркнуть роль системы именно как помощника в принятии решения.

Экспертные системы обладают следующими свойствами, на основе которых их можно отличить от традиционных научно-технических и экономико-хозяйственных программ:

1. Степень разумности и объем знаний. Системы могут решить особенно сложные задачи в некоторых случаях лучше, чем эксперты-люди.
2. Развитый интерфейс пользователя. С такими системами можно, например, общаться на ограниченном естественном языке (в письменной форме) таким же образом, как и с экспертом-человеком.
3. Способность обосновывать действия. Кажется, что система обладает (мета)сведениями и пониманием своих знаний. У многих систем можно спросить, почему они задали некоторый вопрос или на основе чего они пришли к некоторому заключению. Таким образом пользователь может наблюдать над процессом рассуждений и разумностью решений.
4. Эвристики и работа с нечеткими данными. Системы часто в состоянии обрабатывать неполные, нечет-

- кие, недостаточные или даже ошибочные данные и знания.
5. Постепенное накопление знаний. Работу системы можно улучшить, добавив порцию знаний. Знания наиболее старых экспертных систем собраны в течение уже пары десятков лет.
  6. Системы часто работают с изображениями и задачами, представленными в символьном виде.



В экспертной системе не обязательно должны присутствовать все перечисленные здесь характеристики. Традиционные программы также могут содержать свойства, присущие экспертным системам.

С точки зрения деятельности экспертов задачи в различных областях часто классифицируются следующим образом:

1. Исследование и поиск ошибки (*diagnosis*). Например, поиск ошибки в технических устройствах и системах, медицинское диагностирование и т. д.
2. Интерпретация (*interpretation*). Поиск значения в неупорядоченной информации низкого уровня. Например, анализ сигналов измеряющего устройства или анализ результатов измерения эксперимента.
3. Наблюдение (*monitoring*). Например, наблюдение за состоянием процессов ядерной энергетики или за состоянием здоровья пациентов отделения интенсивной терапии.
4. Планирование действий (*planning*). Например, планирование хода решения проблем, выполнения экспериментов, мероприятий по реализации проектов и т. п.
5. Прогнозирование (*prediction*). Например, прогнозы и модели экономики, погоды, состояния окружающей среды.



*"Frieden Schaffen, ohne Waffen"*, символ движения за мир конференции IJCAI-83.

6. Управление производством (*control*). Например, планирование производства на предприятиях, производство (*CAM*) и контроль продукции (*CAT*) и др.
7. Проектирование (*design*). Проектирование и обсчет (*CAD*) машин, устройств, строений и т. п.
8. Обучение (*instruction, tutoring*). Компьютерное обучение и образование (*CAI*).

К возможным областям применения экспертных систем относятся почти все области человеческой деятельности:

- нетехнические профессии и науки, такие как медицина, юриспруденция и т. д.,
- инженерные науки и методы,
- обработка данных,
- обучение, консультирование и информационное обслуживание,
- социальные применения,
- машинная музыка, графика и другие искусства,
- и многие другие.

При этом, естественно, нужно учитывать возможности современной технологии и разумность разработки с экономической точки зрения.

### Символьные и алгебраические вычисления

Символьные и алгебраические вычисления (*symbolic and algebraic computing, SAC*) представляют собой одну из основных областей применения символьной обработки. Первыми ее достижениями были системы дифференцирования и интегрирования выражений. Для осуществления математических преобразований разработаны даже специальные вычислительные машины, например советские вычислительные машины серии "Мир" и имеющие более общее применение Лисп- и Пролог-машины, которые непосредственно реализуют язык символьной обработки высокого уровня.

Такие современные математические системы, как **MACSYMA**, **REDUCE**, **АНАЛИТИК** и другие, могут обрабатывать сложные математические выражения и

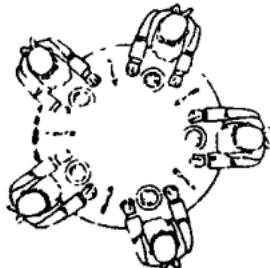
решать задачи, при решении которых вручную возникло бы много ошибок, или же они потребовали бы очень большой затраты труда вычислителей. В этих системах легко осуществляется сокращение выражений, разложение полинома на множители, решение системы линейных уравнений в символьном виде, работа с рядами, матрицами и тензорами или выполняются вычисления с рациональными или десятичными числами неограниченной точности. Алгебраические системы можно считать математическими экспертными системами.

### Доказательства и логическое программирование

*Доказательство теорем* (theorem proving) также является одной из традиционных областей применения искусственного интеллекта. Целью исследований является разработка общей, не зависящей от задачи процедуры и системы доказательства и решения проблем.

Современные исследования процедуры доказательства входят в область логического программирования (*logic programming*), которая в настоящее время

представляет собой одно из наиболее интенсивно развивающихся направлений. Для логического программирования разработаны специальные языки. Прежде всего нужно упомянуть о Прологе и его ответвлении, из которых сформировался сильный конкурент Лиспу. Логическое программирование используют, кроме всего прочего, в управлении реляционными базами данных, в математической логике, в решении абстрактных задач, аппаратных средствах, работающих с естественным языком, в проектировании архитектур ЭВМ, в символьных и алгебраических вычислениях, в исследовании строения биологических и химических молекул, в разработке экспертных систем и т.д.



они

Другими связанными с данной областью направлениями исследований являются *синтез* (program synthesis) и *верификация программ* (program proving, verification). Методы решения проблем в более свободной логической форме представляют собой рассуждения на уровне здравого смысла (common sense reasoning) и эвристические, или основанные на интуиции, процедуры (heuristics).

### Программирование игр

Программирование различных *игр* и *ведение игр* (game playing) вызывают постоянный интерес как у исследователей искусственного интеллекта, так и любителей игр. Известны программы, играющие в шахматы, и крупные международные турниры, устраиваемые между такими программами.



В связи с программированием игр разработаны специальные процедуры поиска, оценки и выбора, а также компьютерная графика, которая используется также и в производственных применениях. Интеллектуальные игры используются как для развлечения, так и для обучения.

### Моделирование



Методы искусственного интеллекта применяются в *моделировании* (modelling) различных систем и в изучении их деятельности путем *моделирования работы* (simulation). Моделирование работы подробно изучалось и в традиционных системах. Для этого даже были разработаны специальные языки, такие как Симула. С помощью символьной обработки и методов искусственного интеллекта особенно удобно моделировать многие дискретные системы. Содержащиеся в новых Лисп-системах, как, например, в Лисп-машинах, средства *объектно-ориентированного программирования* (object

programming, flavor) предлагают для моделирования работы различных систем гораздо лучшие механизмы, чем могут предложить традиционные языки моделирования.

**Когнитивное моделирование** (cognitive modelling) составляет самостоятельную область исследования, примыкающую к когнитивной психологии. Его задача — изучение и моделирование механизмов когнитивных процессов человека, т. е. каким образом человек познает, учится, рассуждает, изобретает и т. д. Эта наука, возникшая в конце 70-х годов на стыке исследований в областях искусственного интеллекта и психологии, называется когнитивной наукой (cognitive science).

### Обработка сигналов и распознавание образов

Под *распознаванием образов* (pattern recognition) понимается автоматическое наблюдение и идентификация (или классификация) объектов. Распознавание осуществляется на основе характеристик и закономер-

ностей, присущих принимаемым сигналам или с учетом строения образа. Задачи распознавание образов первоначально возникли при обработке сигналов (signal processing) и при разработке методов регулировки, затем оказалось, что они существенно связаны с обработкой изображений и речи. Многие физиологические, особенно относящиеся к физиологии нервов и мозга, психологические и лингвистические проблемы из-за своей естественной аналогии очень интересны с точки зрения методов и применения технического распознавания образов.

Задачи распознавания образов имеют много общего и с исследованиями по искусственному интеллекту. Их объединяют, например, когнитивные процедуры различного уровня и вида, такие как поиск, идентификация, классификация, обучение и другие. Однако решение проблемы с точки зрения распознавания образов отличается от решения той же проблемы с точки зрения символьной обработки. Оно больше заинтересовано в форме информации, а не в вопросах



ее содержания или значения. К распознаванию образов в символьной обработке относится поиск и сравнение различных символьных структур, *сопоставление с образцом* (*pattern matching*). Идентификация и ассоциирование символьных структур нужны в обработке языка и логическом программировании.

### Машинное зрение и обработка изображений

*Машинное зрение* (*machine vision*) и обработка изображений применимы во многих сферах и используют весьма различные методы. В обработке изображений различают *обработку изображений* (*image processing*) для непрерывных изображений и *обработку рисунков* (*picture processing*) для дискретных рисунков. Первая



по своему характеру является числовой и примыкает к обработке сигналов и распознаванию образов. Обработка рисунков традиционно входит в область *компьютерной графики* (*computer graphics*), применяемой, например, в *машинном проектировании* (*computer aided design*, *computer aided engineering*).

Символьная обработка и методы объектно-ориентированного программирования хорошо подходят для обработки рисунков, но их можно использовать в *интерпретации изображений* (*scene analysis*), заданных непрерывным образом, после того, как объекты на изображении и их границы идентифицированы с помощью методов обработки сигналов и распознавания образов.

### Робототехника и автоматизация производства



Обработка изображений занимает важное место и в *робототехнике* (*robotics*), которая направлена на снабжение машин различными измеряющими устройствами и датчиками, аналогами органов чувств, и другими механизмами ориентации в пространстве. Исследования по интеллектуальной робототехнике сосредоточены на создании

движущихся роботов и на методах самостоятельного принятия решений и планирования. Использование робототехники связано в основном с автоматизацией производства, в том числе с мекатроникой, объединяющей традиционные машины и новую вычислительную технику.

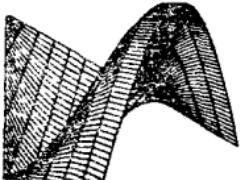
Технология знаний нужна и для глубокой интеграции планирования производства и его автоматизации, и создания так называемого *гибкого производства* (*flexible manufacturing*).

### **Машинное проектирование**

С ростом количества знаний и возможностей, заложенных в различных системах машинного проектирования и автоматизированного управления, возникла потребность в применении методов технологии знаний и

символьной обработки в этих системах. В машинном проектировании, *ориентированном на знания* (*knowledge based*), предполагается использование глубоких профессиональных знаний в нескольких областях, однако независимо от содержательной части для работы в каждой из этих областей

можно применять методы символьной обработки.



В области CAD технология знаний применяется, например, в проектировании схем СБИС. Наиболее развитые системы, так называемые кремниевые компьютеры (*silicon compiler*), способны автоматически порождать непосредственно из высокогоуровневого функционального описания схемы расположения слоев.

### **Языки и средства программирования искусственного интеллекта**

Обычно при программировании задач искусственного интеллекта традиционные языки программирования (Фортран, Паскаль, Бейсик, Кобол, Си и другие) не используются, а применяются языки, ориентированные на символьную обработку (Лисп, Пролог, Смолтолк, Лого и другие), основанные на них среды программиро-

вания (Интерлисп, Зеталисп и другие), языки представления и обработки знаний более высокого уровня (FRL, KL-TWO, OPS5 и другие), системы анализа (ATN, DCG и другие), оболочки экспертных систем (Етусіп, Expert и другие) и другое программное обеспечение, а также эффективная аппаратура, имеющая большой объем оперативной памяти, как, например, Лисп-машины.



Исследования в области искусственного интеллекта всегда касались и развития используемой техники и используемых программных средств и сред. Наиболее разносторонний инструментарий и интегрированные среды (Интерлисп, Зеталисп, KEE, ART и другие) имеются в современных Лисп-машинах.

### Повышение производительности программирования

Кроме новых возможностей применения используемые в исследованиях по искусенному интеллекту и технологии знаний языки программирования, механизмы и методы могут принести пользу и в традиционных применениях, позволяя достичь в большинстве случаев более высокой производительности труда программиста. Например, с помощью среды программирования Лисп-машин в лучших случаях можно повысить производительность программирования в несколько десятков раз по сравнению с традиционными системами.

### Автоматическое программирование и обучение

В более далекой перспективе развитие инструментария программиста приведет к *автоматическому программированию* (automatic programming, program synthesis) и *самообучающимся* (learning) системам. Целью исследователей является заставить вычислительную машину написать самой себе программу на основе даваемых ей намеков или используя описание высокого уровня.

## Литература

### Обработка естественного языка

1. Brady M., Berwick R. (Eds.) *Computational Models of Discourse*. MIT Press, Cambridge, Massachusetts, 1984.
2. Charniak E., Wilks Y. (Eds.) *Computational Semantics*. North Holland, Amsterdam, 1976.
3. Schank R., Reisbeck C. *Inside Computer Understanding: Five programs plus Miniatures*. Erlbaum, Hillsdale, New Jersey, 1981.
4. Tennant H. *Natural Language Processing*. North Holland, Amsterdam, 1978.
5. Walker D. *Understanding Spoken Language*. North Holland, Amsterdam, 1978.
6. Winograd T. *Language as a Cognitive Process*, Vol 1, *Syntax*. Addison-Wesley, Reading, Massachusetts, 1983. (Vol. 2, *Semantics*, в печати.)



### Экспертные системы

1. Clancey W., Shortliffe E. *Readings in Medical Artificial Intelligence*. Addison-Wesley, New York, 1984.
2. Hayes-Roth F., Waterman D., Lenat D. (Eds.) *Building Expert Systems*. Addison-Wesley, Reading, Massachusetts, 1983. [Имеется перевод: Хейес-Рот Ф., Уотерман Д., Ленат Д. Построение экспертных систем. -М.: Мир, 1987.]
3. Hyvönen E. *Asiantuntijajärjestelmät*. TKK, Digiitaaltekniikan laboratorio, raportti A21, 1985.
4. Hyvönen E., Seppänen J., Syrjänen M. (Eds.) *SteP-84 Tutorial and Industrial Papers*. Tietojenkäsittelytieteen seura, julkaisu No.4, Espoo, 1985.
5. Michie D. *Expert Systems in the Microelectronic Age*. Edinburgh University Press, 1979.
6. Shortliffe E. *Computer Based Medical Consultations: MYCIN*. Elsevier Publ. Comp., Amsterdam, 1976.

7. Waterman D. *A Guide to Expert Systems*. Addison-Wesley, Reading, Massachusetts, 1986. [Имеется перевод: Уотерман Д. Руководство по экспертным системам. -М.: Мир, 1989.]
8. Weiss S., Kulikowski C. *A Practical Guide to Designing Expert Systems*. Chapman and Hall, London, 1984.

### *Доказательство теорем*

1. Chang C., Lee R. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, New York, 1973. [Имеется перевод: Чень Ч., Ли Р. Математическая логика и автоматическое доказательство теорем. -М.: Наука, 1983.]
2. Loveland D. *Automated Theorem Proving*. North Holland, Amsterdam, 1978.
3. Bibel W. *Automated Theorem Proving*. Vieweg, BRD, 1982.

### *Логическое программирование*

1. Bramer M., Bramer D. *The Fifth Generation – An Annotated Bibliography*. Addison-Wesley, Reading, Massachusetts, 1984.
2. Campbell J.A. *Implementations of Prolog*. Ellis Horwood, Chichester, Great Britain, 1984.
3. Clocksin W., Mellish C. *Programming in Prolog*. Springer-Verlag, New York, 1981. [Имеется перевод: Клоксин У., Меллиш К. Программирование на языке Пролог. -М.: Мир, 1987.]
4. Kowalski R. *Logic for Problem Solving*. North Holland, Amsterdam, 1979.
5. Clark K., Tärnlund S.-Å. (Eds.) *Logic Programming*. Academic Press, London, 1982.

### *Машинное зрение и обработка изображений*

1. Winston P. *The Psychology of Computer Vision*. McGraw-Hill, New York, 1975. [Имеется перевод:

- Уинстон П. Психология машинного зрения. -М.: Мир, 1978.]
2. Hanson A., Riseman H., (Eds.) *Computer Vision Systems*. Academic Press, London, 1978.
  3. Hall E. *Computer Image Processing and Recognition*. Academic Press, London, 1979.
  4. Brady M. (Eds.) *Special Volume on Computer Vision. Journal of Artificial Intelligence*, Vol. 17, № 1-3, 1981.
  5. Marr D. *Vision*. Freeman, San Francisco, 1982.

### Экономические приложения

- 
1. Hyvönen E. *Tekoälytutkimuksen kaupalliset sovellukset ja markkinat*. TKK, Digitaaltekniikan laboratorio, raportti A 17.
  2. Johnston T. *The Commercial Application of Expert Systems Technology*. Ovum Ltd., London, 1984.
    3. Johnston T. *Natural Language Computing: The Commercial Applications*. Ovum Ltd., London, 1985.
    4. Mäkelin M., Huomo T., Riivari J. *Asian-tuntijajärjestelmät. Perusteet, sovellustulueet, kypsyys, merkitys yrityksille*. TT-Innovation Oy, Helsinki, 1985.
  5. *The International Directory of Artificial Intelligence Companies*. Artificial Intelligence Software, Rovigo, Italy, 1984.
  6. Winston P., Prendergast K. (Eds.) *The AI Business*. MIT Press, Cambridge, Massachusetts, 1984.
  7. *1984 Inventory of Expert Systems*. SEAI Institute, Madison, Georgia, 1984.

### Искусственный интеллект и философия

1. Boden M. *Artificial Intelligence and Natural Man*. Harvester Press, San Francisco, 1977.
2. Dreyfus H. *What Computers Can't Do*. Harper et. Row, New York, 1979. [Имеется перевод: Дрейфус Х. Чего не могут вычислительные машины. -М.: Прогресс, 1979.]

3. McCorduck P. *Machines Who Think*. Freeman, San Francisco, 1979.
4. Raphael B. *Thinking Computer – Mind Inside Matter*. Freeman, San Francisco, 1976. [Имеется перевод: Рафаэл Б. Думающий компьютер. –М.: Мир, 1979.]
5. Weizenbaum J. *Computer Power and Human Reason*. Freeman, San Francisco, 1976.

### *Искусственный интеллект и искусство*

1. Cohen H., Cohen B., Nii P. *Art and Computers. The First Artificial Intelligence Coloring Book*. William Kaufmann Inc., 1983.
2. Hofstadter D. *Gödel, Escher, Bach, An Eternal Golden Braid*. Basic Books, New York, 1979.

### **Упражнения**

1. Перечислите повседневные проблемы, при решении которых в настоящее время нельзя прибегнуть к помощи вычислительной техники из-за ее "тупости".
2. Попробуйте определить понятие "разум". Разумен ли калькулятор? А играющая в шахматы или переводящая с одного языка на другой вычислительная машина? Меняется ли со временем определение разумности?



*Революционным является то,  
что обновляет самое себя.*

*Л. Витгенштейн*

## 1.3 ЛИСП – ЯЗЫК ПРОГРАММИРОВАНИЯ ИСКУССТВЕННОГО ИНТЕЛЛЕКТА

- Символьная обработка
- Лисп опередил свое время
- Однаковая форма данных и программы
- Хранение данных, не зависящее от места
- Автоматическое и динамическое управление памятью
- Функциональный образ мышления
- Возможность различных методов программирования
- Пошаговое программирование
- Интерпретирующий или компилирующий режимы выполнения
- Лисп – бестиповый язык программирования
- Единый системный и прикладной язык программирования
- Интегрированная среда программирования
- Широко распространенные заблуждения и предрассудки
- Простой и эффективный язык
- Учебная литература по Лиспу
- Упражнения

Вычислительные машины первоначально разрабатывались для осуществления численных вычислений и обработки больших объемов данных. Численным

**3** вычислениям свойственно выполнение большого числа алгоритмов или сложных, но заранее точно определенных вычислений над сравнительно простыми однообразными элементами данных – числами, векторами, массивами и т. п. Наиболее эффективные суперкомпьютеры способны выполнять в настоящее время уже миллиарды операций в секунду.

Численные вычисления хорошо подходят для решения научно-технических задач, их понятия количественно измеримы и точно представимы в числовой форме.

В свою очередь в экономических задачах обрабатывается большое количество данных, состоящих часто из элементов данных различных типов, которые, однако, регулярны по форме и относятся к количественным типам данных, таких, например, как записи, файлы и базы данных. Осуществляемые действия сравнительно просты, но программы могут быть довольно велики.

### Символьная обработка

Вычислительные машины, ориентированные на решение описанных выше задач, и языки программирования не подходят столь же хорошо для символьной обработки, которая имеет дело со сложными структурами данных и базами знаний, содержащими правила принятия решений и другие многообразные объекты. Символьная обработка позволяет эффективно работать с такими структурами, как предложения естественного языка, значения слов и предложений, нечеткие понятия и т.д., и на их основе принимать решения, проводить рассуждения и осуществлять другие, свойственные человеку способы обращения с данными. В качестве типичного примера мы уже ранее рассматривали экспертные системы, содержащие профессиональные знания по некоторой специальности, программы, работающие с естественным языком и т.д.



В этих применениях предполагается представление в подходящей форме символьных и сложно структурированных данных (*knowledge representation*) и работа с ними часто ведется в заранее непредсказуемых ситуациях. Характерно, что кроме сложной структуры таким данным свойственно разнообразие форм их выражения. Большая часть объектов данных конкретной проблемной области может иметь отличное от других индивидуальное строение. Однако при их обработке поведение программы определяется на основе задаваемых на более общем уровне принципов,

законов и правил, а также на основе типов ситуаций и образцов, распознаваемых в этих ситуациях.

Например, в играющей в шахматы программе невозможно заранее учесть все позиции. Анализ игры осуществляется на основе классификации позиций, распознавания стандартных позиций, определения характеристик позиций, построения оценок текущей позиции и использования ограниченного набора стратегий, правил принятия решения и т.д. Так программа может оценить и такие позиции, которые программист специально не предусматривал. При удачном стечении обстоятельств программа может победить и своего создателя.

С помощью структур, имеющих множество форм представления, стало возможным решать задачи, которые ранее считались практически или даже принципиально неразрешимыми. Наибольшие ограничения в создании разумных программ накладывает недостаточное знание нами механизмов принятия решений, строения знаний об естественном языке и о других областях, а также уровень нашего мастерства в программировании.

### **Лисп опередил свое время**

Лисп является наиболее важным языком программирования, используемым в исследованиях по искусственному интеллекту и в математической лингвистике. Название языка "Лисп" происходит из "list processing" (обработка списков). Буквально английское слово "lisp" означает "лепетать", "шепелявить" и "сюсюкать". В качестве имени языка программирования искусственного интеллекта это метко, поскольку именно с помощью Лиспа вычислительные машины научились в некоторой мере лепетать на человеческом языке.

И все же Лисп ни в каком смысле не является младенцем. Скорее он старик среди языков программирования. Лисп разработан уже в 50-х годах и является вслед за Фортраном старейшим еще используемым языком программирования. Но в отличие от Фортрана будущее Лиспа еще впереди.

Лисп представляет собой язык так называемого функционального программирования. Он основан на алгебре списочных структур, лямбда-исчислении и теории рекурсивных функций. Благодаря неразвитости традиционной вычислительной техники, отличающемуся от других языков программирования характеру и из-за наличия элементарных средств обработки списков Лисп долгое время являлся основным инструментом исследователей искусственного интеллекта и средством теоретического подхода к анализу программирования. Однако в 80-х годах Лисп, наконец, вышел из лабораторий и нашел применение в прикладных проблемах.



Обычно языки программирования не изобретают, а проектируют. Однако по отношению к Лиспу можно говорить об изобретении. Первоначальная версия языка, в частности, содержала множество понятий и принципов, которые сначала казались очень странными, но многие из которых позже оказались существенным нововведением. Кроме этого, возможность добавления в Лисп в течение десятилетий многих новых черт подтвердила свойство расширяемости этого языка. Вокруг Лиспа возникла широкая культура, охватывающая многочисленные школы и разнообразные диалекты языка, различные системы и методы программирования, программные среды и Лисп-машины.

Далее мы коротко рассмотрим многочисленные свойства Лиспа и его отличия от других языков программирования.

### **Однаковая форма данных и программы**

В Лиспе формы представления программы и обрабатываемых ею данных одинаковы. И то и другое представляется списочной структурой, имеющей одинаковую форму. Таким образом программы могут обрабатывать и преобразовывать другие программы и даже самих себя. В процессе трансляции можно введенное и сформированное в результате вычислений выражение данных проинтерпретировать в качестве программы и непосредственно выполнить. Предоставленная этим

возможность так называемого *программирования, управляемого данными* (*data driven programming*), и непосредственное влияние программы на программу (или на программы) тесно связаны между собой.

Это свойство обладает не только теоретическим, но и большим практическим значением. Единообразность представления данных и программы можно сопоставить



с принципом фон Неймана, по которому программа и данные хранятся в единой памяти. Благодаря этому появилась великолепная возможность создания универсальной вычислительной машины обработки данных. Соответственно единообразное представление программы и данных и реализация программы путем интерпретации открывают совершенно новые возможности программирования.

Универсальный единообразный и простой лисповский синтаксис списка не зависит от применения, и с его помощью легко определять новые формы записи, представления и абстракции. Даже сама структура языка является, таким образом, расширяемой и может быть заново определена. В то же время достаточно просто написание интерпретаторов, компиляторов, преобразователей, редакторов и других средств. К Лиспу стоит подойти как к языку программирования, с помощью которого реализуются специализированные языки, ориентированные на приложение, и создается окружение более высокого уровня. Присущая Лиспу гибкая расширяемость не встречается в традиционных замкнутых языках программирования.

### **Хранение данных, не зависящее от места**

Списки, представляющие программы и данные, состоят из списочных ячеек, расположение и порядок которых в памяти несущественны. Структура списка определяется логически на основе имен символов и указателей. Добавление новых элементов в список или удаление из списка может производиться без переноса списка в другие ячейки памяти. Резервирование и освобождение могут в зависимости от потребности осуществляться



динамически, ячейка за ячейкой. По-другому обстоит дело, например, при обращении с фортрановскими строками или массивами.

### Автоматическое и динамическое управление памятью

Пользователь не должен заботится об учете памяти. Система резервирует и освобождает память автоматически в соответствии с потребностью. Когда память кончается, запускается специальный *мусорщик* (*garbage collector*). Он собирает неиспользуемые символы и списки, включает их в работу путем вторичного использования. Среда Лиспа постоянно содержитя в порядке. Во многих системах мусор не образуется, поскольку он сразу же учитывается. Управление памятью просто и не зависит от физического расположения, поскольку свободная память логически состоит из цепочки списочных ячеек.

В первую очередь данные обрабатываются в оперативной и виртуальной памяти, которая может быть довольно большой (с адресацией в 28–32 разряда). Файлы используются в основном для хранения программ и данных в промежутке между сессиями.

### Функциональный образ мышления

Используемое в Лиспе так называемое *функциональное программирование* (*functional programming*) основывается на той простой идеи, что в результате каждого действия возникает значение. Значения становятся аргументами следующих действий, и конечный результат всей задачи выдается пользователю.

Программы строятся из логически расчлененных определений функций. Определения состоят из организующих вычисления управляющих структур и из вложенных, часто вызывающих самих себя (рекурсивных) вызовов функций. Основными средствами функцио-



нального программирования как раз и являются композиция и рекурсия.

### **Возможность различных методов программирования**

Кроме функционального программирования в Лиспе можно использовать программирование, основанное на обычном последовательном исполнении операторов с присваиваниями, передачами управления и специальными операторами цикла. С помощью макропрограммирования можно запрограммировать новые структуры языка или реализовать совершенно новые языки. Кроме того, в Лиспе можно применять множество методов программирования, известных из традиционных языков. В зависимости от системы в Лиспе можно использовать методы программирования более высокого уровня например такие, как *объектно-ориентированное программирование* (*object oriented programming*), *ситуационное программирование* (*event-based programming*), *продукционное программирование* (*rule-based programming*) и *логическое программирование* (*logic programming*). Различные методы можно применять совместно в единой интегрированной среде.



Более высокий уровень программирования в Лиспе достигается за счет использования макропрограммирования. Макропрограммы позволяют создавать новые языковые конструкции, которые могут быть использованы для решения различных задач. Это делает Лисп очень гибким языком, способным адаптироваться к различным потребностям.

Методы программирования в Лиспе включают в себя:

- Функциональное программирование**: основано на применении функций как основных единиц вычислений. Операторы присваивания и циклов не используются.
- Следовательное программирование**: основано на последовательном исполнении операторов.
- Объектно-ориентированное программирование**: основано на представлении данных и операций в виде объектов.
- Ситуационное программирование**: основано на обработке событий.
- Продукционное программирование**: основано на использовании правил для вывода информации.
- Логическое программирование**: основано на решении логических задач.

### **Пошаговое программирование**

При пошаговом программировании программа разрабатывается постепенно, шаг за шагом. Каждый шаг включает в себя написание и исправление кода, а также его тестирование. Такой подход позволяет выявлять ошибки на ранней стадии разработки и корректировать программу до тех пор, пока она не будет работать корректно. Пошаговое программирование является эффективным методом для создания сложных приложений, так как оно позволяет контролировать каждый этап разработки и убедиться в том, что программа работает правильно.

Вспомогательные средства, находящиеся в распоряжении пользователя, такие как, например редактор,



трассировщик, инспектор, транслятор, структурная печать и другие образуют общую интегрированную среду, язык которой нельзя четко отличить от системных средств. Отдельные средства по своему принципу являются прозрачными,

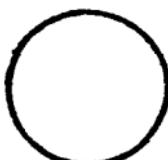
чтобы их могли использовать другие средства. Например, интерпретатор может вызвать редактор, редактор – интерпретатор и так далее даже рекурсивно. Работа может производится часто на различных уровнях или в различных рабочих окнах. Такой способ работы особенно хорошо подходит для *исследовательского программирования* (*exploratory programming*) и быстрого построения прототипов (*rapid prototyping*).

### **Интерпретирующий или компилирующий режимы выполнения**

Лисп является в первую очередь интерпретируемым языком. Пользователь может просто попробовать новые идеи и получить непосредственный отклик об их плодотворности. Программы не нужно транслировать, и их можно исправлять в процессе исполнения. Отлаженную функцию можно передать на трансляцию, тогда она выполняется быстрее. В одной и той же программе могут быть транслированные и интерпретируемые функции. Оттранслированную один раз функцию не нужно транслировать вновь из-за ошибок в других функциях. *Транслирование по частям* (*incremental compiling*) экономит усилия программиста и время вычислительной машины.

### **Лисп – бестиповый язык программирования**

В Лиспе имена символов, переменных, списков, функций и других объектов не закреплены предварительно за какими-нибудь типами данных. Типы в общем не связаны с именами объектов данных, а сопровождают сами объекты. Таким образом, переменные могут в различные моменты времени представлять



различные объекты. В этом смысле Лисп является *бестиповым* (*typeless*) языком.

Например, в Фортране имена переменных, начинающиеся с букв I, J и K, закреплены за целыми числами уже с момента определения языка в 50-х годах. В Паскале тип переменной закрепляется на этапе написания программы. В некоторых языках тип переменной определяется на этапе трансляции. В Лиспе тип определяется по ходу выполнения программы.

Динамическая, осуществляемая лишь в процессе исполнения, проверка типа и позднее связывание (*late binding*) допускают разностороннее использование символов и гибкую модификацию программ. Функции можно определять практически независимо от типов данных (*genericity, orthogonality*), к которым они применяются.

Однако указанная бестиповость не означает, что в Лиспе вовсе нет данных различных типов. Мы далее увидим, что набор типов данных наиболее развитых Лисп-систем необычайно разнообразен.

Одним из общих принципов развития Лисп-систем было свободное включение в язык новых возможностей и структур, если считалось, что они найдут более широкое применение. Это было возможно в связи с естественной расширяемостью языка.



Платой за динамические типы являются действия по проверке типа на этапе исполнения. В более новых Лисп-системах (Коммон Лисп) возможно факультативное определение типов. В этом случае транслятор может использовать эту информацию для оптимизации кода. В Лисп-машинах проверка типа эффективно осуществляется на уровне аппаратуры (*tagged architecture*).

### Единый системный и прикладной язык программирования

Лисп является одновременно как языком прикладного, так и системного программирования. Он напоминает машинный язык тем, что как данные, так и программа

представлены в одинаковой форме. Язык превосходно подходит для написания интерпретаторов и трансляторов как для него самого, так и для других языков. Например, ядро интерпретатора Лиспа, написанное на самом Лиспе, составляет пару страниц красивого кода. Примерно в том же объеме уместится и ядро интерпретатора Пролога. Как известно, наиболее короткий интерпретатор Пролога, написанный на Лиспе занимает несколько десятков строк.

Традиционно Лисп-системы в основной своей части написаны на Лиспе. При программировании, например транслятора, есть возможность для оптимизации



объектного кода использовать сложные преобразования и методы искусственного интеллекта. Лисп можно в хорошем смысле считать языком машинного и системного программирования высокого уровня. И это особенно характерно для Лисп-машин,

которые вплоть до уровня аппаратуры спроектированы для Лиспа и системное программное обеспечение которых написано на Лиспе.

### **Интегрированная среда программирования**

Лисп-системы и среды программирования развиваются уже начиная с конца 50-х годов. С самого начала разработки в них был заложен принцип возможности использования отдельных средств непосредственно из интерпретатора.

В реализациях для микроЭВМ количество системных и встроенных функций обычно ограничивается порядком десяти, сотни. В больших системах разделения времени, таких как Маклисп (MacLisp) и Интерлисп (Interlisp), функций многие сотни. Коммон Лисп и автоматизированные рабочие места (рабочие станции) для исследователей искусственного интеллекта и Лисп-машины предлагают особенно широкий выбор различных вспомогательных средств и функций интегрированной среды, которая, кроме всего прочего, открыта для модификаций пользователя.

Системное программное обеспечение Лисп-машин содержит более 10.000 функций и десятки мегабайтов

кода. Во многих системах исходные тексты лисповского кода предоставляются пользователю. Системные функции можно переопределить или модифицировать, и пользователь может свободно расширить и подогнать систему для себя.



Перечень нововведений и парадигм Лисп-культуры можно было бы легко продолжить, рассматривая свойства и принципы, связанные с методами и средой программирования. Мы вернемся к ним в связи с типами данных и функций, методами программирования, а также Лисп-системами и Лисп-машинами.

### Широко распространенные заблуждения и предрассудки

Про Лисп говорят, что это неэффективный язык, особенно в части арифметики. Это было верно на начальном этапе. На численную сторону не обращалось внимание, поскольку язык был предназначен для символьной обработки. Теперь это уже не так. Лисп-системы могут в численных вычислениях быть более эффективными, чем Фортран на той же машине. Это объясняется в том числе и тем, что в трансляторах с Лиспом можно применять более сложные преобразования для оптимизации кода.

Естественным является то, что большие системы, предлагающие многосторонние и разумные услуги, требуют большой вычислительной мощности и громоздки для системы разделения времени. Однако это связано не с Лиспом или его плохой реализацией, а с тем вниманием и с той готовностью помочь, которые система предлагает пользователю.

Например, Интерлисп сохраняет полный перечень всех действий пользователя и полное описание более ранних состояний системы и способен автоматически исправлять многие неточности, как, например, ошибки в написании. Если эти свойства правильно использовать, то они повышают производительность профессио-



нального программиста в большей степени, чем тратят ресурсы машины.

С точки зрения программирования критике подвергается и внешний облик языка: изобилие скобок и кажущийся беспорядок. Лисп ("Lots of Idiotic Silly Parentheses") представляется как труднопонимаемый и трудноизучаемый язык.

Такой подход проистекает из используемых в Лиспе функционального образа мышления и техники программирования, которые чужды программистам, привыкшим к операторному программированию традиционных



языков. Естественно используемые в программировании на Лиспе структуры данных и управляющие структуры часто сложны, поскольку проблемы искусственного интеллекта из-за своей сложности предполагают сложные структуры и программы. Иерархические списочные структуры и Лисп как раз и задумывались для работы со сложными проблемами. Искусственное упрощение структур означало бы пренебрежение действительной сложностью проблем.

Идеей Лиспа является попытка упростить решение проблемы, структурируя используемые данные и упрощая программы. Такой подход оказался полезным, например, в объектно-ориентированном программировании и в экспертных системах. Эти области содержат больше количества знаний со сложной структурой, которые интерпретируются часто довольно простыми процедурами поиска и принятия решения.

### Простой и эффективный язык

Структура языка Лисп проста и последовательна. Функции Лиспа можно напечатать в ясно структурированном и хорошо читаемом виде. Во многих системах существует возможность использования формы записи с малым количеством скобок и близкой по виду к более традиционным языкам.

Хорошим доказательством простоты использования Лиспа является его широкое применение в исследовательских работах по программированию сложных методов обработки знаний. Большая часть значимых



программ искусственного интеллекта запрограммирована на Лиспе или на основанном на нем языке более высокого уровня. По результатам некоторых исследований можно, скажем, в среде программирования Лисп-машины

достичь повышения производительности программирования в несколько десятков раз по сравнению, например, с программированием на Коболе.

### Учебная литература по Лиспу

1. Danicic I. *Lisp Programming*. Blackwell Scientific Publications, Boston, 1983.
2. Friedman D. *The Little Lisper*. Science Research Associates, Chigaco, 1974.
3. Gloess P. *Understanding Lisp – A Concise Introduction to the Language of Artificial Intelligence. An Alfred Handy Guide*, Alfred Publishing, Sherman Oaks, California, 1982.
4. Hamann C.-M. *Einfuerung in das Programmieren in Lisp*. de Gruyter, Berlin, 1982.
5. Hasemer T. *A Beginner's Guide to Lisp*. Addison-Wesley, Reading, Massachusetts, 1984.
6. Henderson P. *Functional Programming – Application and Implementation*. Prentice-Hall, London, 1980.
7. Лавров С., Силагадзе Г. *Автоматическая обработка данных. Язык лисп и его реализация*. -М.: Наука, 1978.
8. Maurer W.D. *A Programmer's Introduction to Lisp*. McDonald Monographs, American Elsevier, New York, 1972. [Имеется перевод: Майреп У. Введение в программирование на языке ЛИСП. -М.: Мир, 1976.]
9. McCarthy J. *Lisp 1.5 Programmers Manual*. MIT Press, Cambridge, Massachusetts, 1963.
10. Norman A., Cattel G. *Lisp on the BBC Microcomputer*. Acornsoft, Cambridge, England, 1983.
11. Queinnec C. *Language d'un autre type: Lisp*. Eyrolles. Paris, 1984.



12. Queinnec C. *Lisp – Mode d'emploi*. Eyroll Paris, 1984.
13. Ribbens D. *Programmation non numerique Lisp 1.5*. Dunod, Paris, 1969.
14. Schraeger J., Bagley S. *Learning Lisp*. Gnosis, Engelwood Cliffs, New Jersey, 1984.
15. Seppänen J. *Lisp kielenopas*. OtaData, Espoo, 1972.
16. Siklossy L. *Let's Talk Lisp*. Prentice-Hall, London, 1976.
17. Steele G. *Common Lisp – the Language*. Digital press, Hannover, Massachusetts, 1984.
18. Stoyan H., Görz G. *Lisp – Eine Einführung in die Programmierung*. Springer-Verlag, Berlin, 1984.
19. Touretzky D. *Lisp – A Gentle Introduction to Symbolic Computation*. Harper et Row, New York, 1984.
20. Tracton K. *Programmer Guide to Lisp*. Tab Books Inc., Palo Alto, California, 1983.
21. Weismann C. *Lisp 1.5 Primer*. Dickenson Press, Bellmont, California, 1967.
22. Wilensky R. *LispCraft*. Norton et Co., New York, 1984.
23. Winston P., Horn B. *Lisp*. Addison-Wesley, Reading, Massachusetts, 1981, 1984.

### Упражнения

1. Какие мифы связаны с языком программирования Лисп? Как на самом деле обстоят дела?



## **2 ОСНОВЫ ЯЗЫКА ЛИСП**

- 2.1 СИМВОЛЫ И СПИСКИ**
- 2.2 ПОНЯТИЕ ФУНКЦИИ**
- 2.3 БАЗОВЫЕ ФУНКЦИИ**
- 2.4 ИМЯ И ЗНАЧЕНИЕ СИМВОЛА**
- 2.5 ОПРЕДЕЛЕНИЕ ФУНКЦИЙ**
- 2.6 ПЕРЕДАЧА ПАРАМЕТРОВ И ОБЛАСТЬ ИХ ДЕЙСТВИЯ**
- 2.7 ВЫЧИСЛЕНИЕ В ЛИСПЕ**
- 2.8 ВНУТРЕННЕЕ ПРЕДСТАВЛЕНИЕ СПИСКОВ**
- 2.9 СВОЙСТВА СИМВОЛА**
- 2.10 ВВОД И ВЫВОД**

*В этой главе мы познакомимся с основами использования списков, средствами работы с ними, с их выражением в языке Лисп и с принятыми в Лиспе способами осуществления основных операций. Прежде всего рассмотрим, каким образом представляются данные и программы в виде списков, или в виде списочной нотации (2.1). Затем ознакомимся с лисповским пониманием функции, на котором основаны программирование в Лиспе и так называемое функциональное программирование (2.2), представим базовые функции Лиспа, составляющие основу работы со списками (2.3). После этого научимся пользоваться именованными переменными и получать значения выражений (2.4), использовать так называемые лямбда-выражения для определения своих лисповских функций (программ) (2.5) и передавать значения параметров этих функций (2.6). В главе 2.7 ознакомимся с наиболее важными управляющими конструкциями, такими как условные и итеративные предложения. В конце главы рассмотрим представление списков в памяти (2.8), списки свойств символов (2.9), а также программирование ввода и вывода.*

*Слова – это крючки, при помощи которых мысли цепляются одна за другую.*

Г. Бичер

## 2.1 СИМВОЛЫ И СПИСКИ

- Символы используются для представления других объектов
- Символы в языке Кёммон Лисп
- Числа являются константами
- Логические значения Т и NIL
- Константы и переменные
- Атомы = Символы + Числа
- Построение списков из атомов и подсписков
- Пустой список = NIL
- Список как средство представления знаний
- Значение способа записи
- Различная интерпретация списка
- Упражнения

Символы используются для представления других объектов

В программировании на языке Лисп используются *символы* и построенные из них *символьные структуры*. В соответствии со словарным определением, символом (греч. "symbolon", знак) является опознавательный знак, условное обозначение, художественный образ, обозначающие какую-нибудь мысль, идею и т. п. В Лиспе понятие символа используется в более узком и точном смысле: под ним подразумевается запись или обозначение.

М А  
Х Г

Знаки, которые ставили вместо подписи.

Символ – это имя, состоящее из букв, цифр и специальных знаков, которое обозначает какой-нибудь предмет, объект, вещь, действие из реального мира. В

Лисп символы обозначают числа, другие символы или более сложные структуры, программы (функции) и другие лисповские объекты. Например, символ "+" может обозначать определение действия сложения, "углерод-14" – изотоп углерода и т.п.

Примеры символов:

x  
Символ  
defun  
STeP-1984

### Символы в языке Коммон Лисп

Символы языка Коммон Лисп могут состоять из букв, цифр и некоторых других знаков, а именно

+ - \* / @ \$ % ^ & \_ \ < > ~ □ .

В принципе, вопросительный и восклицательный знаки, так же как тильда (~), и квадратные скобки тоже могут входить в состав символов, но эти знаки лучше приберечь для специального использования.

Символы могут состоять как из прописных, так и из строчных букв, хотя в большинстве Лисп-систем, как и в описываемом в данной книге диалекте Коммон Лисп, прописные и строчные буквы отождествляются и представляются прописными буквами. Поэтому, например:

**слово ◊ Слово ◊ СЛОВО**

При желании в состав символов можно включать пробелы и другие зарезервированные специальные знаки. Для этого либо символ с двух

сторон ограничивается вертикальной чертой (:, bag), либо перед каждым таким специальным знаком, входящим в символ, ставится обратная косая черта (\, back-slash). С точки зрения финского языка это соглашение языка Коммон Лисп неудобно, так как на скандинавской клавиатуре



**Магические знаки.**

на месте вертикальной черты и обратной косой черты стоят соответственно буквы ё и Ö. Поэтому для использования буквы Ö в составе символа перед ней нужно поставить еще одну Ö или с обеих сторон символа поставить букву ё. Буквы Ä и Å могут свободно использоваться в составе символов, однако, учитывая то, что



в коде ASCII им соответствуют квадратные скобки (ä и Å) и фигурные скобки (å и Å), эти строчные и прописные буквы не будут отождествляться (так же как и ё и Ö):

`hölmöä ; не является символом Коммон Лиспа  
öhölmöäö ; это символ HöLMöä, выделен весь символ  
hÖölmÖöÄ ; символ HöLMöÄ, ё выделены поодиночке`

В этой книге мы предполагаем, что Лисп-система модифицирована таким образом, что скандинавские буквы можно использовать точно так же, как все другие. Например, при вводе в стандартную Коммон Лисп систему можно просто преобразовывать строчные и прописные Ä и Å в букву A, а Ö в O, и тогда с этими буквами не будет проблем.

### Числа являются константами

Наряду с символами в Лиспе используются и числа, которые, как и символы, записываются при помощи ограниченной пробелами последовательности знаков.



Числа все же не являются символами, так как число не может представлять иные лисповские объекты, кроме самого себя, или своего числового значения.

Как и в других языках программирования, в Лиспе для различных целей используется много различных типов чисел. Примеры чисел:

`746 ; целое число`

`-3.14 ; десятичное число`

`3.055E8 ; число, представленное мантиссой и порядком`

Числа отличаются от символов способом записи. Впоследствии типы лисповских чисел будут рассмотрены подробнее.

### Логические значения Т и NIL

Символы Т и NIL имеют в Лиспе специальное назначение: Т обозначает логическое значение истина (true), а NIL – логическое значение ложь (false). (В некоторых системах для обозначения логического значения ложь используется специальный символ F (false).) Символом NIL обозначается также и пустой список, этот момент мы вскоре рассмотрим подробнее. Символы Т и NIL имеют всегда одно и то же фиксированное встроенное значение. Их нельзя использовать в качестве имен других лисповских объектов.

### Константы и переменные

Числа и логические значения Т и NIL являются *константами* (constant), остальные символы – *переменными* (variable), которые используются для обозначения других лисповских объектов.

Кроме этого, система Коммон Лисп содержит *глобальные специальные переменные* (global special/dynamic variable), имеющие изменяемые

*π* встроенные значения. Например, такими переменными являются символ PI, представляющий значение числа π, и символ \*STANDARD-INPUT\*, определяющий, откуда вводятся значения.

*p*. В языке Лисп предусмотрена специальная директива (DEFCONSTANT), используемая для превращения любого символа в константу. Символ, определенный как константа, может обозначать лишь то значение, которое ему предписано таким определением.

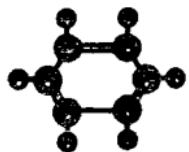
### Атомы = Символы + Числа

Символы и числа представляют собой те простейшие объекты Лиспа, из которых строятся остальные структуры. Поэтому их называют атомарными объектами или просто *атомами* (atom). Позднее мы увидим, что в

Лиспе, как и в физике, можно при необходимости расщепить атом на элементарные составляющие.

### Построение списков из атомов и подсписков

Атомы и списки (*list*) – это основные типы данных языка Лисп. В естественном языке под списком понимают перечень. Список в Лиспе – это тоже перечень или даже точнее – упорядоченная последовательность, **элементами** (*element*) которой являются атомы либо списки (*подсписки*). Списки заключаются в круглые скобки, элементы списка разделяются пробелами. Список всегда начинается с открывающей скобки и заканчивается закрывающей скобкой. Например, следующий список состоит из трех символов и одного подсписка, который в свою очередь состоит из двух атомов:



(**a b (c d) e**)

Таким образом, список – это многоуровневая или иерархическая структура данных, в которой открывающие и закрывающие скобки находятся в строгом соответствии. Например, приведенные ниже выражения являются правильно составленными списками:

(+ 2 3) ; список из трех элементов  
 (((((первый) 2) третий) 4) 5) ; список из двух  
     ; элементов

В следующих списках соответственно 5 и 4 элемента:

(Добрый день сказал бородатый мужчина)  
 (кот-37 ( кличка Петя) ( цвет ?) ( хвост nil))

**Пустой список = NIL**

Список, в котором нет ни одного элемента, называется **пустым списком** и обозначается "(" или символом **NIL**. Пустой список – это не то же самое, что "ничего". Он

выполняет ту же роль, что и нуль в арифметике. NIL может быть, например, элементом других списков:

<b>NIL</b>	; то же, что и ()
<b>(NIL)</b>	; список, состоящий из атома NIL
<b>()</b>	; то же, что и (NIL)
<b>((()))</b>	; то же, что и ((NIL))
<b>(NIL ())</b>	; список из двух пустых списков

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

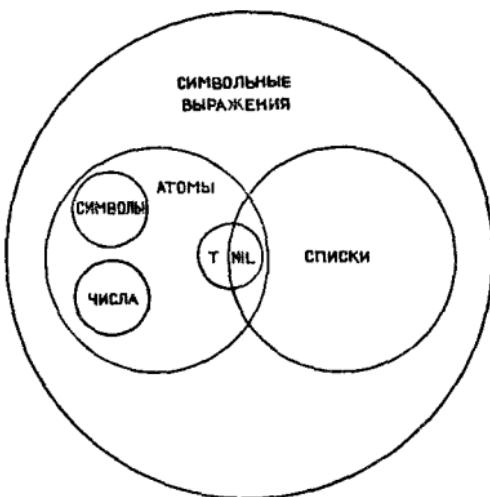


Рис. 2.1.1 Символьные выражения.

### Список как средство представления знаний

Списки можно использовать для представления всевозможных знаний. Например, для характеристики героя комиксов Zippy the Pinhead подошло бы следующее выражение:

(герой-56  
 (имя Zippy)  
 (кличка Pinhead)  
 (язык английский)  
 (приметы

**(голова грушевидная)  
(волосы редкие))  
(встречается Лисп-машины и комиксы)**

Запись в виде списка гибка, свободна по форме и одновременно достаточно точна и понятна. Как видно из примера, такая запись с раскрывающим структуру расположением текста помогает получить общее представление о структуре и вникнуть в нее. В Лисп-системах существует специальная *структурная печать* (*pretty-printer*), которая выводит списки в красивом удобочитаемом виде. При вводе выражений в Лисп-систему пустые строки, пробелы и расположение текста игнорируются.



*Древнеиндийское дерево жизни.*

### Значение способа записи

Планирование способа представления данных и знаний и решение проблемы их изображения являются важнейшими моментами при программировании на языке Лисп. Речь идет о нахождении способа записи, который бы облегчал программирование и был естественней как для описания задачи, так и для работы с данными. Часто удается предложить форму записи, в какой-то мере аналогичную самой задаче и позволяющую упростить дальнейшую работу.



Программируя на языке Лисп, задачу обычно пробуют разбить на более простые составляющие. В структуре данных пытаются максимально отобразить все, что известно о задаче, и, таким образом, упростить программирование. Это позволяет добавлять в систему новые знания, не меняя программу, применять ту же программу с разными знаниями и решать сложные задачи с помощью простых программ.

### Различная интерпретация списков

Одним из основных отличий языка Лисп от традиционных языков программирования является запись в

Лиспе в виде списков не только данных, но и программ (или функций). Например, список: (+ 2 3) можно интерпретировать в зависимости от окружения и использования либо как действие, дающее в результате число 5, либо как список, состоящий из трех элементов.



Способ интерпретации определяется положением выражения и алгоритмом функционирования Лисп-системы. В дальнейшем мы еще вернемся к этой важной особенности языка Лисп, в том числе к взаимосвязям между действием интерпретатора языка Лисп и программированием, управляемым данными.

### Упражнения

1. Сколько элементов самого верхнего уровня в следующих списках:
  - a) ((1 2 3))
  - b) ((a b) c (d (e)))
  - c) (a ((( ))) nil nil)
  - d) (((((a (b (c d) e) f) g) h ((i (j) k) l) m) n))
  
2. Замените в следующих списках все пустые списки на символ пустого списка NIL:
 
  - a) (( ))
  - b) (( ) ( ))
  - c) ((( ) ( ) ( )) ( ))
  
3. Чем отличаются:
  - a) атомы и символы?
  - b) переменные и символы?
  - c) выражения и списки?
  
4. Как можно записать в виде списков выражения логики высказываний, образованные при помощи логических операций НЕ, ИЛИ, И, => и <=>.
  
5. Предложите форму записи, при помощи которой можно представить любую электрическую схему, составленную из сопротивлений.

*Математика – это ключ  
к другим наукам.*

*(Et harum scientiarum porta et  
clavis est Mathematica.)*

*P. Бэкон*

## 2.2 ПОНЯТИЕ ФУНКЦИИ

- Функция – отображение между множествами
- Тип аргументов и функций
- Определение и вызов функции
- Единообразная префиксная нотация
- Аналогия между Лиспом и естественным языком
- Диалог с интерпретатором Лиспа
- Иерархия вызовов
- QUOTE блокирует вычисление выражения
- Упражнения

**Функция – отображение между множествами**

Функцией в математике называется *отображение* (mapping), которое однозначно отображает одни значения на другие. Например, запись

$$y=f(x)$$

ставит в соответствие каждому элементу  $x$  из *множества определения* (domain) единственный элемент  $y$  из *множества значений* (range, codomain) функции  $f$ . Это соответствие также можно записать в следующем виде:

$$f(x) \rightarrow y$$



Будем говорить, что функция  $f$  от *аргумента*  $x$  имеет *значение*  $y=f(x)$  (value, image).

У функции может быть произвольное количество аргументов, в том числе их может не быть совсем. Функция без аргументов имеет постоянное значение. Приведем примеры функций:

$abs(-3)$	$\rightarrow 3$	; абсолютная величина числа
$+(2,3)$	$\rightarrow 5$	; сумма
$union((a,b),(c,b))$	$\rightarrow (a,b,c)$	; объединение множеств
$финский(John)$	$\rightarrow \text{ложь}$	; определение языка
$дети(адам,ева)$	$\rightarrow (\text{каин},\text{авель})$	; множество детей

### Тип аргументов и функций

В общем случае функция может задавать отображение из нескольких множеств в множество значений. Для изображения *типов* (type) значений функции и ее аргументов позаимствуем еще одно принятное в математике обозначение. Пусть  $f$  - функция от двух аргументов:

$f(x,y) \rightarrow z$  ; отображение пар элементов  $x,y$  в множество  $z$

Типы аргументов  $x$  и  $y$ , а также тип значения  $z$  можно обозначить следующим образом:

$f: A \times B \rightarrow C$

Здесь знак " $\times$ " между  $A$  и  $B$  обозначает прямое произведение множеств. Приведенная запись означает, что тип первого аргумента функции  $f$  есть  $A$  (т.е. он принадлежит множеству  $A$ ), тип второго аргумента –  $B$  и тип значения функции –  $C$ . Функция  $f$  ставит в соответствие упорядоченным парам, образованным из элементов множеств  $A$  и  $B$ , элемент из множества  $C$ .

Приведем типы значений и аргументов функций из предыдущего примера:



ФУНКЦИЯ	АРГУМЕНТЫ	ЗНАЧЕНИЯ
<i>abs:</i>	число	→ число
<i>+:</i>	число × число	→ число
<i>union:</i>	множ-во × множ-во	→ множество
<i>финский:</i>	слово	→ логическое значение
<i>дети:</i>	человек × человек	→ множество людей

### Определение и вызов функции

Определение (definition) функции в языке Лисп позволяет задать произвольное вычислимое отображение. Например, функцию, вычисляющую сумму квадратов, можно определить с помощью сложения и умножения:

**суммаквадратов:**

аргументы:  $x, y$   
значение:  $x*x+y*y$

Здесь  $x$  и  $y$  переменные, представляющие произвольные числа.

 Вызов (call) функции означает запуск, или применение (apply) определения функции к конкретным значениям аргументов.

Например, применим функцию суммаквадратов к аргументам  $x=2$  и  $y=3$ :

суммаквадратов( $2, 3$ ) → 13

### Единообразная префиксная нотация

В математике и обычных языках программирования вызов функции записывается, как правило, в так называемой префиксной записи (нотации), в которой имя функции стоит перед скобками, окружающими аргументы:

$f(x)$ ,  $g(x, y)$ ,  $h(x, g(y, z))$  и т. д.

В арифметических выражениях используется инфиксная запись, в которой имя функции или действия располагается между аргументами:

$x+y$ ,  $x-y$ ,  $x*(y+z)$  и т. д.

В языке Лисп как для вызова функции, так и для записи выражений принята единообразная префиксная форма записи, при которой как имя функции или действия, так и сами аргументы записываются внутри скобок:

$(f\ x)$ ,  $(g\ x\ y)$ ,  $(h\ x\ (g\ y\ z))$  и т. д.

Таким же образом записываются арифметические действия. Приведенные выше арифметические выражения в лисповской префиксной записи выглядели бы так:

$(+ x y)$ ,  $(- x y)$ ,  $(* x (+ y z))$  и т. д.

Сначала этот новый способ записи покажется затруднительным, но он весьма быстро станет привычным и даже понравится по мере того, как станут очевидны его преимущества.

Лисп содержит также и средства, необходимые для задания способа записи и расширения языка в желаемом направлении. Можно использовать традиционный способ записи либо придумать себе новый способ. Так как все составлено из списков, то преобразование из одной нотации в другую довольно просто. Некоторые Лисп-системы позволяют без дополнительного программирования использовать арифметические операции в инфиксной записи и без излишних скобок.

Единообразная и простая структура вызова функции удобна как для программиста, так и для вычисляющего значения выражений интерпретатора Лиспа. Вычисляя значения функций, не нужно осуществлять сложный синтаксический анализ выражений, так как сразу по первому символу текущего выражения систе-

ма уже знает, с какой структурой имеет дело и как ее интерпретировать или как с ней обращаться.

### **Аналогия между Лиспом и естественным языком**

В математике обычно используются однобуквенные символы. Поскольку букв мало, стали использовать специальные символы, знаки препинания, индексы,

Шведский язык это  
обычные слова в той же  
форме, в которой  
они в этом языке  
здесь также говорят  
в различных видах  
или написанных письмом

греческие буквы и другие способы записи. В Лиспе (как и во многих других языках программирования) объекты можно называть их естественными именами. И так как в обычном языке

обращение к функции выглядит примерно так же, как в Лиспе, то при использовании в качестве имен функций и аргументов обыкновенных слов, получающиеся выражения начинают напоминать фразы естественного языка. Например:

**(переведи шведский (Знаю шведский))**

Запись вызова функции не использует такого числа технических деталей и соглашений, как это обычно принято в языках программирования. Если не принимать во внимание используемые для выделения структур скобки и искусственное объединение нескольких слов, то с точки зрения записи нет особой разницы между Лиспом и естественным языком.

### **Диалог с интерпретатором Лиспа**

Лисп напоминает естественный язык и тем, что он обычно используется в диалоге (интерактивно) и в

режиме *интерпретации* (*interpretation*).

Интерпретатор Лиспа функционирует следующим образом. Когда пользователь заканчивает ввод какого-либо вызова функции или другого выражения, то интерпретатор *вычисляет* (*evaluate*) и выдает значение этого выражения. Если мы хотим, например, вычислить выражение (+ 2 3), то введем его в машину и сразу получим результат:



```
_(+ 2 3) ; пользователь вводит
; S-выражение
5           ; интерпретатор вычисляет и
; выдает результат
```

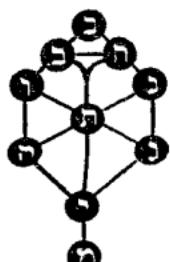
Символ подчеркивания “\_”, показанный перед вводимым выражением, – это так называемое *приглашение* (prompt), при помощи которого интерпретатор дает знать, что он выполнил вычисление предыдущего выражения и ждет новое выражение<sup>1)</sup>:

```
_(- 3 2) ; вычитание
1
-
; интерпретатор выдает на экран
; приглашение и ждет ввода
; очередного выражения
```

### Иерархия вызовов

В конструкцию вводимой функции могут, в свою очередь, входить функциональные подвыражения. Тогда аргументы вычисляемой функции заменяются на

новые вычисления, ведущие к определению значений этих аргументов. Таким путем вычисления сводятся в конечном счете к вычислению выражений, значения которых, как, например констант, можно определить непосредственно. Приведем пример:



$(* (+ 1 2) (+ 3 4))$

Вычисляя значения выражения, интерпретатор Лиспа сначала пытается слева направо вычислить значения аргументов внешнего вызова. Если первый аргумент является константой или иным объектом, значение

<sup>1)</sup> Вид приглашения может не только видоизменяться от системы к системе, но и варьироваться в рамках данной системы, неся информацию о режиме работы интерпретатора.– *Прим. ред.*

которого можно определить непосредственно, то вычисление аргументов вычисляемой функции можно продолжить. Если аргументом будет вложенный вызов функции или другая вычислимая форма, то перед ее вычислением определяются значения ее аргументов и т. д.

Наконец, опустившись на самый нижний уровень, можно начать в качестве значений аргументов предыдущего уровня подставлять значения, даваемые вложенными вычислениями. Последним возвращается значение всего выражения. Как и в математике, в Лиспе в первую очередь вычисляются выражения, заключенные в скобки:

```
(* (+ 1 2) (+ 3 4))
21 ; =(1+2)*(3+4)
(* (+ 1 2) (* 1 (+ 2 3)))
15 ; =(1+2)*(1*(2+3))
```

Позже мы увидим, что в Лиспе можно задавать и произвольный порядок вычислений.

### **QUOTE блокирует вычисление выражения**

В некоторых случаях не надо вычислять значение выражения, а нужно само выражение. Нас, например, может не интересовать значение функционального вызова  $(+ 2 3)$ , равное 5, а мы хотим обрабатывать форму  $(+ 2 3)$  как список. В таких случаях выражения, которые интерпретатору не надо вычислять, помечаются особым образом. Чтобы предотвратить вычисление значения выражения, нужно перед этим выражением поставить апостроф  $''$  (quote). В Лиспе начинающееся с апострофа выражение соответствует заключению в кавычки в обыкновенном тексте, и в обоих случаях это означает, что цитату нужно использовать в том виде, как она есть. Например:

```
'(+ 2 3)
(+ 2 3)
'y
'y
```

Апостроф перед выражением – это на самом деле сокращение лисповской формы QUOTE, записываемой в единообразной для Лиспа префиксной нотации:

**'выражение  $\Leftrightarrow$  (QUOTE выражение)**

QUOTE можно рассматривать как специальную функцию с одним аргументом, которая ничего с ним не делает, даже не вычисляет его значение, а возвращает в качестве значения вызова сам этот аргумент.

```
(quote (+ 2 3))
(+ 2 3)
(quote y)
y
```

Интерпретатор Лиспа, считывая начинающееся с апострофа выражение, автоматически преобразует его в соответствующий вызов функции QUOTE.

```
'(a b '(c d)) ; QUOTE внутри QUOTE
(A B (QUOTE (C D)))
(quote 'y) ; все апострофы преобразуются
(QQUOTE Y) ; в вызов QUOTE
' 'y
(QQUOTE Y)
(quote quote)
QUOTE
'quote
QUOTE
```

Числа не надо предварять апострофом, так как интерпретатор считает, что число и его значение совпадают. Однако в принципе использование апострофа перед числом не запрещено. Перед другими константами (T и NIL), как и перед числами, тоже не надо ставить апостроф, поскольку и они представляют самих себя.

Однако заметим, что в приведенных примерах на месте имени функции в функциональном вызове апо-



```

'317.0 ; апостроф не нужен
317.0
(+ '2 3)
5
t      ; значением Т является Т
T
't      ; апостроф не нужен
T
nil    ; NIL представляет самого себя
NIL

```

строф не использовался. Интерпретатор Коммон Лиспа не допускает использование в качестве первого элемента вызова функции вычисляемого выражения. В некоторых других Лисп-системах имя функции допустимо задавать с помощью вычисляемого выражения.

### Упражнения

- Сколько значений может быть у функции? Может ли функция принимать одно и то же значение на разных аргументах?
- Проделайте следующие вычисления с помощью интерпретатора Лиспа:
  - $3.234 * (4.56 + 21.45)$
  - $5.67 + 6.342 + 12.97$
  - $(454 - 214 - 675) / (456 + 678 * 3)$
- Напишите выражение, вычисляющее среднее арифметическое чисел 23, 5, 43 и 17.
- Определите значения следующих выражений:
  - $'(+ 2 (* 3 4))$
  - $(+ 2 '(* 3 4))$
  - $(+ 2 ('* 3 4))$
  - $(+ 2 (* 3 '4))$
  - $(quote 'quote)$
  - $(quote 2)$
  - $'(quote NIL)$



*Необходимо лишь объединение.*

*Э. Форстер*

## 2.3 БАЗОВЫЕ ФУНКЦИИ

- Основные функции обработки списков
- Функция CAR возвращает в качестве значения головную часть списка
- Функция CDR возвращает в качестве значения хвостовую часть списка
- Функция CONS включает новый элемент в начало списка
- Связь между функциями CAR, CDR и CONS
- Предикат проверяет наличие некоторого свойства
- Предикат ATOM проверяет, является ли аргумент атомом
- EQ проверяет тождественность двух символов
- EQL сравнивает числа одинаковых типов
- Предикат = сравнивает числа различных типов
- EQUAL проверяет идентичность записей
- EQUALP проверяет наиболее общее логическое равенство
- Другие примитивы
- NULL проверяет на пустой список
- Вложенные вызовы CAR и CDR можно записывать в сокращенном виде
- LIST создает список из элементов
- Упражнения

В предыдущем разделе мы ознакомились с функционированием Лиспа с точки зрения общеизвестных арифметических функций. Однако с самого начала Лисп был предназначен не для проведения операций с числами, а для работы с символами и списками.

## Основные функции обработки списков

В Лиспе для построения, разбора и анализа списков существуют очень простые базовые функции, которые в этом языке являются примитивами. В определенном смысле они образуют систему аксиом языка (алгебру обработки списков), к которым в конце концов сводятся символические вычисления. Базовые функции обработки списков можно сравнить с основными действиями в арифметических вычислениях или в теории чисел.

Простота базовых функций и их малое число

– это характерные черты Лиспа. С этим связана математическая элегантность языка.

 Разумно выбранные примитивы образуют, кроме красивого формализма, также и практическую основу программирования.

Базисными функциями обработки символьных выражений (атомов и списков) являются:

### CAR, CDR, CONS, ATOM и EQ

Функции по принципу их использования можно разделить на функции разбора, создания и проверки:

#### ИСПОЛЬЗОВАНИЕ ВЫЗОВ

Разбор:  
 (CAR список)  
 (CDR список)

Создание: (CONS s-выражение список)

Проверка: (ATOM s-выражение)  
 (EQ символ символ)

#### РЕЗУЛЬТАТ

s-выражение  
 список

список

Т или NIL  
 Т или NIL

У функций CONS и EQ имеются два аргумента, у остальных примитивов – по одному. В качестве имен аргументов и результатов функций мы использовали названия типов, описывающих аргументы, на которых определена (т.е. имеет смысл) функция и вид возвращаемого функциями результата. S-выражение обозначает атом или список.

Функции ATOM и EQ являются базовыми предикатами. Предикаты (predicate) – это функции, которые проверяют выполнение некоторого условия и возвращают

ют в качестве результата логическое значение Т (в более общем виде, произвольное выражение, отличное от NIL) или NIL.

Рассмотрим теперь базовые функции более подробно.

### **Функция CAR возвращает в качестве значения головную часть списка**

Первый элемент списка называется *головой* (head), а остаток списка, т.е. список без первого его элемента, называется *хвостом* списка (tail). С помощью селекторов CAR и CDR можно выделить из списка его голову и хвост.



Функция CAR (произносится "кар") возвращает в качестве значения первый элемент списка, т.е. головной элемент списка:

```
(car '(первый второй третий)) ; внимание,
ПЕРВЫЙ ; используется '
  (car '((head) tail))
  HEAD
```

Обратите внимание на то, что в первом примере мы не вычисляем значение аргумента (ПЕРВЫЙ ВТОРОЙ ТРЕТИЙ) и не применяем к нему операцию CAR, а используем аргумент в таком виде, как он есть. Ситуация отличается, например, от следующего функционального вызова:

```
(* 4 (+ 3 2))
20
```

где нужно умножить 4 на значение аргумента (+ 3 2), т.е. на 5, а не на список (+ 3 2). Вызов функции CAR с аргументом (ПЕРВЫЙ ВТОРОЙ ТРЕТИЙ) без апострофа был бы проинтерпретирован как вызов функции

**ПЕРВЫЙ** с аргументами **ВТОРОЙ** и **ТРЕТИЙ**, и было бы получено сообщение об ошибке<sup>1)</sup>.

Функция **CAR** имеет смысл только для аргументов, являющихся списками, а следовательно, имеющих голову:

*car: список → s-выражение*

Для аргумента атома результат функции **CAR** неопределен, и вследствие этого появляется следующее сообщение об ошибке:

```
(car 'кот)
Error: KOT is not a list
```

(т.е. КОТ не является списком. – Ред.).

Головной частью пустого списка считают для удобства **NIL**:

```
(car nil)      ; голова пустого списка –
NIL           ; пустой список
(car 'nil)    ; знак ' можно опускать
NIL
(car '(nil a)) ; голова списка NIL
NIL
```

**Функция CDR возвращает в качестве значения хвостовую часть списка**

Функция **CDR** (произносится "кудр") применима к спискам. Значением ее будет хвостовая часть списка, т.е. список, получаемый из исходного списка после удаления из него головного элемента:

*cdr: список → список*

<sup>1)</sup> Так, функция **ПЕРВЫЙ** в системе не задана. Заметим, что в Коммон Лиспе имеется встроенная функция **FIRST**, эквивалентная **CAR**. – Прим. ред.



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

Функция CDR не выделяет второй элемент списка, а берет весь остаток списка, т.е. хвост. Заметим, что хвост списка – тоже список, если только список не состоял из одного элемента. В последнем случае хвостом будет пустой список (), т.е. NIL:

```
(cdr '(a))
NIL
```

Из соображений удобства значением функции CDR от пустого списка считается NIL:

```
(cdr nil)
NIL
```

Так же как и CAR, функция CDR определена только для списков. Значение для атомов не определено, что может приводить к сообщению об ошибке

```
(cdr 'кот)
Error: KOT is not a list
```

CAR	SXA PDX CLA PAX	Необычные имена функций CAR и CDR возникли по историческим причинам. Эти имена являются сокращениями от наименований регистров Contents of Address Register (CAR) и Contents of Decrement Register (CDR) вычислительной машины IBM 605. Автор Лиспа Джон Маккарти (США) реализовал первую Лисп-систему именно на этой машине и использовал регистры CAR и CDR для хранения головы и хвоста списка.
CARX	PXD AXT TRA	
CDR	SXA PDX CLA PDX	
CDRX	PXD AXT TRA	

В Коммон Лиспе и на некоторых других диалектах Лиспа наряду с именами CAR и CDR используют и более наглядные имена FIRST и REST<sup>1)</sup>.

```
(first '(1 2 3 4))
1
(first (rest '(1 2 3 4))) ; композиция
2 ; вызовов
(first '(rest '(1 2 3 4)))
REST
```

(В некоторых системах используются имена HEAD и TAIL.)

**Функция CONS включает новый элемент в начало списка**

Функция CONS (construct) строит новый список из переданных ей в качестве аргументов головы и хвоста:



(CONS *голова* *хвост*)

Функция добавляет новое выражение в список в качестве первого элемента:

```
(cons 'a '(b c))
(A B C)
(cons '(a b) '(c d))
((A B) C D)
(cons '(Mauno Koivisto)
      '((Urho Kekkonen) ...))
((MAUNO KOIVISTO) ((URHO KEKKONEN) ...))
(cons (+ 1 2) '(+ 4)) ; первый аргумент
(3 + 4) ; без апострофа, поэтому
; берется его значение
(cons '(+ 1 2) '(+ 4)) ; первый аргумент
((+ 1 2) + 4) ; не вычисляется
```

<sup>1)</sup> Т.е. первый и остаток.— Прим. ред.

У начинающих при манипуляциях с пустым списком очень легко возникают смысловые ошибки. Проследите за тем, что происходит:

```
(cons '(a b c) nil)
((A B C))      ; однозлементный список
(cons nil '(a b c))
(NIL A B C)    ; NIL - элемент списка
(cons nil nil)
(NIL)           ; не то же самое, что NIL
```

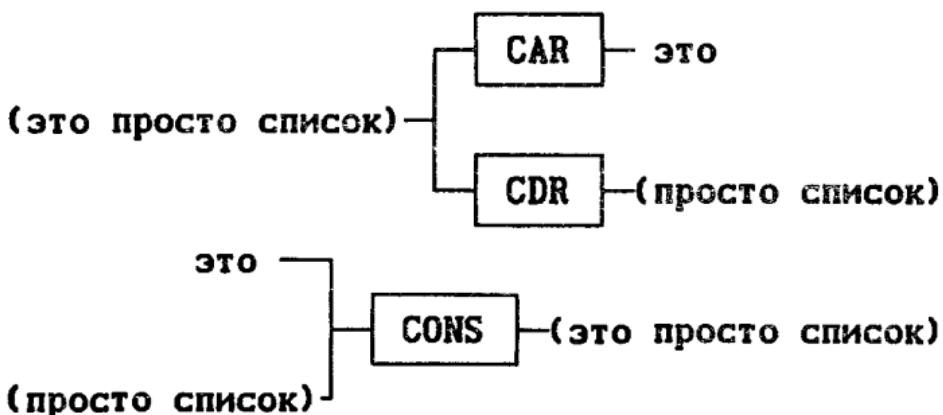
Для того чтобы можно было включить первый аргумент функции CONS в качестве первого элемента значения второго аргумента этой функции, второй аргумент должен быть списком. Значением функции CONS всегда будет список:

*cons: s-выражение × список → список*

(Позже мы увидим, что вторым аргументом может быть и атом, но в таком случае результатом будет более общее символьное выражение, так называемая точечная пара (dotted pair).)

### Связь между функциями CAR, CDR и CONS

Селекторы CAR и CDR являются обратными для конструктора CONS. Список, разбитый с помощью функций CAR и CDR на голову и хвост, можно восстановить с помощью функции CONS:



Эту связь между функциями CAR, CDR и CONS можно выразить следующим образом:

```
(cons (car '(это просто список))
      (cdr '(это просто список)))
(ЭТО ПРОСТО СПИСОК)
```

**Предикат проверяет наличие некоторого свойства**

Чтобы осуществлять допустимые действия со списками и избежать ошибочных ситуаций, нам необходимы, кроме селектирующих и конструирующих функций, средства опознавания выражений. Функции, решающие эту задачу, в Лиспе называются *предикатами* (predicate).

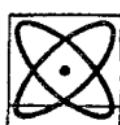


Предикат – это функция, которая определяет, обладает ли аргумент определенным свойством и возвращает в качестве значения логическое значение "ложь", т.е. NIL, или "истина", которое может быть представлено символом Т или любым выражением, отличным от NIL.

ATOM и EQ являются базовыми предикатами Лиспа. С их помощью и используя другие базовые функции, можно задать более сложные предикаты, которые будут проверять наличие более сложных свойств.

**Предикат ATOM проверяет, является ли аргумент атомом**

При работе с выражениями необходимо иметь возможность проверить, является ли выражение атомом или списком. Это может потребоваться, например, перед применением функций CAR и CDR, так как эти функции определены лишь для аргументов, являющихся списками. Базовый предикат ATOM используется для идентификации лисповских объектов, являющихся атомами:



**(ATOM s-выражение)**

Значением вызова ATOM будет Т, если аргументом является атом, и NIL – в противном случае.

```
(atom 'x)
T ; x - это атом
(atom '(я программирую -
        следовательно существую))
NIL
(atom (cdr '(a b c)))
NIL ; ATOM от списка (B C) - это NIL
```

С помощью предиката ATOM можно убедиться, что пустой список NIL, или (), является атомом:

```
(atom nil)
T
(atom ())
T ; () - то же самое, что и NIL
(atom '())
T ; Пустой список с апострофом
T ; все равно есть NIL
(atom '(nil))
NIL ; Аргумент - однозлементный
(atom (atom (+ 2 3)))
T ; Логическое
                ; значение T является атомом
```

В последнем примере результатом сложения является число, а результатом внутреннего вызова предиката ATOM – значение Т, которое, в свою очередь, тоже является атомом.

В Лиспе существует целый набор предикатов, проверяющих тип являемого аргументом выражения или любого другого лисповского объекта и таким образом идентифицирующих используемый тип данных. Например, LISTP идентифицирует списки, ARRAPP – массивы (array) и т. д. Позднее мы рассмотрим типы данных более подробно.

### **EQ проверяет тождественность двух символов**

Предикат EQ сравнивает два символа и возвращает значение Т, если они идентичны, в противном случае – NIL.

```
(eq 'x 'кот)
NIL
(eq 'кот (car '(кот пес)))
T
(eq () nil)
T
(eq t 't)
T
(eq t (atom 'мышь))
```

Предикат EQ накладывает на свои аргументы строго определенные требования. С его помощью можно сравнивать только символы или константы T и NIL, и результатом будет значение T лишь в том случае, когда



аргументы совпадают. Для проверки чисел в Коммон Лиспе EQ не используется. Предикатов ATOM и EQ, несмотря на простоту EQ, вполне достаточно для работы со списками.

Предикат EQ в Лисп-системах обычно таков, что его можно применять к списочным и числовым аргументам, не получая сообщения об ошибке; он не проверяет логического равенства чисел, строк или других объектов, а лишь смотрит, представлены ли лисповские объекты в памяти вычислительной машины физически одной и той же структурой. Одноименные символы представлены в одном и том же месте памяти (не считая нескольких исключений), так что той же проверкой предикатом EQ символы можно сравнить логически. До сих пор, например, списки могли быть логически (внешне) одинаковы, но они могут состоять из физически различных списочных ячеек. К вопросам физического и логического равенства списков мы вернемся позже. Приведем все-таки для предостережения два примера на следующей странице.

Так как EQ определен лишь для символов, то, сравнивая два выражения, прежде всего надо определить, являются ли они атомами (ATOM). Если хотя бы один из аргументов является списком, то предикат EQ

```
(eq '(a b c) '(a b c))
NIL
(eq 3.14 3.14)
NIL
```

нельзя использовать для логического сравнения. При сравнении чисел проблемы возникают с числами различных типов. Например, числа 3.000000, 3 и 0.3E1 логически представляют одно и то же число, но записываются внешне неодинаково. Равенство выражений можно понимать по-разному. Для различных видов и степеней равенства в Лиспе наряду с EQ используются и другие предикаты, которые мы сейчас рассмотрим.

### **EQL сравнивает числа одинаковых типов**

Более общим по сравнению с EQ является предикат EQL, который работает так же, как EQ, но дополнительно позволяет сравнивать однотипные числа (и элементы строк).

```
(eql 3 3)      ; сравниваются два целых
T
; числа
(eql 3.0 3.0) ; сравниваются два
T
; вещественных числа
(eql 3 3.0)   ; не годится для разных
NIL           ; типов
```

Предикат EQL, как правило, используется во многих встроенных функциях, осуществляющих более сложные операции сравнения. Его использование для сравнения списков – это часто встречающаяся ошибка.

### **Предикат = сравнивает числа различных типов**

Сложности, возникающие при сравнении чисел, легко преодолимы с помощью предиката =, значением которого является T в случае равенства чисел независимо от их типов и внешнего вида записи:

```
(= 3 3.0)
T
 (= 3.00 0.3e1)
T
```

Обеспечить применение предиката к числовым аргументам может предикат NUMBERP, который истинен для чисел:

```
(numberp 3e-34)
T
 (numberp t)
NIL
```

### **EQUAL проверяет идентичность записей**

Обобщением EQL является предикат EQUAL. Он работает как EQL, но, кроме того, проверяет одинаковость двух списков:

```
(equal 'x 'x)
T
(equal '(x y z) '(x y z))
T
(equal '(a b c) (cons 'a '(b c)))
T
(equal '(nil) '((nil)))
NIL
```

Принцип работы предиката EQUAL состоит в следующем: если внешняя структура двух лисповских объектов одинакова, то эти объекты между собой равны в смысле EQUAL. Предикат EQUAL также применим к числам и к другим типам данных (например, к строкам). Заметим, что в соответствии со своим принципом работы он не подходит для сравнения разнотипных чисел, так как их внешние представления различаются:

(equal 3.00 3)	; различное внешнее
<b>NIL</b>	; представление
(= 3.00 3)	
<b>T</b>	

### **EQUALP проверяет наиболее общее логическое равенство**

Наиболее общим предикатом, проверяющим в Коммон Лиспе наличие логического равенства, является EQUALP, с помощью которого можно сравнивать произвольные лисповские объекты, будь то числа различных типов, выражения или другие объекты. Этот предикат может потребоваться, когда нет уверенности в типе сравниваемых объектов или в корректности использования других предикатов сравнения.



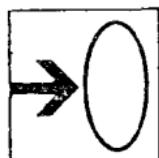
Недостатком универсальных предикатов и функций типа EQUALP является то, что их применение требует от системы несколько большего объема вычислений, чем использование специализированных предикатов и функций.

### **Другие примитивы**

Несмотря на то что обычную обработку списков всегда можно свести к описанным ранее трем базовым функциям (CONS, CAR и CDR) и двум базовым предикатам (ATOM и EQ), программирование лишь с их использованием было бы очень примитивным и похожим на программирование на внутреннем машинном языке. Поэтому в Лисп-систему включено множество встроенных функций для различных действий и различных ситуаций. Далее мы рассмотрим некоторые такие примитивы и их полезные свойства. К более сложному программированию с помощью базовых примитивов мы вернемся подробнее в главе, посвященной функциональному программированию.

**NULL проверяет на пустой список**

Встроенная функция NULL проверяет, является ли аргумент пустым списком:



```
(null '())
T
(null (cddr '(a b c)))
NIL
(null NIL)
T
(null T)
NIL
```

Из последних двух примеров видно, что NULL работает как логическое отрицание, у которого в Лиспе есть и свой, принадлежащий логическим функциям, предикат (NOT x):

```
(not (null nil))
NIL
```

Функции NULL и NOT можно выразить через EQ:

**(NULL x)  $\leftrightarrow$  (EQ NIL x)**

**Вложенные вызовы CAR и CDR можно записывать в сокращенном виде**

Комбинируя селекторы CAR и CDR, можно выделить произвольный элемент списка. Например:

```
(cdr (cdr (car '((a b c) (d e) (f)))))

(C)
```

Комбинации вызовов CAR и CDR образуют уходящие в глубину списка обращения, и в Лиспе используется для этого более короткая запись: желаемую комбинацию вызовов CAR и CDR можно записать в виде одного вызова функции:

**(C...R список)**



Вместо многоточия записывается нужная комбинация из букв A (для функции CAR) и D (для функции CDR):

Алхимический  
знак, означа-  
ющий соеди-  
нение.

$$\begin{array}{lll} (\text{cad}\text{r } x) & \Leftrightarrow & (\text{ca}\text{r } (\text{cd}\text{r } x)) \\ (\text{cdd}\text{ar } x) & \Leftrightarrow & (\text{cd}\text{r } (\text{cd}\text{r } (\text{ca}\text{r } x))) \end{array}$$

Например:

```
A (cadr '(программировать на Лиспе  
просто?))  
B (caddar '((a b c) (d e) (f)))  
C (cadar (cons ' (a b) nil))  
D
```

Для функций CAR, CADR, CADDR, CADDDR и т. д. в Коммон Лиспе используются и более наглядные имена FIRST, SECOND, THIRD, FOURTH и т. д. Можно воспользоваться и наиболее общей функцией NTH, выделяющей п-й элемент списка:

(NTH *n* список)

```
1 (nth 2 '(1 2 3)) ; индексы начинаются с  
2 ; нуля  
3 (third (cons 1 (cons 2 (cons 3 nil))))  
4 (fourth '(1 2 3))  
5 NIL
```

Последний элемент списка можно выделить с помощью функции LAST:

(LAST *x*)

**LIST создает список из элементов**

Другой часто используемой встроенной функцией является



**(LIST x1 x2 x3 ...)**

которая возвращает в качестве своего значения список из значений аргументов. Количество аргументов функции LIST произвольно:

```
(list 1 2)
(1 2)
(list 'a 'b (+ 1 2))
(A B 3)
(list 'a '(b c) 'd)
(A (B C) D)
(list (list 'a) 'b NIL)
((A) B NIL)
(list NIL)
(NIL)
```

Построение списков нетрудно свести к вложенным вызовам функции CONS, причем вторым аргументом последнего вызова является NIL, служащий основой для наращивания списка:

```
(cons 'c NIL)      ; <=> (list 'c)
(C)
_(cons 'b          ; <=> (list 'b 'c)
  (cons 'c NIL))
(B C)
_(cons 'a          ; <=> (list 'a 'b 'c)
  (cons 'b
    (cons 'c NIL)))
(A B C)
```

В дальнейшем станет ясно, как, в частности, функцию LIST описать через комбинацию базовых функций.

### Упражнения

- Перечислите базовые функции языка Лисп. Каковы типы их аргументов, и какие значения они возвращают в качестве результата?
- Запишите последовательности вызовов CAR и CDR, выделяющие из приведенных ниже списков символ "цель". Упростите эти вызовы с помощью функций C...R.
  -  a) (1 2 цель 3 4)  
b) ((1) (2 цель) (3 (4)))  
c) ((1 (2 (3 4 цель))))
- Вычислите значения следующих выражений. Проверьте результат на машине.
  - a) (cons nil '(суть пустой список))  
b) (cons nil nil)  
c) (cons '(nil) '(nil))  
d) (cons (car '(a b)) (cdr '(a b)))  
e) (car '(car (a b c)))  
f) (cdr (car (cdr '(a b c)))))  
g) (list (list 'a 'b) '(car (c d)))
- Какие из следующих вызовов возвращают значение T?
  - a) (atom '(cdr nil))  
b) (equal '(a b) (cons '(a) '(b)))  
c) (atom (\* 2 (+ 2 3)))  
d) (null (null t))  
e) (eq nil (null nil))  
f) (eql 2.0 2)  
g) (equal 2.0 2)  
h) (= 2.0 2)  
i) (equalp 2.0 2)  
j) (equalp (atom nil) (caar '((t))))

— Как твоя фамилия?  
 — Что вы из этого узнаете?  
*И что вы узнаете,  
 если, я скажу, что я  
 Кукконен или Кананен,  
 Питкянен или Пяткянен,  
 Литтунен или Латтунен<sup>1)</sup>.*

Т. Пекканен

## 2.4 ИМЯ И ЗНАЧЕНИЕ СИМВОЛА

- Значением константы является сама константа
- Символ может обозначать произвольное выражение
- SET вычисляет имя и связывает его
- SETQ связывает имя, не вычисляя его
- SETF – обобщенная функция присваивания
- Побочный эффект псевдофункции
- Вызов интерпретатора EVAL вычисляет значение значения
- Основной цикл: READ-EVAL-PRINT
- Упражнения

**Значением константы является сама константа**

Так же как выражение, являющееся вызовом функции, без предшествующего апострофа представляет значение

выражения, а не само выражение, так и атомы могут использоваться для обозначения каких-нибудь значений. Мы уже обращали внимание на то, что перед лисп-овскими константами (числами и символами T и NIL) не надо ставить апостроф.

Как константы они обозначают самих себя. Если мы введем константу, то интерпретатор в качестве результата выдаст саму эту константу:

---

<sup>1)</sup> Эти фамилии образованы от слов: петух, курица, длинный, кусок, плоский.— Прим. перев.



<u>t</u>	; значением Т является его имя
<u>T</u>	
<u>'t</u>	; апостроф излишен
<u>T</u>	
<u>nil</u>	
<u>NIL</u>	
<u>3.14</u>	
<u>3.14</u>	

**Символ может обозначать произвольное выражение**

Символы можно использовать как переменные. В этом случае они могут обозначать некоторые выражения. У символов изначально нет какого-нибудь значения, как у констант. Если, например, введем символ ФУНКЦИИ, то мы получим сообщение об ошибке:

**функции ; у символа нет значения  
Error: Unbound atom ФУНКЦИИ**

Интерпретатор здесь не может вычислить значение символа, поскольку его у него нет.



**SET вычисляет имя и связывает его**

При помощи функции SET символу можно присвоить (set) или связать (bind) с ним некоторое значение. Если, например, мы хотим, чтобы символ ФУНКЦИИ обозначал базовые функции Лиспа, то введем:

**(set 'функции '(car cdr cons atom eq))  
(CAR CDR CONS ATOM EQ)**

Теперь между символом ФУНКЦИИ и значением (CAR CDR CONS ATOM EQ) образована связь (binding), которая действительна до окончания работы, если, конечно, этому имени функцией SET не будет присвоено новое значение. После присваивания интерпретатор уже может вычислить значение символа ФУНКЦИИ:

**функции  
(CAR CDR CONS ATOM EQ)**

Обратите внимание, что SET вычисляет оба аргумента. Если перед первым аргументом нет апострофа, то с помощью функции SET можно присвоить значение имени, которое получается путем вычисления. Например, вызов

**(set (car функции) '(взбрести в голову))  
(ВЗБРЕСТИ В ГОЛОВУ)**

присваивает переменной CAR выражение (ВЗБРЕСТИ В ГОЛОВУ), так как вызов (CAR ФУНКЦИИ) возвращает в качестве значения символ CAR, который и используется как фактический аргумент вызова функции SET:

<b>(car функции)</b>	<b>;</b>	<b>первый аргумент</b>
<b>CAR</b>	<b>;</b>	<b>предыдущего вызова</b>
<b>CAR</b>	<b>;</b>	<b>присвоенное</b>
<b>(ВЗБРЕСТИ В ГОЛОВУ)</b>	<b>;</b>	<b>функцией SET</b>
<b>функции</b>	<b>;</b>	<b>значение</b>
<b>(CAR CDR CONS ATOM EQ)</b>		

На значение символа можно сослаться, записав его без апострофа. Значение имени никак не проявится до тех пор, пока оно не примет участия в вычислениях. Значения символов определяются с помощью специальной функции SYMBOL-VALUE, которая возвращает в качестве своего значения значение символа, являющегося ее аргументом.

**(symbol-value (car функции))  
(ВЗБРЕСТИ В ГОЛОВУ)**

**SETQ связывает имя, не вычисляя его**

Наряду с функцией SET связать символ с его значением можно с помощью функции SETQ. Эта функция отличается от SET тем, что она вычисляет только свой

второй аргумент. Об автоматическом блокировании вычисления первого аргумента напоминает буква Q (quote) в имени функции. Например:

```
(setq функции '(car cdr cons atom eq))
(CAR CDR CONS ATOM EQ)
```



Монограмма императора Юстиниана.

При использовании функции SETQ отпадает надобность в знаке апострофа перед первым аргументом. (В Интерлиспе есть функция SETQQ, которая блокирует вычисление обоих аргументов.)

Проверить, связан ли атом, можно с помощью предиката BOUNDP, который истинен, когда атом имеет какое-нибудь значение:

```
(boundp 'беззначения)
NIL
(boundp 'функции)
T
(boundp 't) ; константа всегда связана
T
```

### SETF – обобщенная функция присваивания

В Коммон Лиспе значение символа сохраняется в ячейке памяти (storage location), связанной с самим символом. Под ячейками памяти при этом понимаются поля списочной ячейки, которую мы рассмотрим ниже, элементы массива и другие структуры, содержащие данные. Так же как на значения символов можно сослаться через их имена, так и на ячейки памяти можно ссылаться через вызов функции SYMBOL-VALUE и в общем случае другими способами, зависящими от типа данных.

Для присваивания, т.е. занесения значения в ячейку памяти, существует обобщенная функция обновления данных SETF, которая записывает в ячейку памяти новое значение:



**(SETF ячейка-памяти значение)**

Через функцию SETF можно представить описанные нами ранее функции SET и SETQ:

$$\begin{aligned} (\text{setq } x \ y) &\Leftrightarrow (\text{setf } x \ y) \\ (\text{set } x \ y) &\Leftrightarrow (\text{setf } (\text{symbol-value } x) \ y) \end{aligned}$$

```
(setf список '(a b c))
(A B C)
список
(A B C)
```

Заметьте, что первый аргумент использован без блокировки вычисления. Переменная СПИСОК без апострофа указывает на ячейку памяти, куда помещается в качестве значения список (A B C).

В дальнейшем мы вернемся к функции SETF при обновлении значений в лисповых объектах различных типов, содержащих в себе ячейки памяти.

**Побочный эффект псевдофункции**

Функции SET, SETQ и SETF отличаются от других рассмотренных функций тем, что помимо того, что они имеют значение, они обладают и побочным эффектом. Эффект функции состоит в образовании связи между символом и его значением, а значением функции является связываемое значение. Символ остается связанным с определенным значением до тех пор, пока это значение не изменят.

 В Лиспе все действия возвращают некоторое значение. Значение имеется даже у таких действий, основное предназначение которых заключается в осуществлении побочного эффекта, таких, например, как связывание символа или вывод результатов на печать. Функции, обладающие побочным эффектом, в Лиспсе называют псевдофункциями. Мы будем все же как для функций, так и для псевдофункций использо-

вать понятие функции, если только нет особой надобности подчеркнуть наличие побочного эффекта.

Вызов псевдофункции, например оператор передачи управления (а это тоже вызов), с точки зрения использования его значения может стоять на месте аргумента другой функции. (В языках программирования, основанных на операторном подходе, это обычно невозможно.)

```
(list (+ (setq a 3) 4) a)
(7 3)
a
3 ; аргументы
(list b (setq b 3)) ; вычисляются
Error: Unbound atom B ; слева направо
```

В практическом программировании псевдофункции полезны и часто необходимы, хотя в теории, чистом функциональном программировании, они не нужны.

### Вызов интерпретатора EVAL вычисляет значение значения

Интерпретатор Лиспа называется EVAL, и его можно так же, как и другие функции вызывать из программы. При обычном программировании вызывать интерпретатор не надо, так как этот вызов неявно присутствует в диалоге программиста с Лисп-системой.



Лишний вызов интерпретатора может, например, снять эффект блокировки вычисления или позволяет найти значение значения выражения, т.е. осуществить двойное вычисление:

```
(quote (+ 2 3))
(+ 2 3)
(eval (quote (+ 2 3)))
5
```

Здесь интерпретатор вычисляет (evaluate) значение первого выражения (QUOTE (+ 2 3)), получая в результате (+ 2 3). Далее вызов EVAL позволяет вновь запустить интерпретатор, и результатом будет значение выражения (+ 2 3), т.е. 5. Исходное выражение обрабатывается в два этапа. Приведем еще несколько примеров:

```
(setq x '(a b c))
(— A B C)
' x
X
(eval 'x)
(— A B C)
(eval x) ; значением X является
Error: Undefined function A ; (A B C)
(eval (quote (quote (a b c))))
(— A B C)
(quote (eval x))
(— EVAL X)
```

QUOTE и EVAL действуют во взаимно противоположных направлениях и аннулируют эффект друг друга.

EVAL – это универсальная функция Лиспа, которая может вычислить любое правильно составленное лисповское выражение. EVAL определяет семантику (semantics) лисповских форм, т.е. определяет, какие символы и формы совместно с чем и что означают и какие выражения имеют значение.



*Птица Ба – изображение души в Древнем Египте.*

Семантику Лиспа можно довольно компактно и метко определить на самом Лиспе. Когда мы будем рассматривать один из примеров программирования, интерпретатор Лиспа, реализующий основные функции обработки списков, уместится на паре листков формата А4. Большая часть системных программ в Лисп-системах написана на Лиспе, и пользователь может легко изменить систему в зависимости от своих потребностей. Новые функции Лиспа можно сделать более эффективными путем их

трансляции на машинный язык. Транслятор также целиком написан на Лиспе.

### Основной цикл: READ-EVAL-PRINT

Диалог с интерпретатором Лиспа на самом верхнем, *командном уровне* (*top level*), можно описать простым циклом:

( <b>print</b> '_)	; вывод приглашения
( <b>setq</b> e ( <b>read</b> ))	; ввод выражения
( <b>setq</b> v ( <b>eval</b> e))	; вычисление его
	; значения
( <b>print</b> v)	; вывод результата
( <b>print</b> '_)	; повторение цикла
...	

Интерпретатор Лиспа отвечает на заданный ему вопрос и ждет новое задание.

Некоторые системы на командном уровне работают в режиме, который называют EVALQUOTE. В этом режиме имя функции можно выносить за открывающую скобку, как это обычно принято при записи математических функций. Кроме того, как это видно из названия, автоматически блокируется вычисление аргументов. Например:

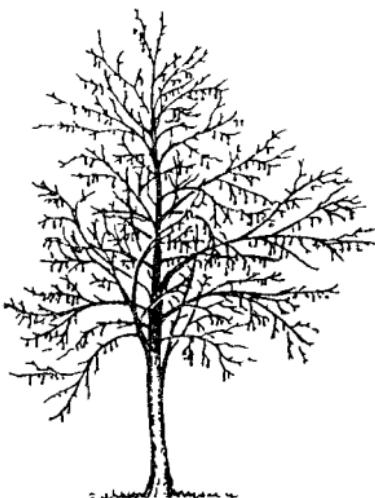
+ (2 3)	⇒ 5
<b>cons</b> (a (b c))	⇒ (a b c)

Хотя при этом можно избежать использования знака апострофа, но в то же время на командном уровне пропадает простая возможность задать вычисляемые аргументы функции. Для аргументов, требующих вычисления, можно либо использовать способ отмены блокировки вычислений, либо в самой функции специально использовать вызов интерпретатора (EVAL).



### Упражнения

1. Каковы будут значения следующих вызовов при условии, что значением X является Y, а значением Y – X:
  - a)   (set x y)
  - b)   (setq x y)
  - c)   (setf x y)
2. Каково будет значение атома A после следующих вызовов:
  - a)   (set (setq a 'a) 'b)  
      b
  - b)   (set (setq b 'a) (setq a 'c))  
      c  
        (set b a)
3. Вычислите значения следующих выражений:
  - a) '(car '(a b c))
  - b) (eval '(car '(a b c)))
  - c) (eval (car '(a b c)))
  - d) (eval (quote (quote quote)))
  - e) (quote (eval (quote (quote quote)))))



*Не существует фактов, есть лишь их интерпретация.*

Ф. Ницше

## 2.5 ОПРЕДЕЛЕНИЕ ФУНКЦИЙ

- **Лямбда-выражение изображает параметризованные вычисления**
- **Лямбда-вызов соответствует вызову функции**
- **Вычисление лямбда-вызова или лямбда-преобразование**
- **Объединение лямбда-вызовов**
- **Лямбда-выражение – функция без имени**
- **DEFUN дает имя описанию функции**
- **SYMBOL-FUNCTION выдает определение функции**
- **Задание параметров в лямбда-списке**
- **Изображение функций в справочных руководствах**
- **Функции вычисляют все аргументы**
- **Многозначные функции**
- **Определение функций в различных диалектах Лиспа**
- **При вычислении NLAMBDA аргументы не вычисляются**
- **Упражнения**

**Лямбда-выражение изображает параметризованные вычисления**

Определение функций и их вычисление в Лиспе основано на **лямбда-исчислении** (lambda calculus) Черча, предлагающем для этого точный и простой формализм. Лямбда-выражение, позаимствованное Лиспом из лямбда-исчисления является важным механизмом в практическом программировании. Подробнее мы вернемся к этому позже при рассмотрении функционалов.

В лямбда-исчислении Черча функция записывается в следующем виде:

*lambda(x<sub>1</sub>, x<sub>2</sub>, ..., x<sub>n</sub>).fn*

В Лиспе лямбда-выражение имеет вид

**(LAMBDA (x<sub>1</sub> x<sub>2</sub> ... x<sub>n</sub>) fn)**

Символ LAMBDA означает, что мы имеем дело с определением функции. Символы *x<sub>i</sub>* являются *формальными параметрами* (formal parameter) определения, которые именуют аргументы в описывающем вычисления *теле* (body) функции *fn*. Входящий в состав формы список, образованный из параметров, называют *лямбда-списком* (lambda list).

λ

*Лямбда.*  
*Однинадцатая буква греческого алфавита, соответствующая букве "Л", численное значение = 30.*

Телом функции является произвольная форма, значение которой может вычислить интерпретатор Лиспа, например: константа, связанный со значением символ или композиция из вызовов функций. Функцию, вычисляющую сумму квадратов двух чисел можно, например, определить следующим лямбда-выражением:

**(lambda (x y) (+ (\* x x) (\* y y)))**

Формальность параметров означает, что их можно заменить на любые другие символы, и это не отразится на вычислениях, определяемых функцией. Именно это и скрывается за лямбда-нотацией. С ее помощью возможно различать понятия определения и вызова функции. Например, функцию LIST для двух аргументов можно определить любым из двух последующих лямбда-выражений:

**(lambda (x y) (cons x (cons y nil)))**  
**(lambda (кот пес) (cons кот (cons пес nil)))**

Здесь значение вызова функции CONS, являющегося телом лямбда-выражения, зависит от значений связей (другими словами, от значений переменных).

## Лямбда-вызов соответствует вызову функции

Лямбда-выражение – это определение вычислений и параметров функции в чистом виде без *фактических параметров*, или *аргументов* (*actual parameter*). Для того, чтобы применить такую функцию к некоторым аргументам, нужно в вызове функции поставить лямбда-определение на место имени функции:

(*лямбда-выражение a<sub>1</sub> a<sub>2</sub> ... a<sub>n</sub>*)

Здесь *a<sub>i</sub>* – формы, задающие фактические параметры, которые вычисляются как обычно. Например, действие сложения для чисел 2 и 3

$$\begin{array}{r} (+ \ 2 \ 3) \\ \hline 5 \end{array}$$

можно записать с использованием вызова лямбда-выражения:

```
—((lambda (x y)
  (+ x y)) ; лямбда-определение
  2 3)          ; аргументы
  5              ; результат
```

Следующий вызов строит список из аргументов A и B:

```
—((lambda (x y) (cons x (cons y nil)))
  'a 'b)
  (A B)
```

Такую форму вызова называют лямбда-вызовом.

### Вычисление лямбда-вызова, или лямбда-преобразование

Вычисление лямбда-вызыва, или применение лямбда-выражения к фактическим параметрам, производится в два этапа. Сначала вычисляются значения фактических параметров и соответствующие формальные

параметры связываются с полученными значениями. Этот этап называется *связыванием параметров* (spreading). На следующем этапе с учетом новых связей вычисляется форма, являющаяся телом лямбда-выражения, и полученное значение возвращается в качестве значения лямбда-вызыва. Формальным параметрам после окончания вычисления возвращаются те связи, которые у них, возможно, были перед вычислением лямбда-вызыва. Весь этот процесс называют *лямбда-преобразованием* (lambda conversion).



### Объединение лямбда-вызовов



Лямбда-вызовы можно свободно объединять между собой и другими формами. Вложенные лямбда-вызовы можно ставить как на место тела лямбда-выражения, так и на место фактических параметров. Например, в следующем вызове тело лямбда-выражения содержит вложенный лямбда-вызов:

```
_((lambda (y) ; у лямбда-вызова
    ((lambda (x) ; тело вновь
        (list y x)) ; лямбда-вызов
        'ВНУТРЕННИЙ))
     'ВНЕШНИЙ)
  (ВНЕШНИЙ ВНУТРЕННИЙ)
```

В приведенном ниже примере лямбда-вызов является аргументом другого вызова:

```
_((lambda (x) ; лямбда-вызов
    (list 'ВТОРОЙ x)) ; у которого
    ((lambda (y) ; аргументом является
        (list y)) ; новый лямбда-вызов
        'ПЕРВЫЙ))
  (ВТОРОЙ (ПЕРВЫЙ))
```

Обратите внимание, что лямбда-выражение без аргументов (фактических параметров) представляет собой лишь определение, но не форму, которую можно

вычислить. Само по себе оно интерпретатором не воспринимается:

```
(lambda (x y) (cons x (cons y nil)))
Error: Undefined function LAMBDA
```

(Функция LAMBDA не определена.— Ред.)

### **Лямбда-выражение – функция без имени**

Лямбда-выражение является как чисто абстрактным механизмом для определения и описания вычислений, дающим точный формализм для параметризации вычислений при помощи переменных и изображения вычислений, так и механизмом для связывания формальных и фактических параметров на время выполнения вычислений. Лямбда-выражение – это **безымянная функция**, которая пропадает тотчас после вычисления значения формы. Ее трудно использовать снова, так как нельзя вызвать по имени, хотя ранее выражение было доступно как списочный объект. Однако используются и **безымянные функции**, например при передаче функции в качестве аргумента другой функции или при формировании функции в результате вычислений, другими словами, при синтезе программ.

### **DEFUN дает имя описанию функции**

Лямбда-выражение соответствует используемому в других языках определению процедуры или функции, а лямбда-вызов – вызову процедуры или функции. Различие состоит в том, что для программиста лямбда-



выражение – это лишь механизм, и оно не содержит имени или чего-либо подобного, позволяющего сослаться на это выражение из других вызовов. Записывать вызовы функций полностью с помощью лямбда-вызовов не разумно, поскольку очень скоро выражения в вызове пришлось бы повторять, хотя разные вызовы одной функции отличаются лишь в части фактических параметров. Проблема разрешима

путем именования лямбда-выражений и использования в вызове лишь имени.

Дать имя и определить новую функцию можно с помощью функции DEFUN (define function). Ее действие с абстрактной точки зрения аналогично именованию данных (SET и другие). DEFUN вызывается так:

**(DEFUN имя лямбда-список тело)**

Что можно представить себе как сокращение записи

**(DEFUN имя лямбда-выражение)**

из которой для удобства исключены внешние скобки лямбда-выражения и сам атом LAMBDA. Например:

```
(defun list1 (lambda (x y) (cons x (cons y nil))))  
⇒  
(defun list1      (x y) (cons x (cons y nil)))
```

DEFUN соединяет символ с лямбда-выражением, и символ начинает представлять (именовать) определенные этим лямбда-выражением вычисления. Значением этой формы является имя новой функции:

```
_ (defun list1 (x y) ; определение  
    (cons x (cons y nil)))  
LIST  
  (list1 'a 'b) ; вызов  
  (A B)
```

Приведем еще несколько примеров:

```
_ (defun lambdap (выражение) ; проверяет  
    (eq (car выражение) 'lambda))  
LAMBDAP ; на лямбда-выражение  
  (lambdap '(list1 'a 'b))  
NIL
```

```

—(defun проценты (часть чего) ; вычисляет
  (* (/ часть чего) 100)) ; X части
ПРОЦЕНТЫ
(проценты 4 20)
20

```

**SYMBOL-FUNCTION** выдает определение функции  
 Ранее был рассмотрен предикат **BOUNDP**, проверяющий наличие у символа значения. Соответственно предикат **FBOUNDP** проверяет, связано ли с символом определение функции:

```

(fboundp 'list1)

```

Значение символа можно было получить при помощи функции **SYMBOL-VALUE**. Аналогично функция **SYMBOL-FUNCTION** дает определение функции, связанное с символом:

```

(symbol-function 'list1)
(LAMBDA (X Y) (CONS X (CONS Y NIL)))

```

Поскольку определение функции задается списком, а он всегда доступен программе, то можно исследовать работу функций и даже время от времени модифицировать ее, изменяя определения (например, в задачах обучения). В традиционных языках программирования, предлагающих трансляцию, это было бы невозможно. Символ может одновременно называть некоторое значение и функцию, и эти возможности не мешают друг другу. Позиция символа в выражении определяет его интерпретацию:



```

(setq list1 'a)
A
(list1 list1 'b)
(A B)

```

Позднее мы увидим, что символы могут дополнительно к значению и определению обладать в более общем виде и другими свойствами, определенными пользователями, т.е. обладать списком свойств. Определение функции и значение переменных являются лишь двумя различными свойствами переменных, которые программист при необходимости может при помощи списка свойств дополнить или изменить.



В некоторых системах (например, Franz Lisp) определение функции может храниться как значение символа, а не располагаться в специально отведенном месте в памяти. В Коммон Лиспе последняя возможность отсутствует:

```
(setq 'список
      (lambda (x y) (cons x (cons y nil))))
(CALMBA (X Y) (CONS X (CONS Y NIL)))
(список 'а 'б) ; не работает в
                  ; Коммон Лиспе
Error: Undefined function СПИСОК
```

Значение символа в таких системах интерпретируется как определение функции лишь тогда, когда с символом не связано определение функции.



В некоторых системах функцию можно задавать произвольным вычислимым выражением. В таких случаях говорят, что функции – это "полноправные граждане" (first class citizen).

### Задание параметров в лямбда-списке

Рассмотренной ранее DEFUN-формы вполне достаточно для изучения Лиспа. Однако DEFUN-форма содержит, кроме этого, очень богатый механизм *ключевых слов* (lambda-list keyword), с помощью которых аргументы



вызыва функции можно при желании трактовать по-разному<sup>1)</sup>.

С помощью ключевых слов в лямбда-списке можно выделить:

- *необязательные (optional)* аргументы,
- параметр, связываемый с хвостом списка аргументов изменяющейся длины,
- *ключевые (key)* параметры,
- *вспомогательные (auxiliary)* переменные функции.

Аргументу и вспомогательной переменной можно присвоить значение *по умолчанию (default)*.

Ключевые слова начинаются со знака &, и их записывают перед соответствующими параметрами в



лямбда-списке. Действие ключевого слова распространяется до следующего ключевого слова. Приведем список наиболее важных ключевых слов и типов параметров или вспомогательных переменных, обозначаемых ими:

Ключевое слово	Значение
&OPTIONAL	Необязательные параметры
&REST	Переменное количество параметров
&KEY	Ключевые параметры
&AUX	Вспомогательные переменные

Параметры, перечисленные в лямбда-списке до первого ключевого слова, являются обязательными. До настоящего момента мы определяли функции без использования ключевых слов, т.е. фактически использовали лишь обязательные параметры.

Значения объявленных при помощи &OPTIONAL необязательных параметров можно в вызове не указывать. В этом случае они связываются со значением NIL или со значением *выражения по умолчанию (init-*

<sup>1)</sup> Использование ключевых слов является спецификой Коммон Лиспа, отличающей его от других популярных систем, например Интерлиспа. Очевидно, что это сделано с целью добиться большей вычислительной эффективности работы Лисп-системы.— Прим. ред.

form), если таковое имеется. Например, у следующей функции есть обязательный параметр X и необязательный Y со значением по умолчанию (+ X 2):

```
_ (defun fn (x &optional (y (+ x 2)))
  (list x y))
FN
  (fn 2)      ; вычисляется значение по
  (2 4)      ; умолчанию Y=X+2
  (fn 2 5)    ; умолчание не используется
  (2 5)
```

Естественно, ключевые слова можно использовать и в лямбда-выражениях:

```
_ ((lambda (x &optional (y (+ x 2)))
  (list x y)) 2 5)
(2 5)
```

Параметр, указанный после ключевого слова &REST, связывается со списком несвязанных параметров, указанных в вызове. Таким функциям можно передавать переменное количество параметров.

Например:

```
_ (defun fn (x &optional y &rest z)
  (list x y z))
FN
  (fn 'a)
  (A NIL NIL)
  (fn 'a 'b 'c 'd)
  (A B (C D))
```

Обратите внимание, что &REST-параметр связывается со значениями последних аргументов, поскольку отсутствует QUOTE.

Фактические параметры, соответствующие формальным параметрам, обозначенным ключевым словом &KEY, можно задавать в вызове при помощи символьных ключей. Ключом является имя формального параметра, перед которым поставлено двоеточие,

например ":X". Соответствующий фактический параметр будет следовать в вызове функции за ключом и отделяется от него пробелом. Достоинством ключевых параметров является то, что их можно перечислять в вызове, не зная их порядок в определении функций или лямбда-выражений. Например, у следующей функции параметры X, Y и Z являются ключевыми:

```
_ (defun fn (&key x y (z 3))
  (list x y z))
FN
  (fn :y 2 :x 1)
(1 2 3)
```

Параметр Z можно было не задавать, так как ключевые параметры являются необязательными.

При помощи ключевых параметров можно определять функции, которые в зависимости от используемой при вызове комбинации параметров запускаются с различными их значениями.

### **Изображение функций в справочных руководствах**

Механизм ключевых слов лямбда-списка применяется не только для определения функций, в справочниках и учебниках его используют, объясняя употребление

аргументов в описаниях функций. Например, тот факт, что функция LIST имеет неограниченное количество параметров записывается в виде



**(LIST & REST элементы)**

Многим функциям Коммон Лиспа можно передавать необязательные и ключевые параметры. Например, традиционная функция Лиспа

**(MEMBER элемент список)**

проверяющая, входит ли данный элемент в данный список, может в Коммон Лиспе иметь три ключевых

аргумента, модифицирующие действия функции по умолчанию:

**(MEMBER элемент список  
&KEY :TEST :TEST-NOT :KEY)**

В некоторых Лисп-системах при описании используется *m-нотация* (*meta-notation*). Эта запись соответствует префиксной записи в математике, когда имя функции пишется непосредственно перед открывающей скобкой. Например:

*list(x1 x2 ... xn)*

### Функции вычисляют все аргументы

В Коммон Лиспе нет механизма, с помощью которого можно было бы обозначать параметры, не требующие вычисления. Блокирующие вычисление аргумента функции и формы, такие как QUOTE, SETQ и DEFUN, определяются через механизм макросов или макрооператоров, который мы рассмотрим позже.

### Многозначные функции

Ранее мы использовали функции, возвращающие одно значение, или один лисповский объект. Во многих Лисп-системах, в том числе и в Коммон Лиспе, можно определить и *многозначные функции* (*multiple valued functions*), которые возвращают множество значений. Этот механизм более удобен, чем возврат значений через глобальную переменную или через построение списка результатов. Для выдачи и принятия многокомпонентных значений используются специальные формы. Ограничимся в этом случае ссылкой на оригинальное руководство по языку.



### Определение функции в различных диалектах Лиспа

Мы рассмотрели определение функций в Коммон Лиспе. Приведем теперь сравнение принципов опреде-

ления функций и их типов в более старых Лисп-системах и в Коммон Лиспе.

В различных диалектах языка Лисп вместо DEFUN используются следующие имена и формы: DEFINE, DEFINEQ, DE, CSETQ и другие. По типам функции разделяются, например в Интерлиспе на обычновенные (normal function) и специальные. Аргументы обычновенной функции вычисляются слева направо перед вычислением значения самой функции. Число аргументов постоянно и описано в определении функции. Лисповские формы (QUOTE, SETQ и другие) не вычисляют аргументы. В то же время число аргументов функции в различных вызовах может отличаться.

В следующих четырех основных типах функций обобщены способы обработки аргументов и их число:

ТИП ФУНКЦИИ	ВЫЧИСЛЕНИЕ КОЛИЧЕСТВА АРГУМЕНТОВ	КОЛИЧЕСТВО АРГУМЕНТОВ
EXPR	Да	Постоянное
EXPR* или LEXPR	Да	Переменное
FEXPR	Нет	Постоянное
FEXPR*	Нет	Переменное

Знак "\*" – это звездочка Клини, и, как в теории языков, она обозначает повторение. Названия типов и разбиение на них взяты из систем Интерлисп и Маклисп.

С помощью лямбда-выражения определяются "обычновенные" функции типа EXPR. Им соответствуют функции Коммон Лиспа, у которых все параметры обязательны.

Функции, у которых количество вычисляемых аргументов может быть заранее неопределено, в Маклиспе называются LEXPR и в Интерлиспе – EXPR\*. EXPR\*/LEXPR определяется так же, как функция EXPR, только формальные параметры задаются не списком, а одним параметром.

При вычислении значения вызова сначала вычисляются значения аргументов, из них формируется безы-



мянный стек и число фактических параметров становится значением формального параметра. Значение любого фактического параметра можно получить при помощи специальной функции, которая возвращает значение n-го аргумента. Функция LIST действует как функция типа EXPR\*/LEXPR. Число ее аргументов, значения которых предварительно вычисляются, может быть неограниченно.

В Коммон Лиспе функции типа EXPR\*/LEXPR можно определять с помощью параметра типа &REST, который связывается с несвязанным остатком списка значений параметров, имеющего произвольную длину.

### **При вычислении NLAMBDA аргументы не вычисляются**

Функции, которые трактуют свои аргументы так же, как QUOTE и DEFUN, часто называют функциями типа FEXPR. Если вычислять значения аргументов не надо, то такую функцию нужно определять с помощью NLAMBDA-выражения (NO-spread lambda), имеющего ту же структуру, что лямбда-выражение:

### **(NLAMBDA параметры тело-функции)**

Определяемые лямбда-выражением функции типа EXPR(\*) и определяемые NLAMBDA-выражением функции типа FEXPR(\*) могут получать фиксированное или неопределенное число параметров в зависимости от того, определяются ли параметры через список атомов или одним атомом.

Как ранее было сказано, в Коммон Лиспе нельзя определить функцию, не вычисляющую значения аргументов (однако можно использовать так называемые макросы).

### **Упражнения**

1. Определите с помощью лямбда-выражения функцию, вычисляющую  $x+y-x*y$ .



2. Вычислите значения следующих лямбда-вызовов:
- ((lambda (x) (cons x NIL)) 'y)
  - ((lambda (x y) (list y x)) 'x y)
  - ((lambda (x) (list x)) (list NIL))
  - ((lambda (x) (list x)) (list NIL)))
- 
3. Ознакомьтесь с функциями, определяющими функции, в различных Лисп-системах. В какой мере они соответствуют Коммон Лиспу?
4. Определите функции (NULL x), (CADDR x) и (LIST x1 x2 x3) с помощью базовых функций. (Используйте имена NULL1, CADDR1 и LIST1, чтобы не переопределять одноименные встроенные функции системы Коммон Лисп.)
5. Определите функцию (НАЗОВИ x y), которая определяет функцию с именем, заданным аргументом *x*, и лямбда-выражением *y*. Определите с помощью этой функции функцию, вычисляющую сумму квадратов двух чисел, и саму функцию НАЗОВИ.
6. Вычислите значения следующих лямбда-вызовов:
- ((lambda (a &optional (b 2)) (+ a (\* b 3)))  
4 5)
  - ((lambda (a b &key c d) (list a b c d))  
:a 1 :d 8 :c 6)
  - ((lambda  
(a &optional (b 3) &rest x &key c (d a))  
(list a b c d x)))  
1)
  - ((lambda  
(a &optional (b 3) &rest x &key c (d a))  
(list a b c d x)))  
1 6 :c 7)

*Знание – это препятствие  
на пути неизвестности.*

*С. Паронен*

## **2.6 ПЕРЕДАЧА ПАРАМЕТРОВ И ОБЛАСТЬ ИХ ДЕЙСТВИЯ**

- В Лиспе используется передача параметров по значению
- Статические переменные локальны
- Свободные переменные меняют свое значение
- Динамическая и статическая область действия
- Одно имя может обозначать разные переменные
- Упражнения

В языках программирования в основном используются два способа передачи параметров – это *передача параметров по значению* (call by value) и *по ссылке* (call by reference). При передаче параметров по значению формальный параметр связывается с тем же значением, что и значение фактического параметра. Изменения значения формального параметра во время вычисления функции никак не отражаются на значении фактического параметра. С помощью параметров, передаваемых по значению, информацию можно передавать только внутрь процедур, но не обратно из них. При передаче параметров по ссылке изменения значений формальных параметров видны извне и можно возвращать данные из процедуры с помощью присваивания значений формальным параметрам.

**В Лиспе используется передача параметров по значению**

Передача параметров в Лиспе осуществляется в основном по значению. Параметры в Лиспе используются преимущественно лишь для передачи данных в функцию, а результаты возвращаются как значение функции.

ции, а не как значения параметров, передаваемых по ссылке. (В Лиспе все-таки можно с помощью псевдофункций, меняющих структуры, использовать формальные параметры таким же образом, как это происходит при передаче параметров по ссылке. Побочные эффекты их использования отражаются на значениях всех переменных, ссылающихся на некоторый элемент данных. Мы вернемся к этим псевдофункциям чуть позже. Кроме того, механизм возврата значений из многозначных функций также напоминает передачу параметров по ссылке.)

### Статические переменные локальны

Формальные параметры функции в Коммон Лиспе называют *лексическими* или *статическими переменными* (lexical/static variable). Связи статической переменной действительны только в пределах той формы, в которой они определены. Они перестают действовать в функциях, вызываемых во время вычисления, но текстуально описанных вне данной формы. Изменение значений переменных не влияет на одноименные переменные вызовов более внешнего уровня. Статические переменные представляют собой лишь формальные имена других лисповских объектов. После вычисления функции, созданные на это время связи формальных параметров ликвидируются и происходит возврат к тому состоянию, которое было до вызова функции.

Например:

```

_ (defun не-меняет (x) ; X статическая
    (setq x 'новое))
НЕ-МЕНЯЕТ
_ (setq x 'старое)
СТАРОЕ
_ (не-меняет 'новое) ; статическое
НОВОЕ ; значение X изменяется
_ x ; первоначальная связь
СТАРОЕ ; не меняется

```

## **Свободные переменные меняют свое значение**

 Возникшие в результате побочного эффекта изменения значений *свободных переменных* (*free variable*), т.е. используемых в функции, но не входящих в число ее формальных параметров, остаются в силе после окончания выполнения функции. Определим далее функцию ИЗМЕНИТЬ, в которой переменная X свободна. Ее значение будет меняться:

```
(defun изменить ()
  (setq x 'новое))      ; X свободная
ИЗМЕНИТЬ
(изменить)
НОВОЕ
x                      ; значение свободной
НОВОЕ                  ; переменной изменилось
```

## **Динамическая и статическая область действия**

 Под *вычислительным окружением* или *контекстом* (*evaluation environment*) будем понимать совокупность действующих связей переменных с их значениями. Связи формальных параметров вызова со значениями аргументов *действительны* (по умолчанию) только в пределах текста определения функции. Будем говорить, что *область действия* (*scope*) или видимость переменных *статическая*.

В Коммон Лиспе существует однако возможность использования *динамических*, или *специальных* (*dynamic/special variable*), переменных. Это обычно достигается при помощи директивы

**(DEFVAR *переменная*  
&OPTIONAL *начальное значение*)**

Значение переменной, объявленной специальной, определяется динамически *во время вычисления*, а не

в зависимости от контекста *места ее определения*, как для статических переменных. Будем говорить, что *временем действия (extent)* связи динамического формального параметра является все время вычисления вызова, в котором возникла эта связь.

Если переменная свободна и является формальным параметром какого-нибудь охватывающего вызова, то значения статической и динамической переменных вычисляются по-разному. Если переменная является статической (как, например, по умолчанию в Коммон Лиспе), то ее использование в качестве формального параметра в более внешней, предшествующей, форме не влияет на значение свободной переменной. У переменной либо не будет значения, что приведет к ошибке, либо ее значение определяется в соответствии с ее *глобальным* значением, присвоенным на самом внешнем уровне функцией SETQ.

Если переменная при помощи DEFVAR объявлена динамической, то связь, установленная в более внешнем вызове, остается в силе для всех вложенных контекстов, возникающих во время вычисления (при условии, что эта переменная снова не связывается). Например:

```
(setq x 100)      ; глобальное значение X
100
(defun первая (x)  ; статическая X
  (вторая 2))
ПЕРВАЯ
(defun вторая (y)
  (list x y))      ; X свободна
ВТОРАЯ
(первая 1) ; свободная нединамическая
            ; переменная
(100 2)   ; значением X является
            ; глобальное значение X
(defvar x 100) ; начиная с этого
X             ; момента X динамическая переменная
(первая 1) ; X определяется динамически
(1 2)       ; по последней связи
```

Если в функции не используется ни одной свободной переменной, то вычисления в обоих случаях производятся совершенно одинаково. Именно таким функциям надо отдавать предпочтение во время обучения языку Лисп.

### Одно имя может обозначать разные переменные

Интересные различия возникают в использовании статических и глобальных (динамических) переменных, когда один и тот же символ является и фактическим, и формальным параметром:

```
_ (setq x ...) ; глобальная X
...
_ (defun fn (x) ...) ; статическая X
FN
_ (fn 'x) ; статическая X связывается
... ; с глобальной X
```

В случае, подобном приведенному, X в функции FN может быть именем как статической, так и глобальной (или динамической) переменной. В этом случае нужно различать, какая переменная что означает.

В начале главы мы описали функцию НЕ-МЕНЯЕТ, которая изменяла значение статической переменной X, и поэтому глобальное значение X сохранилось. В функции SET-DYN будет меняться динамическое значение X, так как значением первого аргумента вызова SET, т.е. статического X, будет динамическое X из вызова функции. Предположим, что X не объявлена динамической через DEFVAR:

```
(setq x 'старое)
СТАРОЕ
_ (defun set-dyn (x)
  (set x 'новое)) ; меняет динамическую X
SET-DYN
_ (set-dyn 'x)
НОВОЕ
```

## Х НОВОЕ



*В зависимости от использования символ обозначает ртуть, Меркурий, среду (третий день недели) или дву-домное растение.*

переменные не могут выступать как лисповские объекты сами по себе или в составе более сложных структур, поскольку они используются лишь как формальные имена, при помощи которых записываются вычисления. Следующая функция SET-DYN работает точно так же, как и предыдущая, так как значение первого аргумента вызова функции SET ссылается не на статическую переменную X, а на динамическую:

```
_ (defun set-dyn (x)
  (set (car '(x y)) 'новое))
```

Функция QUOTE также возвращает в качестве значения лишь динамическую переменную, что видно из следующего примера:

```
(defun проба-eval (x) (eval 'x))
ПРОБА-EVAL
_(setq x 'старое) ; динамическая связь X

СТАРОЕ
_(проба-eval 'новое) ; аргументом EVAL
СТАРОЕ           ; является не статическая,
                   ; а динамическая X
```

В некоторых Лисп-системах переменные по умолчанию являются не статическими, а динамическими. В таких



системах в приведенных выше примерах результаты будут другие. Если, например, переменная `X` была бы определена как динамическая специальная переменная, то функция `SET-DYN` не присвоила бы ей нового глобального значения. Значение можно было бы изменить лишь в динамическом контексте вызова `SET-DYN`, уничтожаемом после завершения вызова.

Современная тенденция развития языка ведет к статическим Лисп-системам. Фактически разница между динамическими и статическими переменными позволяет использовать во вложенных функциях динамическое значение переменной внешнего уровня, несмотря на то что встречались статические переменные с тем же именем. К ошибкам, возникающим из подобных конфликтов имен, мы вернемся подробнее в связи с функциональными вычислениями, работой в вычислительном контексте момента определения и в связи с макросами.

Статические вычисления позволяют сделать более хорошие трансляторы. С другой стороны, динамический интерпретатор Лиспа реализовать легче, чем статический. В некоторых системах интерпретируемые функции вычисляются динамически, а оттранслированные функции вычисляются по статическим правилам. В результате одна и та же программа может работать по-разному в режиме интерпретации и в оттранслированном виде!

### Упражнения

1. В чём различия следующих типов переменных:
  - a) статические переменные
  - b) динамические переменные
  - c) специальные переменные
  - d) глобальные переменные
  - e) свободные переменные
2. Чем отличается статическое вычисление от динамического, если в функции нет свободных переменных?



3. Определим функции F и G следующим образом:

$\overline{F}$   
`(defun f (y) (g y))`

$\overline{G}$   
`(defun g (x) (list x y))`

Какое значение или сообщение об ошибке будет результатом следующих вызовов функций F и G:

- a) `(f 'y)`
- b) `(f (setq y 'y))`
- c) `(g 'y)`
- d) `(g (setq y 'y))`
- e) последовательное вычисление

`(setq y 'x)`  
`(g y)`  
`(f y)`



*Нужно быть терпеливым,  
чтобы научиться терпению.*

*E. Лец*

## 2.7 ВЫЧИСЛЕНИЕ В ЛИСПЕ

- Программа состоит из форм и функций
- Управляющие структуры Лиспа являются формами
- LET создает локальную связь
- Последовательные вычисления: PROG1, PROG2 и PROGN
- Разветвление вычислений: условное предложение COND
- Другие условные предложения: IF, WHEN, UNLESS и CASE
- Циклические вычисления: предложение DO
- Предложения PROG, GO и RETURN
- Другие циклические структуры
- Повторение через итерацию или рекурсию
- Формы динамического прекращения вычислений: CATCH и THROW
- Упражнения

### Программа состоит из форм и функций

Под *формой* (form) понимается такое символьное выражение, значение которого может быть найдено интерпретатором. Ранее мы уже использовали наиболее простые формы языка: константы, переменные, лямбда-вызовы, вызовы функций и их сочетания. Кроме них были рассмотрены некоторые специальные формы, такие как QUOTE и SETQ, трактующие свои аргументы иначе, чем обычные функции. Лямбда-выражение без фактических параметров не является формой.

Вычислимые выражения можно разделить на три группы:

1. *Самоопределенные (self-evaluating) формы.* Эти формы, подобно константам, являются лисповски-

ми объектами, представляющими лишь самих себя. Это такие формы, как числа и специальные константы Т и NIL, а также знаки, строки и битовые векторы, рассматриваемые далее в главе, посвященной типам данных. Ключи, начинающиеся с двоеточия и определяемые через ключевое слово &KEY в лямбда-списке, также являются самоопределенными формами.

2. Символы, которые используются в качестве переменных.
3. Формы в виде списочной структуры, которыми являются:
  - a) Вызовы функций и лямбда-вызовы.
  - b) *Специальные формы* (special form), в число которых входят SETQ, QUOTE и многие описанные в этой главе формы, предназначенные для управления вычислением и контекстом.
  - c) *Макровызовы* (будут рассмотрены позже).

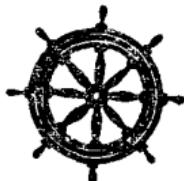
У каждой формы свой синтаксис и семантика, основанные, однако, на едином способе записи и интерпретации.

### Управляющие структуры Лиспа являются формами

В распространенных процедурных языках наряду с основными действиями есть специальные управляющие механизмы разветвления вычислений и организации циклов. В Паскале, например, используются структуры IF THEN ELSE, WHILE DO, CASE и другие.

Управляющие структуры Лиспа (мы будем для них использовать термин *предложение* (clause)) выглядят

внешне как вызовы функций. Предложения будут записываться в виде скобочных выражений, первый элемент которых действует как имя управляющей структуры, а остальные элементы – как "аргументы". Результатом вычисления, так же как у функций, является значение, т.е. управляющие структуры представляют собой формы. Однако предложения не являются вызовами функций, и разные предложения используют аргументы по-разному.



Наиболее важные с точки зрения программирования синтаксические формы можно на основе их использования разделить на следующие группы:

#### **Работа с контекстом:**

- QUOTE или блокировка вычисления;
- вызов функции и лямбда-вызов;
- предложения LET и LET\*.

#### **Последовательное выполнение:**

- предложения PROG1, PROG2 и PROGN.

#### **Разветвление вычислений:**

- условные предложения COND, IF, WHEN, UNLESS;
- выбирающее предложение CASE.

#### **Итерации:**

- циклические предложения DO, DO\*, LOOP, DOTIMES, DOUNTIL.

#### **Передачи управления:**

- предложения PROG, GO и RETURN.

#### **Динамическое управление вычислением:**

- THROW и CATCH, а также BLOCK.

Ранее мы уже рассматривали форму QUOTE, а также лямбда-вызов и вызов функции. Эти формы тесно связаны с механизмом определения функций и их вызова. Остальные формы в основном используются в теле лямбда-выражений, определяющих функции.

#### **LET создает локальную связь**

Вычисление вызова функции создает на время вычисления новые связи для формальных параметров функции. Новые связи внутри формы можно создать и с помощью предложения LET. Эта структура (немного упрощенно) выглядит так:

**(LET ((*m1* знач1) (*m2* знач2) ...)**  
**форма1**  
**форма2 ...)**

Предложение LET вычисляется так, что сначала статические переменные *m1*, *m2*, ... из первого "аргумента" формы связываются (одновременно) с соответствующими значениями знач1, знач2, .... Затем слева направо вычисляются значения форм *форма1*, *форма2*, .... В качестве значения всей формы возвращается значение последней формы. Как и у функций, после окончания вычисления связи статических переменных *m1*, *m2*, ... ликвидируются и любые изменения их значений (SETQ) не будут видны извне. Например:

```
(setq x 2)
2
(let ((x 0))
  (setq x 1))
1
x
2
```

Форма LET является на самом деле синтаксическим видоизменением лямбда-вызыва, в которой формальные и фактические параметры помещены совместно в начале формы:

**(LET ((*m1 a1*) (*m2 a2*) ... (*mn an*))**  
**форма1 форма2 ...)**

↔  
**((LAMBDA**  
*(m1 m2 ... mn)* ; формальные параметры  
*форма1 форма2 ...)*; тело функции  
*a1 a2 ... an)* ; фактические параметры

Тело лямбда-выражения в Коммон Лиспе может состоять из нескольких форм, которые вычисляются

последовательно, и значение последней формы возвращается в качестве значения лямбда-вызыва.

Значения переменным формы LET присваиваются одновременно. Это означает, что значения всех переменных *ti* вычисляются до того, как осуществляется связывание с формальными параметрами. Новые связи этих переменных еще не действуют в момент вычисления начальных значений переменных, которые перечислены в форме позднее. Например:

```
_ (let ((x 2) (y (* 3 x)))
  (list x y)) ; при вычислении Y
Error: Unbound atom X ; у X нет связи
```

Побочный эффект можно наблюдать при работе с формой LET\* подобной LET, но вычисляющей значения переменных *последовательно*:

```
_ (let* ((x 2) (y (* 3 x)))
  (list x y))
(2 6)
```

### Последовательные вычисления: PROG1, PROG2 и PROGN

Предложения PROG1, PROG2 и PROGN позволяют работать с несколькими вычисляемыми формами:

(PROG1 *форма1* *форма2* ... *формаN*)  
 (PROG2 *форма1* *форма2* ... *формаN*)  
 (PROGN *форма1* *форма2* ... *формаN*)

У этих специальных форм *последовательное* число аргументов, которые они последовательно вычисляют и возвращают в качестве значения значение первого (PROG1), второго (PROG2) или последнего (PROGN) аргумента. Эти формы не содержат механизма определения внутренних переменных:

```
(progn (setq x 2) (setq y (* 3 x)))
 $\frac{6}{2}x$ 
```

Многие формы, как, например описанная выше форма LET(\*), позволяют использовать последовательность форм, вычисляемых последовательно, и в качестве результата последовательности возвращают значение последней формы. Это свойство называют *неявным PROGN* (*implicit progn feature*).

### Разветвление вычислений: условное предложение COND

Предложение COND является основным средством разветвления вычислений. Это синтаксическая форма, позволяющая управлять вычислениями на основе определяемых предикатами условий. Структура условного предложения такова:



(COND (*p<sub>1</sub>* *a<sub>1</sub>*)  
           (*p<sub>2</sub>* *a<sub>2</sub>*)  
           ...  
           (*p<sub>N</sub>* *a<sub>N</sub>*))

Предикатами *p<sub>i</sub>* и результирующими выражениями *a<sub>i</sub>* могут быть произвольные формы. Значение предложения COND определяется следующим образом:

1. Выражения *p<sub>i</sub>*, выполняющие роль предикатов, вычисляются последовательно слева направо (сверху вниз) до тех пор, пока не встретится выражение, значением которого не является NIL, т.е. логическим значением которого является истина.
2. Вычисляется результирующее выражение, соответствующее этому предикату, и полученное значение возвращается в качестве значения всего предложения COND.
3. Если истинного предиката нет, то значением COND будет NIL.

Рекомендуется в качестве последнего предиката использовать символ Т, и соответствующее ему результирующее выражение будет вычисляться всегда в тех случаях, когда ни одно другое условие не выполняется.

В следующем примере с помощью предложения COND определена функция, устанавливающая тип выражения<sup>1)</sup>:

```
- (defun тип (l)
  (cond ((null l) 'пусто)
        ((atom l) 'атом)
        (t 'список)))
ТИП
(тип '(а б с))
СПИСОК
(тип (atom '(а т о м)))
ПУСТО
```

В условном предложении может отсутствовать результирующее выражение *ai* или на его месте часто может быть последовательность форм:

(COND (p1 a1))	
...	
(pi)	; результирующее
...	; выражение отсутствует
(pk ak1 ak2 ... akN)	; последовательность форм
...)	; в качестве результата



Если условию не ставится в соответствие результирующее выражение, то в качестве результата предложения COND при истинности предиката выдается само значение предиката. Если же условию соответствует несколько форм, то при его истинности формы вычисляются последовательно слева направо и ре-

<sup>1)</sup> Здесь для примера принимается, что возможны лишь три типа выражений: пустой список, атом и список.— Прим. ред.

зультатом предложения COND будет значение последней формы последовательности (неявный PROGN).

В качестве примера использования условного предложения определим логические действия логики высказываний "и", "или", "не", => (импликация) и <=> (тождество):

```

_(defun И (x y)
  (cond (x y)
        (t nil)))
И
_(defun ИЛИ (x y)
  (cond (x t)
        (t y)))
ИЛИ
_(defun НЕ (x)
  (not x))
НЕ
_(defun => (x y)
  (cond (x y)
        (t t)))
=>
_(defun <=> (x y)
  (и (=> x y) (=> y x)))
<=>

```

Импликацию можно определить и через другие операции:

```

_(defun => (x y)
  (или x (не y)))
=>
_(defun <=> (x y)
  (и (=> x y) (=> y x)))
<=>

```

Предикаты "и" и "или" входят в состав встроенных функций Лиспа и называются AND и OR. Число их аргументов может быть произвольным.

(and (atom nil) (null nil) (eq nil nil))  
T

Предикат AND в случае истинности возвращает в качестве значения значение своего последнего аргумента. Его иногда используют как упрощение условного предложения по следующему образцу:

(AND *условие1* *условие2* ... *условиеN*)  
 $\Leftrightarrow$   
 (COND ((AND *условие1* *условие2* ... *условиеN-1*)  
*условиеN*)  
 (T NIL))

(and (atom nil) (+ 2 3))  
5

Такое использование предиката AND не рекомендуется.

Предложения COND можно комбинировать таким же образом, как и вызовы функций. Например, предикат "исключающее или" (exclusive or или xor), который ложен, когда оба аргумента одновременно либо истинны, либо нет, можно определить следующим образом:

(defun xor (x y)  
 (cond (x (cond (y nil)  
 (t t)))  
 (t y)))  
**XOR**  
 (xor t nil)  
 T  
 (xor nil nil)  
NIL

В этой функции на месте результирующего выражения первого условия вновь стоит предложение COND. На

месте, отведенном условию, также можно использовать еще одно условное предложение, и в этом случае мы получим условное условие. Такие построения очень быстро приводят к труднопонимаемым определениям.

### Другие условные предложения: IF, WHEN, UNLESS и CASE

Предложение COND представляет собой наиболее общую условную структуру. Ее критикуют за обилие скобок и за то, что структура этого предложения

совершенно оторвана от естественного языка. Поэтому в Лисп-системах используются и другие, в различных отношениях более естественные, условные предложения. Но можно обойтись и с помощью лишь COND предложения.

В простом случае можно воспользоваться вплоть до естественной и содержащей мало скобок формой IF.

(IF условие то-форма иначе-форма)

↔

(COND (условие то-форма)  
(Т иначе-форма))

(if (atom t) 'атом 'список)  
ATOM

Можно еще использовать формы WHEN и UNLESS:

(WHEN условие форма1 форма2 ...)

↔

(UNLESS (NOT условие) форма1 форма2 ...)

↔

(COND (условие форма1 форма2 ...))

↔

(IF условие (PROGN форма1 форма2 ...) NIL)

Также можно применять подобное используемому в языке Паскаль выбирающее предложение CASE:

**(CASE ключ**

**(список-ключей1 m11 m12 ...)**

**(список-ключей2 m21 m22 ...)**

**...)**

Сначала в форме CASE вычисляется значение ключевой формы *ключ*. Затем его сравнивают с элементами списков ключей *список-ключей<sub>i</sub>*, с которого начинаются альтернативы. Когда в списке найдено значение ключевой формы, начинают вычисляться соответствующие формы *m<sub>i1</sub>*, *m<sub>i2</sub>*, ..., значение последней из которых и возвращается в качестве значения всего предложения CASE (неявный PROGN).

### Циклические вычисления: предложение DO

В случае повторяющихся вычислений в Лиспе используются вызывающие сами себя (рекурсивные) функции (и условные предложения) либо известные в основном по процедурным языкам циклы, передачи управления и другие подобные механизмы. Прежде всего познакомимся сначала с предлагаемыми Лиспом возможностями по использованию циклов.

Самым общим циклическим или итерационным предложением в Коммон Лиспе является DO. С его помощью можно задать

1. Набор внутренних статических переменных с их начальными значениями, как в предложении LET(\*) .
2. Ряд форм, вычисляемых последовательно в цикле.
3. Изменения внутренних переменных после каждой итерации (например, наращивание счетчиков и т. п.).
4. Условие окончания цикла и выражение, значение которого будет значением всего предложения.

Предложение DO имеет следующую форму:

**(DO ((var1 знач1 шаг1) (var2 знач2 шаг2) ...)**

**(условие-окончания форма11 форма12 ...)**



*форма21  
форма22  
...)*

Первый аргумент предложения описывает внутренние переменные *var1*, *var2*, ..., их начальные значения *знач1*, *знач2*, ..., а также формы обновления *шаг1*, *шаг2*, .... Вычисление предложения DO начинается с присваивания значений переменным формы таким же образом, как в случае предложения LET. Переменным, начальное значение которых не задано, присваивается по умолчанию NIL. В каждом цикле после присваивания значения переменным вычисляется условие окончания. Как только значение условия не равно NIL, т.е. условие окончания истинно, последовательно вычисляются формы *форма1i*, и значение последней формы возвращается как значение всего предложения DO. В противном случае последовательно вычисляются формы *форма2i* из тела предложения DO. На следующем цикле переменным *vari* присваиваются (одновременно) значения форм *шагi*, вычисляемых в текущем контексте, проверяется условие окончания и т.д. Если для переменной не задана форма, по которой она обновляется, то значение переменной не меняется.

Для примера с помощью предложения DO определим функцию, вычисляющую п-ю степень числа (n – целое, положительное):

```

_ (defun expt1 (x n)
  (do ((результат 1))      ; начальное
       ; значение
       ((= n 0) результат) ; условие
                           ; окончания
       (setq результат (* результат x))
       (setq n (- n 1)))
EXPT1
(expt1 2 3)
8

```

Мы использовали в качестве имени функции символ EXPT1, чтобы не переопределять лисповскую функцию возвведения в степень EXPT.

Идея определения состоит в том, чтобы умножить X на себя N раз, что и является N-й степенью числа X. В каждом цикле значение переменной РЕЗУЛЬТАТ умножается на X до тех пор, пока значение счетчика N не уменьшится до 0 и конечное значение переменной РЕЗУЛЬТАТ можно будет выдать в качестве значения предложения DO.

Так как в предложении DO можно совместно с переменными описать и закон их изменения, то функцию EXPT можно было бы задать и такой формой:

```
(defun expt2 (x n)
  (do ((результат x (* результат x))
        (разы n (- разы 1)))
    ((= разы 1) результат)) ; условие
                                ; окончания
```

В этом определении нет вычисляемого в цикле тела предложения DO, присутствуют только описания переменных, законов их изменения и условие завершения. Обратите внимание, что условие окончания функции EXPT1 отличается от условия окончания EXPT2. (Как вы думаете, почему?)

Аналогично тому, как предложению LET соответствовало последовательно вычисляющее свои переменные предложение LET\*, так и у предложения DO есть свой вариант DO\*.

### Предложения PROG, GO и RETURN

На Лиспе можно писать программы и в обычном операторном стиле с использованием передачи управления, как, например, в Фортране. Для этой цели уже в первых Лисп-системах существовало предложение PROG или *PROG-механизм* (prog feature). Значимость PROG-механизма в программировании уменьшилась в связи с введением в современных Лисп-системах более развитых условных и циклических форм, таких как форма DO, так что использование PROG-механизма,

например в Коммон Лиспе, в общем-то не рекомендуется. Можно показать, что все выразимое предложением PROG можно записать и с помощью предложения DO и, как правило, в более понятной форме.

С помощью предложения PROG можно:

1. Осуществлять последовательное вычисление форм.
2. Организовывать циклы с помощью команды перехода.
3. Использовать локальные переменные формы.

Структура предложения PROG такая же, как и в более старых системах:

(PROG (*m<sub>1</sub>* *m<sub>2</sub>* ... *m<sub>N</sub>*)  
*оператор<sub>1</sub>*  
*оператор<sub>2</sub>*  
...  
*оператор<sub>M</sub>*)

Перечисленные в начале формы *m<sub>i</sub>* являются локальными статическими переменными формы,



которые можно использовать для хранения промежуточных результатов так же, как это делается при программировании на операторных языках. Если какая-нибудь форма *оператор<sub>i</sub>* является символом или целым числом, то это *метка перехода* (tag). На такую метку можно передать управление оператором GO:

(GO *метка*)

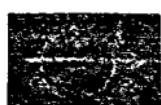
GO не вычисляет значение своего "аргумента".

Кроме этого, в PROG-механизм входит оператор окончания вычисления и возврата значения:

(RETURN *результат*)

Операторы предложения PROG вычисляются слева направо (сверху вниз), пропуская метки перехода. Оператор RETURN прекращает выполнение предложе-

ния PROG; в качестве значения всего предложения возвращается значение аргумента оператора PROG. Если во время вычисления оператор RETURN не встретился, то значением PROG после вычисления его последнего оператора станет NIL. (Когда PROG-механизм используется для получения побочного эффекта, то возвращаемое значение не играет никакой роли.)



*"Следуйте дальше" –  
знак, используемый  
следопытами.*

Через список переменных можно определить локальные для предложения PROG *программные переменные* (program variable). Перед вычислениями им присваиваются значения NIL. Если переменных нет, то на месте списка переменных

нужно оставить NIL. После вычисления значения формы связи программных переменных исчезают также, как и значения переменных форм LET(\*) и DO(\*) или как связи формальных параметров лямбда-выражения в вызове функции.

В следующем примере предложение PROG используется для определенной нами ранее через DO функции возвведения в степень EXPT:

```

(defun expt3 (x n)
  (PROG (результат)
        (setq результат x)
        loop ; метка
        (if (= n 1)
            (RETURN результат) ; выход
            (setq результат (* результат x))
            (setq n (- n 1))
            (GO loop))) ; передача управления
EXPT3
(expt3 2 3)
8
результат
Error: Unbound atom РЕЗУЛЬТАТ

```

Это определение явно более громоздко, чем описанные выше версии, основанные на DO.

Механизм передачи управления и предложение RETURN можно использовать наряду с PROG и в некоторых других формах, как, например, DO(\*) .

Формы GO и RETURN являются примерами статических форм, т.е. они управляют вычислением только в пределах текста определения.

### Другие циклические структуры

В Коммон Лиспе есть еще циклические предложения. Форма

`(LOOP m1 m2 ...)`

реализует цикл, в котором формы *m1, m2, ...* вычисляются снова и снова. Цикл завершается лишь в случае, если в какой-нибудь из вычисляемых форм не встретится явный оператор завершения RETURN (или другая форма, прекращающая вычисления).



Часто некоторый цикл надо выполнить определенное количество раз или выполнить его с каждым элементом списка. В Коммон Лиспе для этого имеются формы DOTIMES и DOLIST.

Повторяющиеся вычисления используются и в МАР-функциях, и в других функциях с функцией в качестве аргумента, или в функционалах. В дальнейшем мы рассмотрим функционалы более подробно. В Лиспе, как мы увидим далее, довольно легко и самим описывать новые структуры управления.

### Повторение через итерацию или рекурсию

В "чистом" функциональном Лиспе нет ни циклических предложений (DO, PROG и другие), ни тем более операторов передачи управления. Для программирования повторяющихся вычислений в нем используются лишь условные предложения и определения *рекурсивных*, или вызывающих самих себя, функций.

Например, рекурсивный вариант функции EXPT можно было бы определить так:

```
(defun expt4 (x n)
  (if (= n 1) x
      (* x (expt4 x (- n 1)))))
EXPT4
```

Функция EXPT4 вызывает себя до тех пор, пока используемый как счетчик аргумент N не уменьшится до единицы, т.е. всего N-1 раз. После этого в качестве результата вызова функции EXPT4 возвращается значение X. При каждой передаче значения на предыдущий уровень результат умножается на X. Так X окажется перемноженным на себя N раз.

 Рекурсивное определение функции EXPT короче и в большей степени отражает суть функции, чем версии, основанные на DO и PROG.

Рассмотрим еще одну функцию, просто определяемую через рекурсию, – факториал (факториал – это произведение данного числа на все целые положительные числа, меньшие данного. Например, факториал от 5, обозначаемый как 5!, есть  $1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 = 120$ . Факториалом нуля считается 1):

```
(defun ! (n)
  (if (= n 0) 1
      (* n (! (- n 1)))))
!
(! 5)
120
```

Итеративные и рекурсивные программы теоретически одинаковы по своим вычислительным возможностям, иными словами, множества вычислимых с их помощью функций совпадают (так называемые частично рекурсивные функции). Так что любое вычисление в принципе, можно запрограммировать любым из этих способов. Однако свойства итеративных и рекурсивных вариантов программ могут существенно отличаться. В связи с

этим часто приходится решать, какой из способов программирования больше подходит для данной задачи. От сделанного выбора зависит простота и естественность программирования, а также его эффективность с точки зрения времени исполнения и использования памяти.

С помощью итеративного программирования, как правило, более длинного и трудного в осуществлении, результат может вычисляться значительно проще и быстрее. Это происходит по двум причинам, во-первых, потому, что вычислительные машины в общем ориентированы на последовательные вычисления, и, во-вторых, потому, что трансляторы не всегда в состоянии преобразовать рекурсивное определение в итеративное и используют при вычислениях стек, несмотря на то что он не всегда нужен.

Рекурсивное программирование в общем более короткое и содержательное. Особенно полезно использовать рекурсию в тех случаях, когда решаемая задача или обрабатываемые данные по сути своей рекурсивны. Например, математическое определение факториала рекурсивно и его реализация через рекурсивную функцию совершенно естественна:

$$\begin{aligned} n! &= 1 && \text{, если } n=0 \\ n! &= n*(n-1)! && \text{, если } n>0 \end{aligned}$$

Рекурсивное решение хорошо подходит для работы со списками, так как списки могут рекурсивно содержать подсписки. Рекурсивными функциями можно с успехом пользоваться при работе с другими динамическими структурами, которые заранее не полностью известны. Рекурсивные процедуры занимают важное место почти во всех программах, связанных с искусственным интеллектом.

В главе, посвященной технике функционального программирования, мы попробуем показать, что программирование с помощью рекурсии и условного предложения вполне осуществимо и разумно. Мы увидим, что рекурсия позволяет взглянуть на программирование с другой точки зрения, которая значительно

отличается от обычной основанной на операторах и отраженной, например, в предложении PROG.

### **Формы динамического прекращения вычислений: CATCH и THROW**

До сих пор мы рассматривали лишь структуры, вычисление которых производится в одном *статическом* контексте. В некоторых случаях возникает необходимость прекратить вычисление функции *динамически* из другого вычислительного контекста, где вычисляются некоторые подвыражения, и выдать результат в более раннее состояние, не осуществляя нормальную



последовательность возвратов из всех вложенных вызовов (*dynamic pop-local exit*). Это нужно, например, тогда, когда какая-нибудь вложенная программа обнаружит ошибку, по которой можно решить, что дальнейшая работа бесполезна либо может даже навредить. Возврат же управления по обычным правилам привел бы к продолжению вычислений на внешних уровнях.

Такое динамическое прерывание вычислений можно запрограммировать в Коммон Лиспе с помощью форм CATCH и THROW, которые, как это видно из их имен (ПОИМАТЬ, БРОСИТЬ), передают управление. Подготовка к прерыванию осуществляется специальной формой CATCH:

**(CATCH метка форма1 форма2 ...)**

Например:

**(CATCH 'возврат1  
(делай-раз) (делай-два) (делай-три))**

При вычислении формы сначала вычисляется метка, а затем формы *форма1*, *форма2*, ... слева направо. Значением формы будет последнее значение (неявный &PROG) при условии, что во время вычисления непосредственно этих форм или форм вызванных из них не встретится предложение THROW:

**(THROW метка значение)**

Если аргумент *метка* вызова THROW представляет собой тот же лисповский объект, что и метка в форме CATCH, то управление передается обратно в форму CATCH и его значением станет значение второго аргумента формы THROW. В приведенном ранее примере в результате вычисления формы

**(THROW возврат1 'сделано)**

вызов CATCH получает значение СДЕЛАНО, и если это произошло во время вычисления функции ДЕЛАЙ-ДВА, то вычисление ДЕЛАЙ-ТРИ отменяется. Механизм CATCH-THROW позволяет осуществлять возврат управления из динамического окружения, вложенного на любую глубину.

**Упражнения**

1. Запишите следующие лямбда-вызовы с использованием формы LET и вычислите их значения на машине:
  - ((lambda (x y) (list x y))  
'(+ 2 3) 'c)
  - ((lambda (x y) ((lambda (z) (list x y z)) 'c)  
'a 'b)
  - ((lambda (x y) (list x y))  
((lambda (z) z) 'a)  
'b)
2. С помощью предложений COND или CASE определите функцию, которая возвращает в качестве значения столицу заданного аргументом государства:

**(столица 'Финляндия)  
ХЕЛЬСИНКИ**

3. Предикат сравнения (> x y) истинен, если x больше, чем y. Опишите с помощью предиката > и условного



предложения функцию, которая возвращает из трех числовых аргументов значение среднего по величине числа:

$$\frac{(4 \ 7 \ 6)}{6}$$

4. Можно ли с помощью предложения COND запрограммировать предложение IF как функцию?
5. Запрограммируйте с помощью предложения DO итеративную версию функции факториал.
6. Определите функцию (ПРОИЗВЕДЕНИЕ n m), вычисляющую произведение двух целых положительных чисел.
7. Функция (LENGTH x) является встроенной функцией, которая возвращает в качестве значения длину списка (количество элементов на верхнем уровне). Определите функцию LENGTH сначала рекурсивно с помощью предложения COND и затем итеративно с помощью предложения PROG.
8. В математике числа Фибоначчи образуют ряд 0, 1, 1, 2, 3, 5, 8, .... Эту последовательность можно определить с помощью следующей функции FIB:

$$\begin{aligned} fib(n) &= 0 && , \text{ если } n=0 \\ fib(n) &= 1 && , \text{ если } n=1 \\ fib(n) &= fib(n-1)*fib(n-2) && , \text{ если } n>1 \end{aligned}$$

Определите лисповскую функцию fib(n), вычисляющую n-й элемент ряда Фибоначчи.

9. Определите функцию ДОБАВЬ, прибавляющую к элементам списка данное число:

$$\frac{(2 \ 7 \ 3) \ 3}{(5 \ 10 \ 6)}$$

*Неужели мы существуем лишь  
в воображении других?*

*E. Лец*

## 2.8 ВНУТРЕННЕЕ ПРЕДСТАВЛЕНИЕ СПИСКОВ

- Лисповская память состоит из списочных ячеек
- Значение представляется указателем
- CAR и CDR выбирают поле указателя
- CONS создает ячейку и возвращает на нее указатель
- У списков могут быть общие части
- Логическое и физическое равенство не одно и то же
- Точечная пара соответствует списочной ячейке
- Варианты точечной и списочной записей
- Управление памятью и сборка мусора
- Вычисления, изменяющие и не изменяющие структуру
- RPLACA и RPLACD изменяют содержимое полей
- Изменение структуры может ускорить вычисления
- Упражнения



В этой главе рассматриваются представление списков и атомов в памяти машины, а также специальные функции, с помощью которых можно изменять внутреннюю структуру списков. Без знания внутренней структуры списков и принципов ее использования изучение работы со списками и функциями останется неполным.

В чистом функциональном программировании специальные функции, изменяющие структуры, не используются. В функциональном программировании лишь создаются новые структуры путем анализа, расчленения и копирования ранее созданных структур. При этом созданные структуры никогда не изменяются и не уничтожаются структуры, значения которых уже не нужны. В практи-

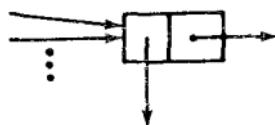
ческом программировании псевдофункции, изменяющие структуры, иногда все-таки нужны, хотя их использования в основном пытаются избежать.

### **Лисповская память состоит из списочных ячеек**

Оперативная память машины, на которой работает Лисп-система, логически разбивается на маленькие области, которые называются *списочными ячейками* (memory cell, list cell, cons cell или просто cons).

Списочная ячейка состоит из двух частей, *полей CAR* и *CDR*. Каждое из полей содержит *указатель* (pointer). Указатель может ссылаться на другую списочную ячейку или на некоторый другой лисповский объект,

как, например, атом. Указатели между ячейками образуют как бы цепочку, по которой можно из предыдущей ячейки попасть в следующую и так, наконец, до атомарных объектов. Каждый известный системе атом записан в определенном месте памяти лишь один раз. (В действительности в Коммон Лиспе можно использовать много пространств имен, в которых атомы с одинаковыми именами хранятся в разных местах и имеют различную интерпретацию.)



*Рис. 2.8.1.*

Графически списочная ячейка представляется прямоугольником (рис. 2.8.1), разделенным на части (поля) CAR и CDR. Указатель изображается в виде стрелки, начинающейся в одной из частей прямоугольника и заканчивающейся на изображении другой ячейки или атоме, на которые ссылается указатель.

### **Значение представляется указателем**

Указателем списка является указатель на первую ячейку списка. На ячейку могут указывать не только



поля CAR и CDR других ячеек, но и используемый в качестве переменной символ, указатель из которого ссылается на объект, являющийся значением символа. Указатель на значение хранится вместе с символом в качестве его системного свойства. Кроме значения системными свойствами символа могут быть определение функции, представленное в виде лямбда-выражения, последовательность знаков, задающая внешний вид переменной (print name), выражение, определяющее пространство, в которое входит имя, и список свойств (property list). Эти системные свойства, которые мы рассмотрим подробнее в следующей главе, не будут отражены на наших рисунках.

Побочным эффектом функции присваивания SETQ является замещение указателя в поле значения символа. Например, следующий вызов:

```
(setq список '(a b c))
(A B C)
```

создает в качестве побочного эффекта изображенную на рис. 2.8.2 штриховую стрелку.

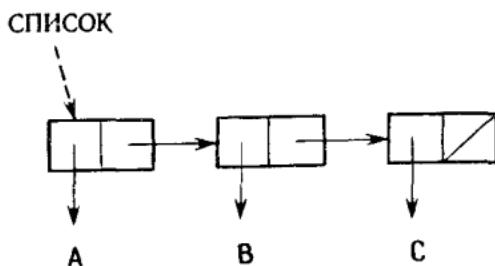


Рис. 2.8.2.

Изображение одноуровневого списка состоит из последовательности ячеек, связанных друг с другом через указатели в правой части ячеек. Правое поле последней ячейки списка в качестве признака конца списка ссылается на пустой список, т.е. на атом NIL. Графически ссылку на пустой список часто изображают в виде перечеркнутого поля. Указатели из полей CAR

ячеек списка ссылаются на структуры, являющиеся элементами списка, в данном случае на атомы A, B и C.

### CAR и CDR выбирают поле указателя



Работа селекторов CAR и CDR в графическом представлении становится совершенно очевидной. Если мы применим функцию CAR к списку СПИСОК, то результатом будет содержимое левого поля первой списочной ячейки, т.е. символ A:

```
(car список)
A
```

Соответственно вызов

```
(cdr список)
(B C)
```

возвращает значение из правого поля первой списочной ячейки, т.е. список (B C).

### CONS создает ячейку и возвращает на нее указатель

Допустим, что у нас есть два списка:

```
(setq голова '(b c))
(B C)
(setq хвост '(a b c))
(A B C)
```

Вызов функции

```
(cons голова хвост)
((B C) A B C)
```

строит новый список из ранее построенных списков ГОЛОВА и ХВОСТ так, как это показано на рис. 2.8.3.

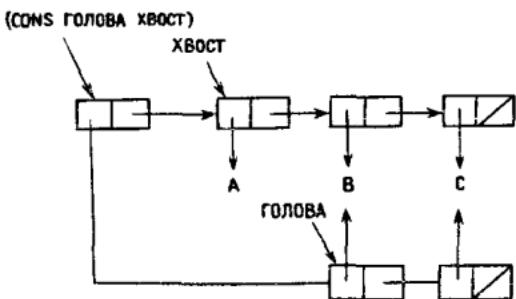


Рис. 2.8.3.

**CONS** создает новую списочную ячейку (и соответствующий ей список). Содержимым левого поля новой ячейки станет значение первого аргумента вызова, а правого – значение второго аргумента. Обратите внимание, что одна новая списочная ячейка может связать две большие структуры в одну новую структуру. Это довольно эффективное решение с точки зрения создания структур и их представления в памяти. Заметим, что применение функции **CONS** не изменило структуры списков, являющихся аргументами, и не изменило значений переменных **ГОЛОВА** и **ХВОСТ**.

### У списков могут быть общие части

На одну ячейку может указывать одна или более стрелок из списочных ячеек, однако из каждого поля ячейки может исходить лишь одна стрелка. Если на некоторую ячейку есть несколько указателей, то эта ячейка будет описывать общее подвыражение. Например, в списке

(кто-то приходит кто-то уходит)

символ **КТО-ТО** является общим подвыражением, на которое ссылаются указатели из поля **CAR** из первой и из третьей ячейки списка.

Если элементами списка являются не атомы, а подсписки, то на месте атомов будут находиться первые ячейки подсписков. Например, построенная вызовом

```
(setq список1 '((b c) a b c)
          (B C) A B C)
```

структура изображена на рис. 2.8.4.

СПИСОК 1

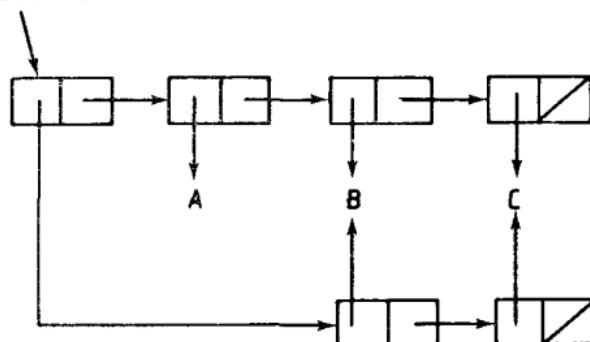


Рис. 2.8.4.

Из этого рисунка видно, что логически идентичные атомы содержатся в системе один раз, однако логически идентичные списки могут быть представлены различными списочными ячейками. Например, значения вызовов

```
(car список1)
(B C)
```

и

```
(caddr список1)
(B C)
```

являются логически одинаковым списком (B C), хотя они и представлены различными списочными ячейками:

```
(equal (car список1) (caddr список1))
T
```

Однако список (B C), как видно из рис. 2.8.5, может состоять и из тех же ячеек.

Эту структуру можно создать с помощью следующей последовательности вызовов:

```

(setq bc '(b c))
(B C)
(setq abc (cons 'a bc))
(A B C)
(setq список2 (cons bc abc))
((B C) A B C)
список2
((B C) A B C)

```

Таким образом, в зависимости от способа построения логическая и физическая структуры двух списков могут оказаться различными. Логическая структура всегда топологически имеет форму двоичного дерева, в то время как физическая структура может быть ациклическим графом, или, другими словами, ветви могут снова сходиться, но никогда не могут образовывать замкнутые циклы, т.е. указывать назад. В дальнейшем мы увидим, что, используя псевдофункции, изменяющие структуры (поля) (RPLACA, RPLACD и другие), можно создать и циклические структуры.

### СПИСОК 2

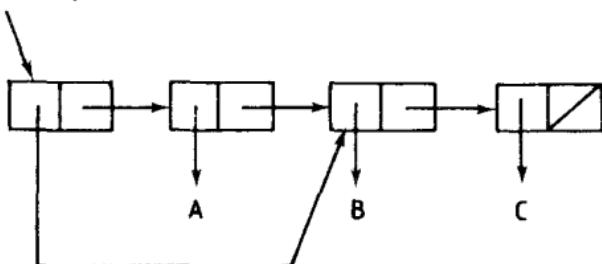


Рис. 2.8.5.

### Логическое и физическое равенство не одно и то же

Логически сравнивая списки, мы использовали предикат EQUAL, сравнивающий не физические указатели, а совпадение структурного построения списков и совпадение атомов, формирующих список.

Предикат EQ можно использовать лишь для сравнения двух символов. Во многих реализациях языка Лисп



предикат EQ обобщен таким образом, что с его помощью можно определить физическое равенство двух выражений (т.е. ссылаются ли значения аргументов на один физический лисповский объект) не зависимо от того, является ли он атомом или списком. При сравнении символов все равно, каким предикатом пользоваться, поскольку атомарные объекты хранятся всегда в одном и том же месте. При сравнении списков нужно поступать осторожнее.

Вызовы функции EQ из следующего примера возвращают в качестве значения NIL, так как логически одинаковые аргументы в данном случае представлены в памяти физически различными ячейками:

```
(eq '((b c) a b c) '((b c) a b c))
NIL
(eq список1 список2)
NIL ; рис. 2.8.4 и 2.8.5
(eq (car список1) (cddr список1))
NIL ; рис. 2.8.4
```

Поскольку части CAR и CDR списка СПИСОК2 представлены при помощи одних и тех же списочных ячеек, то получим следующий результат:

```
(eq (car список2) (cddr список2))
T
```

### Точечная пара соответствует списочной ячейке



Определяя базовую функцию CONS, мы предполагали, что ее вторым аргументом является список. Это ограничение не является необходимым, так как при помощи списочной ячейки можно было бы, например, результат вызова

```
(cons 'a 'b)
```

представить в виде структуры, изображенной на рис. 2.8.6.

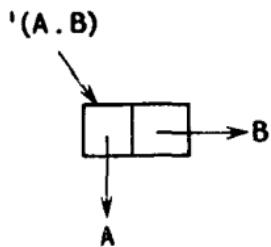


Рис. 2.8.6.

На рис. 2.8.6 показан не список, а более общее символьное выражение, так называемая *точечная пара* (dotted pair). Для сравнения на рис. 2.8.7 мы изобразили список (A B).

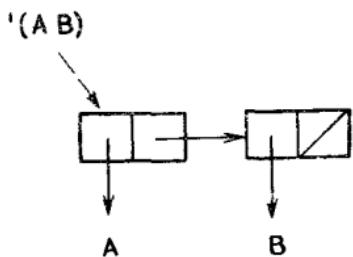


Рис. 2.8.7.

Название точечной пары происходит из использованной в ее записи *точечной нотации* (dot notation), в которой для разделения полей CAR и CDR используется выделенная пробелами точка:

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

Выражение слева от точки (атом, список или другая точечная пара) соответствует значению поля CAR списочной ячейки, а выражение справа от точки – значению поля CDR. Базовые функции CAR и CDR действуют совершенно симметрично:

~~(car '(a . b)) ; обратите внимание на  
 А ; пробелы, выделяющие точку  
 (cdr '(a . (b . c)))  
 (B . C)~~

Точечная нотация позволяет расширить класс объектов, изображаемых с помощью списков. Ситуацию можно было бы сравнить с началом использования дробей или комплексных чисел в арифметике.

### Варианты точечной и списочной записей

Любой список можно записать в точечной нотации. Преобразование можно осуществить (на всех уровнях списка) следующим образом:

"Точка – изначальный элемент всех знаков и их центр."

$$(a_1 \ a_2 \ \dots \ a_N)$$
  

$$\Downarrow$$
  

$$(a_1 . (a_2 . \dots (a_N . \text{NIL}) \dots ))$$

Приведем пример:

$$(a \ b \ (c \ d) \ e)$$
  

$$\Downarrow$$
  

$$(a . (b . ((c . (d . \text{NIL})) . (e . \text{NIL}))))$$

Признаком списка здесь служит NIL в поле CDR последнего элемента списка, символизирующий его окончание.

Транслятор может привести записанное в точечной нотации выражение частично или полностью к списочной нотации. Приведение возможно лишь тогда, когда поле CDR точечной пары является списком или парой. В этом случае можно опустить точку и скобки вокруг выражения, являющегося значением поля CDR:

$(a_1 . (a_2 \ a_3))$	$\Leftrightarrow$	$(a_1 \ a_2 \ a_3)$
$(a_1 . (a_2 . a_3))$	$\Leftrightarrow$	$(a_1 \ a_2 . a_3)$
$(a_1 \ a_2 . \text{NIL})$	$\Leftrightarrow$	$(a_1 \ a_2 . ())$
		$\Leftrightarrow$
		$(a_1 \ a_2)$

Точка останется в выражении лишь в случае, если в правой части пары находится атом, отличный от NIL. Убедиться в том, что произведенные интерпретатором преобразования верны, можно, нарисовав структуры, соответствующие исходной записи и приведенному виду:

```

'(a . (b . (c . (d))))
(A B C D)
'((a b) . (b c))
((A B) B C)
'(a . nil)
(A)
'(a . (b . c))
(A B . C)
'(((nil . a) . b) . c) . d)
(((NIL . A) . B) . C) . D)

```

Использование точечных пар в программировании на Лиспе в общем-то излишне. С их помощью в принципе можно несколько сократить объем необходимой памяти. Например структура данных

**(a b c)**

записанная в виде списка, требует трех ячеек, хотя те же данные можно представить в виде

**(a b . c)**

требующем двух ячеек. Более компактное представление может несколько сократить и объем вычислений за счет меньшего количества обращений в память.

Точечные пары применяются в теории, книгах и справочниках по Лиспу. Часто с их помощью обозначают список заранее неизвестной длины в виде

**(голова . хвост)**

Точечные пары используются совместно с некоторыми типами данных и с ассоциативными списками, а также

в системном программировании. Большинство программистов не используют точечные пары, поскольку по сравнению с требуемой в таком случае внимательностью получаемый выигрыш в объеме памяти и скорости вычислений обычно не заметен.

### Управление памятью и сборка мусора

В результате вычислений в памяти могут возникать структуры, на которые потом нельзя сослаться. Это происходит в тех случаях, когда вычисленная структура не сохраняется с помощью SETQ или когда теряется ссылка на старое значение в результате побочного эффекта нового вызова SETQ или другой функции. Если, например, изображеному на рис. 2.8.8 списку СПИСОК3

```
(setq список3
      '((это станет мусором) cdr часть))
((ЭТО СТАНЕТ МУСОРОМ) CDR ЧАСТЬ)
```

присвоить новое значение

```
(setq список3 (cdr список3))
(CDR ЧАСТЬ)
```

то CAR-часть как-бы отделяется, поскольку указатель из атома СПИСОК3 начинает ссылаться так, как это изображено на рисунке при помощи штриховой стрелки. Теперь уже нельзя через символы и указатели добраться до четырех списочных ячеек. Говорят, что эти ячейки стали *мусором*.



Мусор возникает и тогда, когда результат вычисления не связывается с какой-нибудь переменной. Например, значение вызова

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

лишь печатается, после чего соответствующая ему структура останется в памяти мусором.

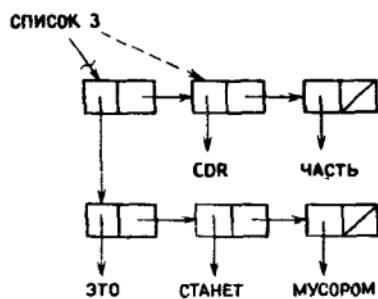


Рис. 2.8.8.

Для повторного использования ставшей мусором памяти в Лисп-системах предусмотрен специальный **мусорщик** (garbage collector), который автоматически запускается, когда в памяти остается мало свободного места. Мусорщик перебирает все ячейки и собирает являющиеся мусором ячейки в список свободной памяти (free list) для того, чтобы их можно было использовать заново.

Пользователь может заметить работу мусорщика только по тому, что вычисления время от времени приостанавливаются на момент, когда система выводит успокаивающее программиста сообщение. Для сборки мусора в разных системах предусмотрены различные процедуры. В некоторых системах мусорщик непрерывно работает на фоне вычислений. В таких системах не происходит внезапных остановок вычислений, которые недопустимы в так называемых системах реального времени.

### Вычисления, изменяющие и не изменяющие структуру

Все рассмотренные до сих пор функции манипулировали выражениями, не вызывая каких-либо изменений в уже существующих выражениях. Например, функция CONS, которая вроде бы изменяет свои аргументы, на самом деле строит новый список, функции CAR и CDR в свою очередь лишь выбирают один из указателей. Структура существующих выражений не могла измениться как побочный эффект вызова функции. Значе-

ния переменных можно было изменить лишь целиком, вычислив и присвоив новые значения целиком. Самое большое, что при этом могло произойти со старым выражением, – это то, что оно могло пропасть.

В Лиспе все-таки есть и специальные функции, при помощи которых на структуры уже существующих выражений можно непосредственно влиять так же, как, например, в Паскале. Это осуществляют функции, которые, как хирург, оперируют на внутренней структуре выражений.



Такие функции называют *структуро-разрушающими* (*destructive*), поскольку с их помощью можно разорвать структуру и склеить ее по-новому.

### RPLACA и RPLACD изменяют содержимое полей

Основными функциями, изменяющими физическую структуру списков, являются RPLACA (*replace CAR*) и RPLACD (*replace CDR*)<sup>1)</sup>, которые уничтожают прежние и записывают новые значения в поля CAR и CDR списочной ячейки:

**(RPLACA ячейка значение-поля) ; поле CAR  
(RPLACD ячейка значение-поля) ; поле CDR**



Первым аргументом является указатель на списочную ячейку, вторым – записываемое в нее новое значение поля CAR или CDR. Обе функции возвращают в качестве результата указатель на измененную списочную ячейку.

```
(setq поезд '(паровоз1 А В С))
(ПАРОВОЗ1 А В С)
  (rplaca поезд 'паровоз2)
(ПАРОВОЗ2 А В С)
```

<sup>1)</sup> Т.е. замена CAR, или "головы" списка, и замена CDR, или "хвоста" списка.– Прим.ред.

```

поезд
(ПАРОВОЗ А В С)
(рplaca (cdr поезд) 'тендер)
(ТЕНДЕР В С)
поезд
(ПАРОВОЗ ТЕНДЕР В С)

```

Функция RPLACD выполняется так же, как RPLACA, с той разницей, что меняется значение поля CDR:

```

(рplacd поезд '(к л м))
(ПАРОВОЗ К Л М)
поезд
(ПАРОВОЗ К Л М)

```

Используя функцию RPLACD, можно, например, определить функцию КРУГ, превращающую произвольный список в кольцо:

```

(defun круг (x) (делай-круг x x))
КРУГ
_ (defun делай-круг (x y)
  (cond ((null x) x)
        ((null (cdr x)) (рplacd x y))
        (t (делай-круг (cdr x) y))))
ДЕЛАЙ-КРУГ
_ (круг '(а б с))
(А В С А В С ...

```

Печатая это значение, интерпретатор зациклится.

В Коммон Лиспе поля списочной ячейки являются ячейками памяти, поэтому значения в них можно менять и с помощью обобщенной функции присваивания SETF. Функции RPLACA и RPLACD можно представить через функцию SETF следующим образом:



(RPLACA x y)  $\leftrightarrow$  (SETF (CAR x) y)  
(RPLACD x y)  $\leftrightarrow$  (SETF (CDR x) y)

Приведем пример:

```

поезд
(ПАРОВОЗ2 К L M)
(setf (car поезд) 'паровоз3)
ПАРОВОЗ3
поезд
(ПАРОВОЗ3 К L M)
(setf (fourth поезд) 'тормозной-вагон)
ТОРМОЗНОЙ-ВАГОН
поезд
(ПАРОВОЗ3 К L ТОРМОЗНОЙ-ВАГОН)
(setf (cdr поезд) nil)
NIL
поезд
(ПАРОВОЗ3)

```

### Изменение структуры может ускорить вычисления

Разумно использованные структуроразрушающие функции могут, как и нож хирурга, быть эффективны-



ми и полезными инструментами. Далее мы для примера рассмотрим, как можно с помощью структуроразрушающих псевдофункций повысить эффективность лисповской функции APPEND.

APPEND объединяет в один список списки, являющиеся его аргументами:

```

(setq начало '(a b))
(A B)
(setq конец '(c d))
(C D)
(setq результат (append начало конец))
(A B C D)

```

Может показаться, что приведенный выше вызов APPEND, как бы изменяет указатели так, как это показано штриховыми стрелками на рис. 2.8.9.

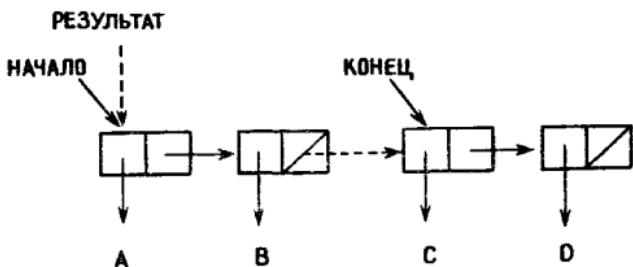


Рис. 2.8.9.

Однако это не верно, так как значение списка **НАЧАЛО** не может измениться, поскольку **APPEND** не является структуроразрушающей функцией. Вычисляется и присваивается лишь новое значение переменной **РЕЗУЛЬТАТ**. Мы получим структуры, изображенные на рис. 2.8.10.

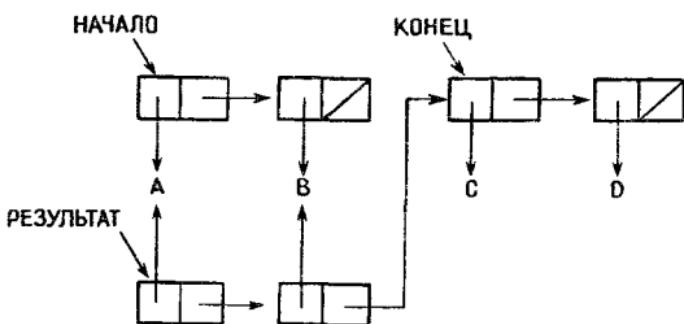


Рис. 2.8.10.

**Из рис.2.8.10 видно, что APPEND создает копию списка, являющегося первым аргументом. Если этот список очень длинный, то долгими будут и вычисления. Создание списочных ячеек с помощью функции CONS требует времени и в будущем добавляет работы мусорщику. Если, например, список НАЧАЛО содержит 1000 элементов, а КОНЕЦ – один элемент, то во время вычисления будет создано 1000 новых ячеек, хотя вопрос состоит лишь в добавлении одного элемента к списку. Если бы последовательность**



аргументов была другой, то создалась бы одна ячейка, и списки были бы объединены приблизительно в 1000 раз быстрее.

Если для нас не существенно, что значение переменной НАЧАЛО изменится, то мы можем вместо функции APPEND использовать более быструю функцию NCONC (concatenate). Функция NCONC делает то же самое, что и APPEND, с той лишь разницей, что она просто объединяет списки, изменяя указатель в поле CDR последней ячейки списка, являющегося первым аргументом, на начало списка, являющегося вторым аргументом, как это показано на рис. 2.8.9.

```

начало
(A B)
конец
(C D)
  (setq изменится (append конец начало))
  (C D A B) ; это значение изменится
  (setq результат1 (nconc начало конец))
  (A B C D) ; разрушающее объединение,
              ; см. рис. 2.8.9
  начало      ; побочный эффект
  (A B C D)  ; см. рис. 2.8.9
  конец
  (C D)
  изменится   ; наблюдается побочный
  (C D A B C D) ; эффект

```

Использовав функцию NCONC, мы избежали копирования списка НАЧАЛО, однако в качестве побочного эффекта изменилось значение переменной НАЧАЛО. Так же изменятся и все другие структуры (например, значение символа ИЗМЕНИТСЯ), использовавшие ячейки первоначального значения переменной НАЧАЛО.

Псевдофункции, подобные RPLACA, RPLACD и NCONC, изменяющие внутреннюю структуру списков, используются в том случае, если нужно сделать небольшие изменения в большой структуре данных. С помощью структуроразрушающих функций можно

эффективно менять за один раз сразу несколько структур, если у этих структур есть совмещенные в памяти подструктуры.

Коварные побочные эффекты, вроде изменения значений внешних переменных, могут привести к сюрпризам в тех частях программы, которые не знают об изменениях. Это осложняет написание, документирование и сопровождение программ. В общем случае функции, изменяющие структуры, пытаются не использовать.

### Упражнения

1. Нарисуйте следующие списки при помощи списочных ячеек и стрелок:

- a) (a)
- b) (a (b (c) d))
- c) (nil (b . c) . d)

2. Почему значением (`(eq '(a b) '(a b))`) будет NIL?

3. Каковы будут значения выражений (`(RPLACA x x)`) и (`(RPLACD x x)`), если

- a)  $x = '(a b)$
- b)  $x = '(a)$

и оба аргумента состоят из различных ячеек? Как изменятся значения, если аргументы будут физически идентичны?

4 Вычислите значения следующих выражений:

- a) (`rplacd '(a) 'b`)
- b) (`rplaca '(a) 'b`)
- c) (`rplacd (cddr '(a b x)) 'c`)
- d) (`rplacd '(nil) nil`)
- e) (`rplacd '(nil) '(nil)`)

5. Что делает следующая псевдофункция:



```
(defun бесполезная (x)
  (cond ((null x) x)
        (t (rplacd x (бесполезная (cdr x))))))
```



*Отсутствие свойств предмета –  
это тоже его свойство.*

*Автор неизвестен*

## 2.9 СВОЙСТВА СИМВОЛА

- У символа могут быть свойства
- У свойств есть имя и значение
- Системные и определяемые свойства
- Чтение свойства
- Присваивание свойства
- Удаление свойства
- Свойства глобальны
- Упражнения

### У символа могут быть свойства

В Лиспе с символом можно связать именованные *свойства* (property). Свойства символа записываются в хранимый вместе с символом *список свойств* (property list, p-list).

### У свойств есть имя и значение

Список свойств может быть пуст или содержать произвольное количество свойств. Его форма такова:



(имя<sub>1</sub> значение<sub>1</sub> имя<sub>2</sub> значение<sub>2</sub>  
... имя<sub>N</sub> значение<sub>N</sub>)

*Ogal – свойство. Рунический знак.*

Например, у символа ЯГОДА-РЯБИНЫ может быть такой список свойств:

(вкус кислый цвет красный)

Список свойств символа можно использовать без особых ограничений, его можно по необходимости обновлять или удалять. Программист должен сам

предусматривать и обрабатывать интересующие его свойства.

### Системные и определяемые свойства

В предыдущих главах мы показали, что с символом связаны лишь его имя, произвольное, назначенное функцией присваивания (`SETQ`), значение и назначенное определением функции (`DEFUN`) описание вычислений (лямбда-выражение). Значение и определение функции являются встроенными системными свойствами, которые управляют работой интерпретатора в различных ситуациях. Функции, используемые для чтения и изменения этих свойств (`SETQ`, `SYMBOL-VALUE`, `DEFUN`, `FUNCTION-VALUE` и другие), мы уже ранее рассматривали. Весь список свойств также является системным свойством. Работающие со свойствами символов прикладные системы могут свободно определять новые свойства.

Далее мы рассмотрим псевдофункции для чтения, изменения и удаления свойств, определяемых пользователем.

#### Чтение свойства



Выяснить значение свойства, связанного с символом, можно с помощью функции `GET`:

**(`GET` символ свойство)**

Если, например, с символом `ЯГОДА-РЯБИНЫ` связан определенный нами ранее список свойств, то мы получим следующие результаты:

```
(get 'ягода-рябины 'вкус)
КИСЛЫЙ
(get 'ягода-рябины 'вес)
NIL
```

Так как у символа `ЯГОДА-РЯБИНЫ` нет свойства `ВЕС`, то `GET` вернет значение `NIL`.

### **Присваивание свойства**

Присваивание нового свойства или изменение значения существующего свойства в основных диалектах языка Лисп осуществляется псевдофункцией PUTPROP (put property)<sup>1)</sup> или PUT:

**(PUTPROP символ свойство значение)**

В Коммон Лиспе функции PUTPROP не существует. Свойства символов находятся в связанных с символами ячейках памяти, для присваивания значений которым используется обобщенная функция присваивания SETF. Присваивание свойства в Коммон Лиспе осуществляется через функции SETF и GET следующим образом:

**(SETF (GET символ свойство) значение)**

Здесь вызов GET возвращает в качестве значения ячейку памяти для данного свойства, содержимое которой обновляется вызовом SETF. Присваивание будет работать и в том случае, если ранее у символа не было такого свойства. Приведем пример:

```
(setf (get 'ягода-рябины 'вес) '(2 g))
(2 g)
(get 'ягода-рябины 'вес)
(2 g)
```

Побочным эффектом вызова будет изменение списка свойств символа ЯГОДА-РЯБИНЫ следующим образом:

**(ВЕС (2 G) ВКУС КИСЛЫЙ ЦВЕТ КРАСНЫЙ)**

Псевдофункция SETF меняет физическую структуру списка свойств. Поэтому использование других списков как части списка свойств без их предварительного копирования может привести к неожиданным ошибкам.

---

<sup>1)</sup> Т.е. "поместить свойство". – Прим. ред.

### **Удаление свойства**

Удаление свойства и его значения осуществляется псевдофункцией REMPROP:

**(REMPROP *символ* *свойство*)**

Приведем пример:

```
(remprop 'ягода-рябины 'вкус)
ВКУС
(get 'ягода-рябины 'вкус)
NIL
```

Псевдофункция REMPROP возвращает в качестве значения имя удаляемого свойства. Если удаляемого свойства нет, то возвращается NIL. Свойство можно удалить, присвоив ему значение NIL. В этом случае имя свойства и значение NIL физически остаются в списке свойств.



Читать из списка свойств, создавать и обновлять в нем свойства можно не только по отдельности, но и целиком. Например, в Коммон Лиспе значением вызова

**(SYMBOL-PLIST *символ*)**

является весь список свойств:

```
(symbol-plist 'ягода-рябины)
(ВЕС (2 G) ЦВЕТ КРАСНЫЙ)
```

### **Свойства глобальны**

Свойства символов независимо от их значений доступны из всех контекстов до тех пор, пока они не будут явно изменены или удалены. Использование символа в качестве функции или переменной, т.е. изменение значения символа или определения функции, не влияет на другие свойства символа, и они сохраняются.

Список свойств используется во многих системных программах Лисп-систем. Наличие свойств полезно как



для поддержки работы самой Лисп-системы, так и во многих типичных случаях представления данных. Использование свойств дает средства для рассматриваемого позже программирования, управляемого данными, с помощью которого можно реализовать различные языки представления знаний и формализмы, такие как семантические сети (*semantic net*), фреймы (*frame*) и объекты объектно-ориентированного программирования (*object, flavor*).

В некоторых системах можно использовать в качестве обобщения так называемые *свободные списки свойств* (*disembodied property list*), несвязанные с каким-либо символом.

### Упражнения

1. Будут ли меняться списки свойств статических и динамических переменных при смене контекста? (Если да, то как?)
2. Предположим, что у имени города есть свойства *x* и *y*, которые содержат координаты места нахождения города относительно некоторого начала координат. Напишите функцию (*РАСТОЯНИЕ a b*), вычисляющую расстояние между городами *a* и *b*, если значением функции (*SQRT x*) является квадратный корень числа *x*.
3. Предположим, что отец и мать некоторого лица, хранятся как значения соответствующих свойств у символа, обозначающего это лицо. Напишите функцию

**(РОДИТЕЛИ x)**



которая возвращает в качестве значения родителей, и предикат

**(СЕСТРЫ-БРАТЬЯ x1 x2)**

который истинен в случае, если  $x1$  и  $x2$  – сестры или братья, родные или с одним общим родителем.

4. Определите функцию REMPROPS, которая удаляет все свойства символа.
5. Функция GET возвращает в качестве результата NIL в том случае, если у символа нет данного свойства, либо если значением этого свойства является NIL. Следовательно, функцией GET нельзя проверить, есть ли некоторое свойство в списке свойств. Напишите предикат

**(HASPROP символ свойство)**

который проверяет, обладает ли символ данным свойством.



*Кто много читает, когда-нибудь захочет и писать.*

*Дж. Крабб*

## 2.10 ВВОД И ВЫВОД

- Ввод и вывод входят в диалог
- READ читает и возвращает выражение
- Программа ввода выделяет формы
- Макросы чтения изменяют синтаксис Лиспа
- Символы хранятся в списке объектов
- Пакеты или пространства имен
- PRINT переводит строку, выводит значение и пробел
- PRIN1 и PRINC выводят без перевода строки
- TERPRI переводит строку
- FORMAT выводит в соответствии с образцом
- Использование файлов
- LOAD загружает определения
- Упражнения

### **Ввод и вывод входят в диалог**

До сих пор в определенных нами функциях ввод данных (READ) и вывод (PRINT) осуществлялись в процессе диалога с интерпретатором. Интерпретатор читал вводимое пользователем выражение, вычислял его значение и возвращал его пользователю. Сами формы и функции не содержали ничего, связанного с вводом или выводом.

Если не использовать специальную команду ввода, то данные можно передавать лисповской функции

только через параметры и свободные переменные. Соответственно, без использования вывода, результат можно получить лишь через конечное значение выражения. Часто все же возникает необходимость вводить исходные дан-



ные и выдавать сообщения и тем самым управлять и получать промежуточные результаты во время вычислений, как это делается и в других языках программирования.

Далее мы рассмотрим чтение и выдачу результатов, осуществляемые между Лисп-системой и пользователем. Эти функции, как будет видно в конце главы, подходят и для управления файлами.

### **READ читает и возвращает выражение**

Лисповская функция чтения READ отличается от ввода в других языках программирования тем, что она обрабатывает выражение целиком, а не одиночные элементы данных. Вызов этой функции осуществляется пользователем (немного упрощенно) в виде



**(READ)**

Как только интерпретатор встречает предложение READ, вычисления приостанавливаются до тех пор, пока пользователь не введет какой-нибудь символ или целиком выражение:

<b>(read)</b>	
<b>(вводимое выражение)</b>	<b>; выражение</b>
	<b>; пользователя</b>
<b>(ВВОДИМОЕ ВЫРАЖЕНИЕ)</b>	<b>; значение функции</b>
<b>...</b>	<b>; READ</b>

Обратите внимание, READ никак не показывает, что он ждет ввода выражения. Программист должен сам сообщить об этом при помощи рассматриваемых позже функций вывода. READ лишь читает выражение и возвращает в качестве значения само это выражение, после чего вычисления продолжаются.

У приведенного выше вызова функции READ не было аргументов, но у этой функции есть значение, которое совпадает с введенным выражением. По своему действию READ представляет собой функцию, но у нее есть побочный эффект, состоящий именно во вводе

выражения. Учитывая это, READ является не чистой функцией, а псевдофункцией.

Если прочитанное значение необходимо сохранить для дальнейшего использования, то вызов READ должен быть аргументом какой-нибудь формы, например присваивания (SETQ), которая свяжет полученное выражение:

```
(setq input (read))
(+ 2 3)           ; введенное выражение
(+ 2 3)           ; значение
input
(+ 2 3)
```

Форма,зывающая интерпретатор, и функция READ совместно с другими функциями позволяют читать выражения внешние по отношению к программе. Из них можно строить новые лисповские выражения или целые программы. Построенные структуры можно вычислить, передав их непосредственно интерпретатору:

```
(eval input)
5
(eval (list (read) (read) (read)))
+ 2 3
5
```

### Программа ввода выделяет формы

Функция READ основана на работающей на системном уровне *процедуре чтения* (Lisp reader). Она читает



s-выражение, образуемое последовательностью знаков, поступающих из файла или иного источника. Внешние устройства становятся доступными из Лисп-системы через объекты, называемые *потоками* (stream). На логическом уровне потоки независимо от характера внешнего устройства являются последовательностью читаемых или записываемых знаков или битов. Для ввода и вывода, как и для

двустороннего обмена, существуют свои типы потоков и специальные операции.

### **Макросы чтения изменяют синтаксис Лиспа**

Процедура чтения содержит *анализатор* (parser), проверяющий знаки в читаемой им последовательности. Чтение обычного алфавитно-цифрового знака никаких особых действий не требует, в то время как чтение специального знака, такого как открывающая или закрывающая скобка, пробел, разделяющий элементы, или точка, приводит к специальным действиям. Соответствие между различными знаками и действиями определяется так называемой *таблицей чтения* (read table), которая задает лисповские функции для знаков.

Знаки, вызывающие специальные действия, называют *макрознаками* (macro character) или *макросами чтения* (read macro), поскольку их чтение требует более сложных действий. Таблица чтения доступна программисту, и он может сам определять новые интерпретации знаков и, таким образом, расширять или изменять синтаксис Лиспа.

Действие макроса чтения определяется в Коммон Лиспе при помощи обыкновенной функции. Она читает и возвращает в качестве значения форму, для построения которой она в свою очередь может предварительно использовать макросы. Определим для примера макрос чтения %, действующий так же, как апостроф. Действие блокировки вычисления пользователь может определить в виде функции, которая рекурсивно читает очередное выражение и возвращает его в составе формы QUOTE:

```
_(defun quote-блокировка (поток знак)
  (list 'quote (read)))
```

Функция, определяющая макрос чтения, имеет в Коммон Лиспе два аргумента, первый из которых описывает поток чтения, а значением второго будет

сам макрознак. В данном примере мы не использовали параметры.

Запись символов и определенных для них макроподстановок в таблицу чтения осуществляется командой

**(SET-MACRO-CHARACTER знак функция)**

```
_ (set-macro-character
  #\% 'quote-блокировка)
T
```

Здесь запись `#\%` обозначает знак процента (`%`) как объект с типом данных знак (в дальнейшем к типам данных мы вернемся подробнее).

После этих определений можно (с точки зрения пользователя) знак процента использовать так же, как апостроф:

```
(list %знак %процента)
(%ЗНАК ПРОЦЕНТА)
%(%а %б %с)
(%А (%QUOTE %Б) %С)
```

Таблиц чтения может быть несколько, но процедура чтения использует в каждый момент времени лишь одну таблицу. Текущая таблица сохраняется как значение системной переменной \*READTABLE\*.

Встроенными макросами чтения в Коммон Лиспе являются:

```
( ; начинает ввод списка или точечной пары
) ; заканчивает ввод списка или точечной пары
, ; возвращает очередное выражение в виде
  ; вызова QUOTE
; ; символы до конца строки считаются
  ; комментарием
\ ; выделение одиночного специального знака
` ; выделение нескольких специальных знаков
```

“ ; !...!  
” ; строка: “...”

Макрознаки нельзя использовать в составе символов наподобие обычных знаков, поскольку процедура чтения проинтерпретирует их в соответствии с таблицей как макросы чтения. Для включения таких знаков

в состав имен нужно использовать специальные выделяющие знаки “\” (backslash) и “!” (bar), которые блокируют макрообработку знаков. *Предупреждение: символы ASCII \ и ! соответствуют в Скандинавии буквам Ø и ö.*

 Из-за такой специальной интерпретации без специального выделения или модификации интерпретатора Лиспа в Коммон Лиспе нельзя использовать букву ö<sup>1)</sup>.

Здесь мы не будем подробнее рассматривать использование макросов чтения и таблиц чтения, так как их редко используют в обыкновенном программировании. Определение новых форм в языке Лисп в основном осуществляется через макросы, которые мы рассмотрим позднее.

### Символы хранятся в списке объектов

Читая и интерпретируя знаки, процедура чтения пытается строить атомы и из них списки. Прочитав имя символа, интерпретатор ищет, встречался ли ранее такой символ или он неизвестен. Для нового символа нужно зарезервировать память для возможного значения, определения функции и других свойств. Символы сохраняются в памяти в *списке объектов* (*object list*, *oblist*) или *массиве объектов* (*obaggray*), в котором они проиндексированы на основании своего имени. Список

<sup>1)</sup> Этот пример показывает, что специфика национального языка может вступить в конфликт с некоторыми символами, используемыми в Коммон Лиспе, рассчитанном на использование стандартной клавиатуры с английским языком. См. наше предисловие к книге.—*Прим. ред.*

объектов содержит как созданные пользователем, так и внутрисистемные символы (например, CAR, CONS, NIL и т.д.). Внесение символа в список объектов называют *включением или интернированием (intern)*.

### **Пакеты или пространства имен**

В более новых Лисп-системах, как и в Коммон Лиспе, можно пользоваться несколькими различными списками объектов, которые называют *пакетами (package)* или *пространствами имен (name space)*. Символы из



различных пространств, имеющие одинаковые имена, могут использоваться различным образом. Это необходимо при построении больших систем, при программировании различных ее подсистем программисты частенько используют одинаковые имена для различных целей. Текущее пространство имен определяется по значению глобальной системной переменной. На атомы из других пассивных пакетов можно сослаться, написав перед символом через двоеточие имя пакета:

**пакет:символ**

Перед использованием такой записи необходимо, чтобы символ, на который ссылаются, был объявлен *внешней (external)* переменной пространства имен. (Однако и на остальные, т.е. *внутренние (internal)* переменные, в принципе, можно сослаться, но более специфическим способом.)

**PRINT переводит строку, выводит значение и пробел**

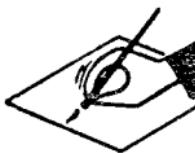
Для вывода выражений можно использовать функцию PRINT. Это функция с одним аргументом, которая сначала вычисляет значение аргумента, а затем выводит это значение. Функция PRINT перед выводом аргумента переходит на новую строку, а после него

выводит пробел. Таким образом, значение выводится всегда на новую строку<sup>1)</sup>:

```
(print (+ 2 3))
5                                ; вывод (эффект)
5                                ; значение
(print (read))
(+ 2 3)                          ; ввод
(+ 2 3)                          ; вывод
(+ 2 3)                          ; значение
```

Как и READ, PRINT является псевдофункцией, у которой есть как побочный эффект, так и значение.

Значением функции является значение его аргумента, а побочным эффектом – печать этого значения.



Лисповские операторы ввода-вывода, как и присваивания, очень гибки, поскольку их можно использовать в качестве аргументов других функций, что в других языках программирования обычно невозможно:

```
(+ (print 2) 3)
2
5
(setq x '(+ 2 3))
(+ 2 3)
(eval (print x))
(+ 2 3)
5
(eval (setq y (print x)))
(+ 2 3)
5
y
(+ 2 3)
```

---

<sup>1)</sup> Во многих Лисп-системах функция PRINT переводит строку не до, а после печати выражения и не оставляет пробела.– Прим. перев.

## PRIN1 и PRINC выводят без перевода строки

Если желательно вывести последовательно на одну строку более одного выражения, то можно использовать функции PRIN1 или PRINC. PRIN1 работает так же, как PRINT, но не переходит на новую строку и не выводит пробел:

```
(progn (prin1 1) (prin1 2) (print 3))
12
3
```

Как функцией PRINT, так и PRIN1 можно выводить кроме атомов и списков и другие типы данных, которые мы рассмотрим позже, например стро-



ки, представляемые последовательностью знаков, заключенных с обеих сторон в кавычки (""). Вывод в такой форме позволяет процедуре чтения (READ) вновь прочесть выведенное выражение в виде, логически идентичном (в смысле функции EQUALP) первоначальному. Таким образом, строка выводится вместе с ограничителями:

```
(prin1 "a b c")
" a b c "
```

Более приятный вид с точки зрения пользователя можно получить при помощи функции PRINC. Она выводит лисповские объекты в том же виде как и PRIN1, но преобразует некоторые типы данных в более простую форму. Такие выведенные выражения нельзя прочесть (READ) и получить выражения, логически идентичные выведенным. Функцией PRINC мы можем напечатать строку без ограничивающих ее кавычек и специальные знаки без их выделения:

```
(princ "a b c")
a b c ; вывод без кавычек
" a b c " ; результат - значение аргумента
```

Предположим, что действует скандинавский вариант кода ASCII ( $\text{ö}=\text{:}$  и  $\text{Ö}=\backslash$ ).

```
(princ 'hÖölMÖöä) ; печать с выделением
HöölMöä           ; выделяющих знаков (Ö) нет
HÖölMÖöä          ; значение
(prin1 'hÖölMÖöä)
HÖölMÖöä          ; выделяющие знаки (Ö) есть
HÖölMÖöä
```

С помощью функции PRINC можно, естественно, напечатать и скобки:



```
(progn (princ "(((")
                (prin1 'луковица)
                (princ "))))")
(((ЛУКОВИЦА)))
"))))"
```

В различных Лисп-системах значения, возвращаемые функциями вывода могут отличаться, хотя функции и называются одинаково. Например, в Маклиспе значением функций PRINT, PRIN1 и PRINC будет Т.

### TERPRI переводит строку

Вывод выражений и знаков часто желательно разбить на несколько строк. Перевод строки можно осуществить функцией PRINT, которая автоматически переводит строку перед выводом, или непосредственно для этого предназначеннной функцией TERPRI (terminate printing). У функции TERPRI нет аргументов и в качестве значения она возвращает NIL:

```
(progn (prin1 'a)
        (terpri)
        (print 'b)
        (prin1 'c))
```

**A**

; два перевода строки (TERPRI и PRINT)  
**B C**  
**C** ; значение

Функцию PRINT можно определить с помощью TERPRI, PRIN1 и PRINC следующим образом:

```
(defun print1 (x)
  (prog2 (terpri) (prin1 x) (princ " ")))
PRINT1
```

### FORMAT выводит в соответствии с образцом

Печать выражений в сложной форме с помощью функций PRINx и TERPRI требует либо предварительного построения специальных структур, либо требует последовательного использования большого числа функций вывода. Программирование, таким образом, усложняется, и поэтому не сразу можно понять, что же на самом деле печатается.



Для решения этих проблем в Коммон Лиспе есть функция FORMAT, при помощи которой можно, используя параметры, гибко задать формат печати. О множестве возможностей и, с

другой стороны, о сложности функции FORMAT свидетельствует то, что в спецификации Коммон Лиспа ей посвящено несколько десятков страниц. Мы рассмотрим лишь самые существенные с точки зрения практического программирования форматы вывода.

Форма вызова функции FORMAT следующая:

**(FORMAT поток образец &REST аргументы)**

Первый аргумент формы поток задает файл или устройство, куда осуществляется вывод. Потоку присваивается значение Т, если вывод осуществляется на экран. При выводе в файл потоком является поток вывода,

представляющий этот файл (подробнее мы вернемся к работе с файлами чуть позже). Вторым аргументом *образец* является ограниченная с обеих сторон кавычками *управляющая строка* (*control string*), которая может содержать *управляющие коды* (*directive*). Они опознаются по знаку ~ (тильда) перед ними. Остальные аргументы формы ставятся в соответствие управляющим кодам строки.

Если управляющие коды и соответствующие им аргументы не используются, то FORMAT выводит строку так же, как функция PRINC, и возвращает в качестве значения NIL.

```
(format t "Это печатается !")
Это печатается !
NIL ; значение
```

Задав NIL значением параметра ПОТОК, получим в качестве результата функции строку, построенную с помощью управляющих кодов и аргументов. У такой формы есть лишь значение и нет побочного эффекта, связанного с выводом.

Гибкость функции FORMAT основана на использовании управляющих кодов. Они выводят в порядке их появления слева направо каждый очередной отформатированный аргумент или осуществляют какие-нибудь действия, связанные с выводом. Наиболее важными управляющими кодами и их назначением являются:

### КОД НАЗНАЧЕНИЕ

- ~% Переводит строку
- ~S Выводит функцией PRIN1 значение очередного аргумента
- ~A Выводит функцией PRINC значение очередного аргумента
- ~nT Начинает вывод с колонки *n*. Если уже ее достигли, то выводится пробел.
- ~~ Выводит сам знак тильды.

Приведем пример:

```

(format t "На~%отдельные~%строки~%")
На                                     ; печать
отдельные                               ; печать
строки                                   ; печать
NIL                                     ; результат
(format nil
  "Ответ - ~S" (+ 2 3))    ; печати нет
"Ответ - 5"                         ; значением является строка
(setq x 2)
2
(setq y 3)
3
(format t "~S плюс ~S будет ~S~%" 
  x y (+ x y))                      ; аргумент
2 плюс 3 будет 5
NIL
(format t "Знак тильды: ~~")
Знак тильды: ~
NIL
(defun таблица (а-список)
  (format t "~%Свойство~15TЗначение~%")
  (do ((пары а-список (rest пары)))
      ((null пары) (terpri))
      (format t "~%~A~15T~A"
        (caar пары) (cadar пары))))
ТАБЛИЦА
(таблица '((имя "Zippy")
            (кличка "Pinhead")
            (язык английский)))
Свойство Значение
ИМЯ      Zippy
КЛИЧКА   Pinhead
ЯЗЫК     АНГЛИЙСКИЙ

```

Различные Лисп-системы дополнительно к основным функциям ввода и вывода содержат целый набор различных встроенных функций, макросов и других средств, при помощи которых можно задавать более сложные виды печати, например *шрифт* (font).

**а  
б  
с  
д  
е  
ф  
г**

Отдельный мир образуют интеллектуальные и программируемые расстровые дисплеи Лисп-машин и более развитых рабочих мест с битовыми картами и средствами указания (мышь, сенсорно-чувствительный дисплей и другие). Они позволяют при вводе и выводе использовать окна и *графические образы* (icons).

Наличие таких средств зависит от возможностей системных и внешних устройств и варьируется от системы к системе. Они не определены в языке Коммон Лисп, и поэтому мы не будем более их рассматривать.

### Использование файлов

Ввод и вывод во время работы с Лиспом в основном осуществляется через дисплей и клавиатуру. Файлы



используются, как правило, лишь для хранения программ в промежутке между сессиями. Однако иногда возникает необходимость работы с файлами из программы. Основные средства работы с файлами нужно знать еще и потому, что во многих системах дисплей и каждое окно в системах с окнами считаются файлами.

В Коммон Лиспе ввод и вывод осуществляются независимо от конфигурации внешних устройств через потоки (stream). Потоки представляют собой специальные резервуары данных, из которых можно читать (поток ввода) или в которые можно писать (поток вывода) знаки или двоичные данные. У системы есть несколько стандартных потоков, которые являются значениями глобальных переменных. Наиболее важные из них

#### \*STANDARD-INPUT\* и \*STANDARD-OUTPUT\*

Эти системные переменные определяют для функций ввода (READ и др.) и соответственно для функций вывода (PRINx, TERPRI и др.) файлы по умолчанию, которыми в начале сеанса являются дисплей или

терминал пользователя. В этом случае нет необходимости в начальном объявлении потока ввода для функции READ или потока вывода для функций вывода (исключением является функция FORMAT).

Если мы хотим обмениваться данными с каким-нибудь новым файлом, то сначала его нужно открыть (open) в зависимости от его использования для чтения или для записи. У директивы OPEN может быть много ключевых параметров. Самым важным из них является :DIRECTION:



### (OPEN *файл &KEY :DIRECTION*)

Значением параметра могут быть следующие ключи:

ПАРАМЕТР	ЗНАЧЕНИЕ
:INPUT	Открыть поток для ввода
:OUTPUT	Открыть поток для вывода
:IO	Открыть двусторонний поток

Ключи в вызове функции не требуют апострофа. Например, следующий вызов открывает поток для вывода в файл ПРОБА.LSP:

```
_ (open 'проба.lsp :direction :output)
  ...
```

Потоковый объект, получаемый в результате вызова OPEN, можно для следующего использования присвоить какой-нибудь переменной:

```
_ (setq поток
      (open 'проба.lsp :direction :io))
  ...
```

Этот вызов присваивает переменной ПОТОК двусторонний поток, связанный с файлом ПРОБА.LSP.

Поток можно при помощи присваивания сделать и системным потоком по умолчанию:

```
(setq *standard-input* поток)
...
```

После этого действие функции READ было бы перенаправлено на файл ПРОБА.LSP, а не на пользовательское устройство ввода.

Наряду с файлами по умолчанию, другим способом задать *источник* (source) для ввода или *получатель* (sink) для вывода является передача желаемого потока в качестве параметра вызова. Для функций READ, PRINx и TERPRI это осуществляется через необязательный параметр. Более подробным описанием параметров этих функций будет:

```
(READ x &OPTIONAL поток)
(PRINT x &OPTIONAL поток)
(PRIN1 x &OPTIONAL поток)
(PRINC x &OPTIONAL поток)
(TERPRI x &OPTIONAL поток)
```

Таким образом, все ранее рассмотренные функции ввода и вывода можно использовать и с указанием файла, задавая соответствующий файлу поток в качестве необязательного параметра. Например, следующий вызов PRIN1 записывает выражение в файл ПРОБА.LSP, и вызов READ читает это выражение:

```
(prin1 '(в файл ПРОБА.LSP) поток)
(В ФАЙЛ ПРОБА.LSP)
(setq x (read поток))
(В ФАЙЛ ПРОБА.LSP)
x
(В ФАЙЛ ПРОБА.LSP)
```

Чтобы записанные в файл данные сохранились, надо не забыть закрыть файл директивой CLOSE:

```
(CLOSE поток)
```

Вообще-то работа с файлами более удобна через форму WITH-OPEN-FILE:

**(WITH-OPEN-FILE**

*(поток файл режим<sub>1</sub> ... режим<sub>N</sub>)  
форма<sub>1</sub> ... форма<sub>M</sub>)*

Вызов WITH-OPEN-FILE позволяет открыть определенный в пределах формы поток, который автоматически создается в самом начале вычислений и закрывается после их завершения. Такие характеристики потока, как его направление, задаются через :DIRECTION и другие ключи так же, как и в директиве OPEN.

Формы *форма<sub>i</sub>* вызова WITH-OPEN-FILE вычисляются последовательно, и значение последней из них возвращается в качестве результата вызова (неявный progn). Например, приведенный ниже вызов открывает поток с именем ПОТОК1 на файл ПРОБА.LSP, читает оттуда записанное нами ранее выражение и закрывает файл. Результатом будет прочитанное выражение.

```
_ (with-open-file
  (поток1 'проба.lsp :direction :input)
  (read поток1))
(В ФАЙЛ ПРОБА.LSP)
```

**LOAD загружает определения**

На практике написание программ осуществляется записью в файл определений функций, данных и других объектов с помощью имеющегося в программном окружении редактора. После этого для проверки определений вызывают интерпретатор Лиспа, который может прочитать записанные в файл выражения директивой LOAD:

**(LOAD файл)**

Читаемые выражения вычисляются так, как будто бы они были введены пользователем. После загрузки можно использовать функции, значения переменных, значения свойств и другие определения.

Стандарт Коммон Лиспа содержит кроме описанных выше средств большое количество возможностей для работы с файлами, в том числе переименование и удаление файлов. Имена файлов можно задавать либо именем относительно используемой системы (pathname), либо не зависящим от устройств логическим именем (pathname).

### Упражнения

- Напишите функцию ЧИТАЙ-ФРАЗУ, которая вводит фразу на естественном языке, заканчивающуюся вопросительным или восклицательным знаком, и преобразует его в список:

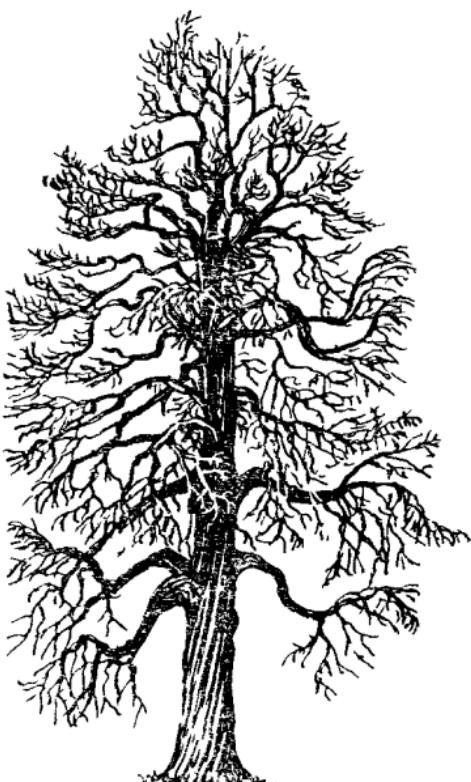
 `_ (читай-фразу)` ; вызов  
`куда ветер дует ?` ; ввод  
`(КУДА ВЕТЕР ДУЕТ ?)` ; результат

- Запрограммируйте функцию ЧИТАЙ, которая вводит вызов функции в виде  
 $fn(a_1 a_2 \dots a_N)$   
 и возвращает значение вызова:

`_ (читай)`  
`+(2 3)`  
`5`  
`_ (читай)`  
`cons ('a '(b c))`  
`(a b c)`

- Определите функцию (ЧЕРТА  $n$ ), печатающую  $n$  раз звездочку (\*).
- Используя функцию ЧЕРТА, напишите функцию  
**(ПРЯМОУГОЛЬНИК  $n m$ )**  
 заполняющую всю область  $n \times m$  звездочками.
- Определите функцию, которая спрашивает у пользователя имя и вежливо ему отвечает:

(начали)  
Введите Ваше имя: Рейска  
Прекрасно, РЕЙСКА, программа работает!



## **3 ФУНКЦИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ**

### **3.1 ОСНОВЫ РЕКУРСИИ**

### **3.2 ПРОСТАЯ РЕКУРСИЯ**

### **3.3 ДРУГИЕ ФОРМЫ РЕКУРСИИ**

### **3.4 ФУНКЦИИ БОЛЕЕ ВЫСОКОГО ПОРЯДКА**

### **3.5 ПРИМЕНЯЮЩИЕ ФУНКЦИОНАЛЫ**

### **3.6 ОТОБРАЖАЮЩИЕ ФУНКЦИОНАЛЫ**

### **3.7 ЗАМЫКАНИЯ**

### **3.8 АБСТРАКТНЫЙ ПОДХОД**

### **3.9 МАКРОСЫ**

*В предыдущей главе мы познакомились с основными понятиями и структурами данных Лиспа и используемыми в нем управляющими структурами и средствами определения. Рассмотрим теперь, каким образом можно комбинировать существующие базовые элементы и как их затем использовать для реализации разных механизмов и средств более высоких уровней.*

*Основное внимание будет сосредоточено на главном для Лиспа способе так называемого функционального программирования и описании его отличий от традиционного операторного стиля программирования. Мы познакомимся сначала с принципами рекурсии и ее применением. Далее рассмотрим функции более высокого порядка, получающие в качестве фактического параметра кроме данных еще и функции, а значением которых снова может быть функция. Это возможно, поскольку в Лиспе данные и программы представляются одинаково. При помощи функций более высокого порядка можно, в частности, обобщить определение функций или запрограммировать циклические структуры. И, наконец, мы познакомимся с использованием макросов в Лиспе. Макросами являются функции, в результате вычислений которых возникает новая, чаще более сложная, форма, которая затем снова вычисляется. Макросы позволяют, в частности, расширить набор воспринимаемых Лиспом структур или, другими словами, расширить синтаксис языка.*

*Головной мозг – это орган,  
которым мы думаем, будто  
мы думаем.*

*А. Бирс*

### 3.1 ОСНОВЫ РЕКУРСИИ

- Лисп – это язык функционального программирования
- Процедурное и функциональное программирование
- Рекурсивный – значит использующий самого себя
- Рекурсия всегда содержит терминальную ветвь
- Рекурсия может проявляться во многих формах
- Списки строятся рекурсивно
- Лисп основан на рекурсивном подходе
- Теория рекурсивных функций

#### **Лисп – это язык функционального программирования**

По одной из классификаций языки программирования делятся на *процедурные* (procedural), называемые также *операторными* или *императивными* (imperative), и *декларативные* (declarative) языки. Подавляющее большинство используемых в настоящее время языков программирования, например Бейсик, Кобол, Фортран, Паскаль, Си и Ада, относятся к процедурным языкам. Наиболее существенными классами декларативных языков являются *функциональные* (functional), или *аппликативные*, и *логические* (logic programming) языки. К категории функциональных языков относятся, например Лисп, FP, Apl, Nial, Krc и Logo. Самым известным языком логического программирования является Пролог.

На практике языки программирования не являются чисто процедурными, функциональными или логическими, а содержат в себе черты языков различных типов. На процедурном языке часто можно написать

функциональную программу или ее часть и наоборот. Может точнее было бы вместо типа языка говорить о стиле или методе программирования. Естественно различные языки поддерживают разные стили в разной степени.

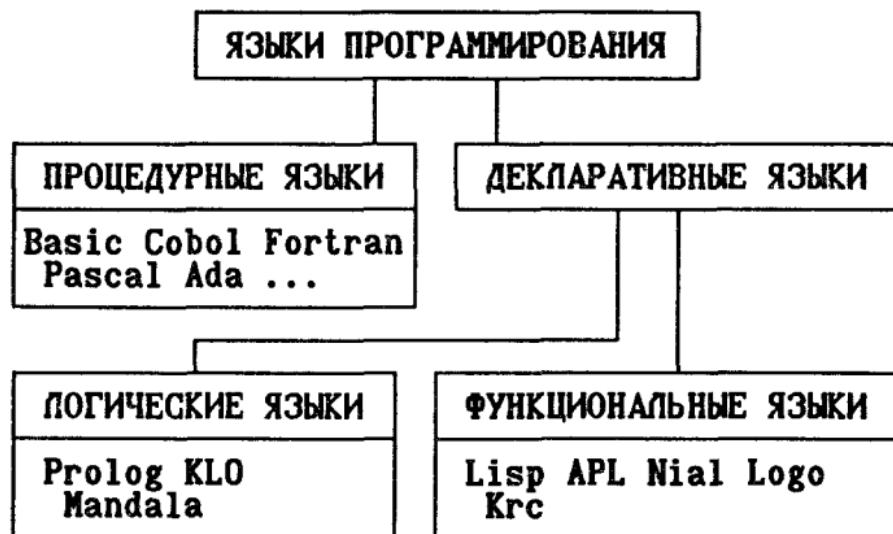


Рис. 3.1.1 Классификация языков и стилей программирования.

### Процедурное и функциональное программирование

Процедурная программа состоит из последовательности операторов и предложений, управляющих последовательностью их выполнения. Типичными операторами являются операторы присваивания и передачи управления, операторы ввода-вывода и специальные предложения для организации циклов. Из них можно составлять фрагменты программ и подпрограммы. В основе такого программирования лежат взятие значения какой-то переменной, совершение над ним действия и сохранение нового значения с помощью оператора присваивания, и так до тех пор пока не будет получено (и, возможно, напечатано) желаемое окончательное значение.

Функциональная программа состоит из совокупности определений функций. Функции, в свою очередь, представляют собой вызовы других функций и предложений, управляющих последовательностью вызовов. Вычисления начинаются с вызова некоторой функции, которая в свою очередь вызывает функции, входящие в ее определение и т.д. в соответствии с иерархией определений и структурой условных предложений. Функции часто либо прямо, либо опосредованно вызывают сами себя.

Каждый вызов возвращает некоторое значение в вызвавшую его функцию, вычисление которой после этого продолжается; этот процесс повторяется до тех пор пока запущенная вычисления функция не вернет конечный результат пользователю.

"Чистое" функциональное программирование не признает присваиваний и передач управления. Разветвление вычислений основано на механизме обработки аргументов условного предложения. Повторные вычисления осуществляются через рекурсию, являющуюся основным средством функционального программирования.

### **Рекурсивный – значит использующий самого себя**



Многие практические ситуации предполагают *рекурсивное* или самоповторяющееся поведение, возвращающееся к самому себе. Предположим, что мы, например, пытаемся прочитать текст на другом языке, пользуясь толковым словарем этого языка. Когда в тексте

встречается незнакомое слово, то его объяснение ищется в словаре. В объяснении слова могут, в свою очередь, встретиться незнакомые слова, которые нужно найти в словаре и т.д. Эту процедуру можно определить с помощью следующих правил:

### **ВЫЯСНЕНИЕ ЗНАЧЕНИЯ СЛОВА:**

1. Найди слово в словаре.

2. Прочитай статью, объясняющую значение этого слова.
3. Если объяснение понятно, т.е. статья не содержит непонятных слов, продолжи чтение с последнего прерванного места.
4. Если в объяснении встречается незнакомое слово, то прекрати чтение, запомни место прекращения и выясни значение слова, придерживаясь совокупности правил **ВЫЯСНЕНИЕ ЗНАЧЕНИЯ СЛОВА**.

Изображенный выше свод правил называют рекурсивным или использующим себя, поскольку в нем содержится ссылка на собственное определение. Речь идет не о порочном круге определений, а об образе действий, который пригоден для использования и вполне выполним.

#### **Рекурсия всегда содержит терминальную ветвь**

В рекурсивном описании действий имеет смысл обратить внимание на следующие обстоятельства. Во-первых, процедура содержит всегда по крайней мере одну терминальную ветвь и условие окончания (пункт 3). Во-вторых, когда процедура доходит до рекурсивной ветви (пункт 4), то функционирующий процесс приостанавливается и новый такой же процесс запускается с начала, но уже на новом уровне. Прерванный процесс каким-нибудь образом запоминается. Он будет ждать и начнет исполняться лишь после окончания нового процесса. В свою очередь, новый процесс может приостановиться, ожидать и т.д.



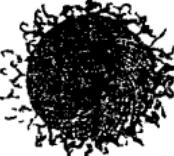
Так образуется как бы стек прерванных процессов, из которых выполняется лишь последний в настоящий момент времени процесс; после окончания его работы продолжает выполняться предшествовавший ему процесс. Целиком весь процесс выполнен, когда стек снова опустеет, или, другими словами, все прерванные процессы выполняются.

## Рекурсия может проявляться во многих формах

В реальном мире рекурсия проявляется в виде различных форм и связей. Она может быть как в структуре, так и в действиях.



Каждому известно изображение, повторяющееся в двух зеркалах, установленных друг против друга, картина, изображающая картину, телевизор, в котором виден телевизор, и т. д. В математике рекурсия встречается в связи с многими различными аспектами, такими как ряды, повторяющиеся линии, алгоритмы, процедуры определения и доказательства, одним словом, в самых различных структурах.



Рекурсия встречается обычно и в природе: деревья имеют рекурсивное строение (ветки образуются из других веток), реки образуются из впадающих в них рек. Клетки делятся рекурсивно. В растениях это часто видно уже на макроуровне. Например, семенная чешуя шишечек и семена некоторых цветов (например, подсолнечника) частично расположены пересекающимися спиралевидными веерами, определяемыми соотношением чисел Фибоначчи.



*Осознание окружающего мира и самого себя (Forster 1981).*

Продолжение жизни связано с рекурсивным процессом. Молекулы ДНК и вирусы размножаются, копируя себя, живые существа имеют потомство, которое, в свою очередь, тоже имеет потомство и т. д.

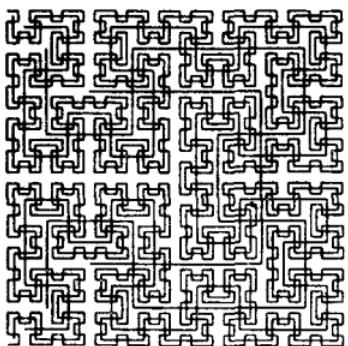
Рекурсия распространена и в языке, и в поведении так же, как в способах рассуждения и познания. Рекурсия в языке, например, может быть в структуре или в содержании:

"Петя сказал, что Вася сказал, что ..."

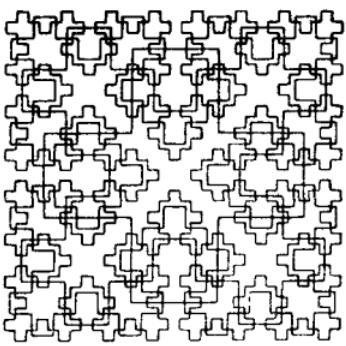
"Знаю, что знаю, но не помню"

"Сделать; заставить сделать; заставить, чтобы заставили сделать; ..."

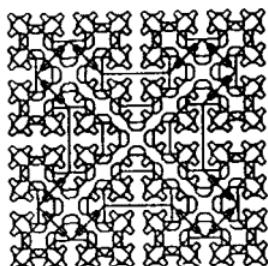
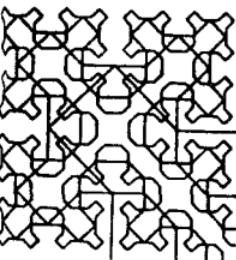
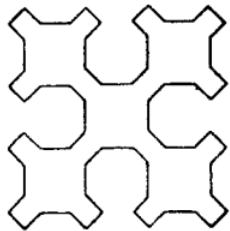
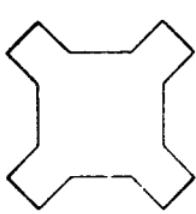
"Замени х этим предложением"



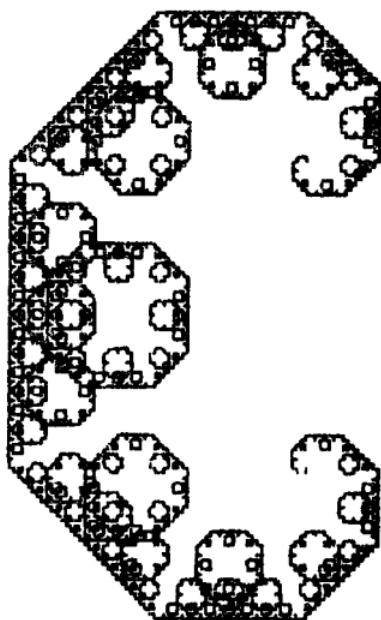
Кривые Гильберта  
1–5 порядков.



Кривые W  
1–4 порядков.



Кривые Серпинского  
1, 2, 3 и 4 порядков.



Рекурсивная кривая С  
или "змейка".

**"Запомни и передай это сообщение"**

...

Музыкальные формы и действия также могут быть рекурсивными во многих отношениях (например, канон, в котором мелодия сопровождается той же мелодией с задержкой, и другие).

Целенаправленное поведение и решение проблем являются рекурсивными процессами. Точно так же и исследования в области искусственного интеллекта рекурсивны в своей попытке исследовать процессы, протекающие в головном мозге, при помощи которых происходит исследование и решение проблем, в том числе и исследования в области искусственного интеллекта.

### **Списки строятся рекурсивно**

Рекурсия в Лиспе основана на математической теории рекурсивных функций. Рекурсия хорошо подходит для работы со списками, так как сами списки могут состоять из подсписков, т.е. иметь рекурсивное строение. Для обработки рекурсивных структур совершенно естественно использование рекурсивных процедур.

Списки можно определить с помощью следующих правил Бэкуса-Наура:

*список → NIL ; список либо пуст, либо это*  
*список → (голова . хвост) ; точечная пара, хвост*  
*; которой является списком*

*голова → атом ; рекурсия "в глубину"*  
*голова → список*

*хвост → список ; рекурсия "в ширину"*

Рекурсия есть как в определении головы, так и в определении хвоста списка. Заметим, что приведенное выше определение напрямую отражает определения функций, работающих со списками, которые могут обрабатывать рекурсивным вызовом голову списка, т.е.

"в глубину" (в направлении CAR), и хвост списка, т.е.  
"в ширину" (в направлении CDR).

Списки можно определить и в следующей форме, которая подчеркивает логическую рекурсивность построения списков из атомов и подсписков:

*список* → *NIL*

*список* → (*элемент* *элемент* ...)

*элемент* → *атом*

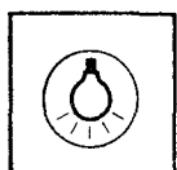
*элемент* → *список* ; *рекурсия*

### Лисп основан на рекурсивном подходе

В программировании на Лиспе рекурсия используется для организации повторяющихся вычислений. На ней же основано разбиение проблемы и разделение ее на подзадачи, решение которых, насколько это возможно, пытаются свести к уже решенным или в соответствии с основной идеей рекурсии к решаемой в настоящий момент задаче.

На рекурсии основан и часто используемый при решении задач и при *поиске* (search) механизм *возвратов* (backtracking), при помощи которого можно вернуться из тупиковой ветви к месту разветвления, аннулировав проделанные вычисления. Рекурсия в Лиспе представляет собой не только организацию вычислений – это образ мыслей и методология решения задач.

### Теория рекурсивных функций

 *Теория рекурсивных функций* наряду с алгеброй списков и лямбда-исчислением является еще одной опорой, на которой покоится Лисп. В этой области математики изучаются теоретические вопросы, связанные с *вычислимостью* (computability). Под вычислимыми понимаются такие задачи, которые в принципе можно запрограммировать и решить с помощью вычислительной машины. Теория рекурсивных функций предлагает наряду с машиной Тьюринга, лямбда-исчислением и другими теоретич-

кими формализмами эквивалентный им формализм алгоритмической вычислимости (*effective computability*).

В теории рекурсивных функций сами функции (алгоритмы) и их свойства рассматриваются и классифицируются в соответствии с тем, какие функции можно получить и вычислить, используя различные формы рекурсии. Основная идея рекурсивного определения заключается в том, что функцию можно с помощью *рекуррентных формул* (немецкое *formula*) свести к некоторым начальным значениям, к ранее определенным функциям или к самой определяемой функции, но с более "простыми" аргументами. Вычисление такой функции заканчивается в тот момент, когда оно сводится к известным начальным значениям. Например, числа Фибоначчи

$0, 1, 1, 2, 3, 5, \dots$

используя префиксную нотацию Лиспа, можно определить с помощью следующих рекуррентных формул:

$$\begin{aligned}(fib\ 0) &= 0, \\(fib\ 1) &= 1, \\(fib\ n) &= (+\ (fib\ (-\ n\ 1))\ (fib\ (-\ n\ 2)))\end{aligned}$$

Класс функций, получаемых таким образом, называют классом *примитивно рекурсивных* (primitive recursive) функций.

Существуют также функции, не являющиеся примитивно рекурсивными. В качестве примера можно привести функцию Аккермана, которую можно задать следующими рекуррентными формулами:

$$\begin{aligned}(Ack\ 0\ x\ y) &= (+\ y\ x), \\(Ack\ 1\ x\ y) &= (*\ y\ x), \\(Ack\ 2\ x\ y) &= (^{\wedge}\ y\ x),\end{aligned}$$

$$\dots (Ack\ (+\ z\ 1)\ x\ y) =$$

к значениям  $y$   $x-1$  раз  
применяется операция  
(*lambda* ( $u\ v$ ) (*Ack*  $z\ u\ v$ ))

Заданная с помощью приведенной выше схемы функция Ack не является примитивно рекурсивной, хотя она и вычислимая, т.е. ее можно определить и на основе этого определения вычислить ее значение за конечное время.

Можно показать, что существуют функции, значения которых можно вычислить с помощью алгоритма, но которые нельзя алгоритмически описать. Вычисление такой функции может быть бесконечным. В качестве примера приведем функцию ( $f\ n\ m$ ), результатом которой является 1 в случае, если в десятичной записи числа  $n$  встречается фрагмент из последовательности повторяющихся цифр  $m$  длиной  $p$ .

Можно показать, что алгоритм вычисления этой функции существует, но нам неизвестно, каков он. Мы можем лишь пытаться вычислять знаки пи в надежде, что искомая последовательность обнаружится, но определить, закончится ли когда-нибудь вычисление, мы не можем. Такие функции называются *общерекурсивными* (general recursive).

Класс примитивно рекурсивных функций достаточно интересен с точки зрения многих практических проблем. Применение рекурсивных функций в Лиспсе не ограничивается лишь численными аргументами, они используются (и даже в первую очередь) для символьных структур, что открывает новые большие возможности по сравнению с численными вычислениями.

В последующих разделах мы более подробно рассмотрим использования рекурсии и техники функционального программирования в Лиспсе. Мы попытаемся:

1. Научиться функциональному и рекурсивному подходу к решению задач.
2. Научиться записывать действия над списками и преобразование списков в виде рекурсивных функций.
3. Научиться использовать разные виды рекурсии и их композиции для решения проблем и задач в различных ситуациях.

4. Одновременно ознакомиться с функциями обработки списков и другими наиболее важными функциями Лиспа.

### Литература

- Brady J. *The Theory of Computer Science, A Programming Approach*. Chapman and Hall, London, 1977.
- Foerster H. *Observing Systems*. The Systems Inquiry Series, Intersystems, Seaside, California, 1981.
- Hofstadter D. *Gödel, Escher, Bach : An Eternal Golden Braid*. Vintage Books, New York, 1979.
- Minsky M. *Computation: Finite and Infinite Machines*. Prentice Hall, Englewood Cliffs, New Jersey, 1967.
- Rogers H. *Theory of Recursive Functions and Effective Computability*. McGraw-Hill, New York, 1967.
- Rozsa P. *Rekursive funktionen in der Komputer-Theorie*. Akademiai Kiado, Budapest, 1976.
- Wegner P. *Programming Languages, Information Structures, and Machine Organisation*. McGraw-Hill, New York, 1968.



*Нужно быть очень терпеливым, чтобы научиться терпению.*

*E. Лец*

## 3.2 ПРОСТАЯ РЕКУРСИЯ

- Простая рекурсия соответствует циклу
- MEMBER проверяет, принадлежит ли элемент списку
- Каждый шаг рекурсии упрощает задачу
- Порядок следования ветвей в условном предложении существенен
- Ошибка в условиях может привести к бесконечным вычислениям
- APPEND объединяет два списка
- REMOVE удаляет элемент из списка
- SUBSTITUTE заменяет все вхождения элемента
- REVERSE обращает список
- Использование вспомогательных параметров
- Упражнения

Функция является рекурсивной, если в ее определении содержится вызов самой этой функции. Мы будем говорить о *рекурсии по значению*, когда вызов является выражением, определяющим результат функции. Если же в качестве результата функции возвращается значение некоторой другой функции и рекурсивный вызов участвует в вычислении аргументов этой функции, то будем говорить о *рекурсии по аргументам*. Аргументом рекурсивного вызова может быть вновь рекурсивный вызов, и таких вызовов может быть много.

**Простая рекурсия соответствует циклу**

Рассмотрим сначала случай простой рекурсии. Мы будем говорить, что рекурсия *простая* (simple), если



вызов функции встречается в некоторой ветви лишь один раз. Простой рекурсии в процедурном программировании соответствует обычный цикл.

Определим функцию КОПИЯ, которая строит копию списка. Копия списка логически идентична первоначальному списку:

```
_defun КОПИЯ (1)
  (cond ((null 1) nil) ;условие окончания
   (t (cons (car 1) ; рекурсия
            (КОПИЯ (cdr 1))))))

КОПИЯ
  (копия '(a b c))
  (A B C) ; логическая копия
  (eq '(a b c) (копия '(a b c)))
  NIL ; физически различные списки
```

Эта функция является рекурсивной по аргументу, поскольку рекурсивный вызов стоит на месте аргумента функции CONS. Условное предложение из тела функции содержит две ветви: ветвь с условием окончания и ветвь с рекурсией, с помощью которой функция проходит список, копируя и укорачивая в процессе этого список по направлению CDR.

Мы можем понаблюдать за работой этой функции при помощи содержащихся в интерпретаторе средств трассировки (TRACE). Трассировка функции включается при помощи директивы, которая в отличие от обычных функций не вычисляет свой аргумент.

### (TRACE функция)

```
(trace копия) ; апостроф ставить не нужно
(КОПИЯ)
```

Трассировку можно отменить аналогичной по форме директивой UNTRACE. После ввода директивы TRACE интерпретатор будет распечатывать значения аргументов каждого вызова функции КОПИЯ перед ее вычисле-

нием и полученный результат после окончания вычисления каждого вызова. При печати вызовы на различных уровнях (*levels*) рекурсии отличаются при помощи отступа. Приведем пример:

<b>(копия '(a b))</b>	
<b>КОПИЯ:</b>	; аргументы вызова
<b>L = (A B)</b>	; 1 уровня
<b>КОПИЯ:</b>	; аргументы вызова
<b>L = (B)</b>	; 2 уровня
<b>КОПИЯ:</b>	; аргументы вызова
<b>L = NIL</b>	; 3 уровня
<b>КОПИЯ = NIL</b>	; значение уровня 3
<b>КОПИЯ = (B)</b>	; значение уровня 2
<b>КОПИЯ = (A B)</b>	; значение уровня 1
<b>(A B)</b>	; окончательное значение



Функция вызывается рекурсивно с хвостом списка L в качестве аргумента до тех пор, пока условие (NULL L) не станет истинным и значением функции не станет NIL, который выступает вторым аргументом последнего вызова функции CONS.

После этого рекурсивные вызовы будут по очереди заканчиваться и возвращаться на предыдущие уровни, и с помощью функции CONS в рекурсивной ветке определения начнет формироваться от конца к началу новый список. На каждом уровне к возвращаемому с предыдущего уровня хвосту (CDR L) добавляется головная часть с текущего уровня (CAR L).

Обратите внимание, что функция копирует не элементы списка (т.е. в направлении CAR, или в глубину), а лишь составляющие список ячейки верхнего уровня (т.е. список копируется в направлении CDR, или в ширину). Позднее мы вернемся к случаю, когда список копируется и по направлению CAR на любую глубину. Для этого в определение нужно добавить лишь еще один рекурсивный вызов.

Копирование списков представляет собой одно из основных действий над списками, и поэтому соответствующая функция входит в число встроенных функций

практически во всех Лисп-системах. И даже если ее нет, то ее очень просто можно определить, как мы это только что сделали. В Коммон Лиспе эта функция называется **COPY-LIST**.

Мы не использовали это имя, а дали функции свое имя, чтобы напрасно не менять определение встроенной функции. Хотя программист в Лиспе может свободно переопределить по своему усмотрению любую функцию, экспериментировать лучше со своими именами, чтобы потом, когда вашим неверным или неполным определением будет вызвана ошибка, не удивляться, почему система не работает так, как надо.

В дальнейшем мы в целях обучения часто будем программировать функции из числа встроенных функций Коммон Лиспа. Будем систематически использовать их стандартные имена, добавив к ним номер варианта, как мы и поступали ранее, определяя различными способами экспоненциальную функцию EXPT (EXPT1, EXPT2 и другие варианты). Так мы одновременно познакомимся с основными функциями и их именами, принятыми в Коммон Лиспе.

Чтобы облегчить восприятие, мы воспользуемся таким типографским приемом, как запись ПРОПИСНЫМИ или строчными буквами. Рекурсивный вызов функции будет выделяться путем написания ее имени прописными буквами.

### **MEMBER** проверяет, принадлежит ли элемент списку

Рекурсию можно использовать для определения как предикатов, так и функций. В качестве второго примера простой рекурсии возьмем встроенный предикат Лиспа **MEMBER**. С его помощью можно проверить, принадлежит ли некоторый элемент данному списку или нет. См. пример на следующей странице.

Тело функции состоит из условного предложения, содержащего три ветви. Они участвуют в процессе вычислений в зависимости от возникающей ситуации:

#### 1. ((null l) nil):

```

(defun MEMBER1 (a l)
  (cond ((null l) nil) ; l пуст?
         ((eql (car l) a) 1) ; a найден?
         (t (MEMBER1 a ; a - в хвосте?
                      (cdr l)))))

MEMBER
(member1 'b '(a b c d))
(B C D)
(member1 'e '(a b c d))
NIL
(member1 '(b c) '(a (b c) d)) ; сравнение
NIL ; предикатом EQL

```

Аргумент – пустой список либо с самого начала, либо потому, что просмотр списка окончен.

### 2. (*((eql (car l) a) 1)*):

Первым элементом является искомый элемент. В качестве результата возвращается список, в котором А – первый элемент.

### 3. (*(t (MEMBER1 a (cdr l)))*):

Ни одно из предыдущих утверждений не верно: в таком случае либо элемент содержится в хвосте списка, либо вовсе не входит в список. Эта задача аналогична первоначальной, только она меньше по объему. Поэтому мы можем для ее решения использовать тот же механизм, или, другими словами, применить сам предикат к хвосту списка.

Если список L пуст либо А в него не входит, то функция возвращает значение NIL. В противном случае она возвращает в качестве своего значения ту часть списка, в которой искомое А является первым элементом. Это отличное от NIL выражение соответствует логическому значению "истина".

**Каждый шаг рекурсии упрощает задачу**

В определении предиката MEMBER1 первоначальная задача разбита на три подзадачи. Первые две из них сводятся к простым условиям окончания. Третья решает такую же задачу, но на шаг более короткую. Ее

решение можно рекурсивно свести к функции, решающей первоначальную задачу.

Понаблюдаем с помощью трассировки за вычислением предиката MEMBER1:

```
(trace member1)
(member1)
_MEMBER1:
(member1 'c '(a b c d))
_MEMBER1: ; вызов уровня 1
A = C
L = (A B C D)
_MEMBER1: ; вызов уровня 2
A = C
L = (B C D)
_MEMBER1: ; вызов уровня 3
A = C
L = (C D)
_MEMBER1 = (C D) ; значение уровня 3
_MEMBER1 = (C D) ; значение уровня 2
_MEMBER1 = (C D) ; значение уровня 1
(C D) ; значение формы
```



На первых двух уровнях рекурсии вычисления осуществляются по третьей, рекурсивной ветви. В рекурсивном вызове первым аргументом является С, так как искомый элемент на каждом шаге один и тот же. Вторым аргументом берется хвост списка текущего уровня (CDR L).

На третьем уровне значением предиката (EQL (CAR L) A) становится Т, поэтому на этом уровне значением всего вызова станет значение соответствующего результирующего выражения L=(C D). Это значение возвращается на предыдущий уровень, где оно будет значением вызова MEMBER1 в рекурсивной ветви и, таким образом, станет значением всего вызова на втором уровне. Далее это значение возвращается далее на уровень и, в конце концов, выводится пользователю.

В процессе спуска по ходу рекурсии на более низкие уровни значение параметра А не меняется, в то время

как значение параметра L меняется при переходе на следующий уровень. Значения предыдущего уровня сохраняются, поскольку связи переменных ассоциируются с уровнем. Значения предыдущих уровней скрыты до тех пор, пока на них не вернется управление, после этого старые связи вновь становятся активными. В приведенном примере после возврата на предыдущие уровни эти связи не используются. Обратите внимание, что связи параметра А на различных уровнях физически различны, хотя значение остается тем же самым.

### Порядок следования ветвей в условном предложении существеннен

В рекурсивном определении существеннен порядок следования условий. Их нужно располагать так, чтобы нужная ветвь была выбрана после отбрасывания условия или ряда условий, а чаще в результате выполнения некоторого условия. Последовательность условий может также повлиять на эффективность вычислений. Правильную последовательность можно получить путем логического рассуждения на основе учета возможных ситуаций. Порядок следования может влиять и на выбор предикатов, например использовать ли сам предикат или его отрицание и т. п.

При определении порядка следования условий основным моментом является то, что сначала проверяются всевозможные условия окончания, а затем ситуации, требующие продолжения вычислений. При помощи условий окончания последовательно проверяются все особые случаи и программируются соответствующие результирующие выражения. На исключенные таким образом случаи можно в последующем не обращать внимания.

Например, в рассмотренном ранее предикате MEMBER1 первое условие (NULL L) будет истиной, когда аргумент L с самого начала пуст или когда первоначально непустой список в процессе рекурсивных вызовов пройден до конца:

```
(member1 'x '()) ; L первоначально пуст
NIL
(member1 'x '(a b c d))
NIL ; L пройден до конца
```

Вычисления в обоих случаях оканчиваются на условии (NULL·L). Обратите внимание, что в приведенных ниже вызовах вычисления останавливаются не по этому условию:

```
(member1 nil '(a b nil d))
NIL D
(member1 nil '(a b c nil))
NIL
```

Рассмотрим в качестве полезного урока вариант предиката MEMBER, в котором условия перечислены в неверном порядке. Для того чтобы была виднее неправильная работа предиката, определим его в отличие от Коммон Лиспа так, чтобы в случае, если элемент найден, он возвращал не хвост списка, а значение T.

```
(defun MEMBER2 (a l)
  (cond ((eql a (car l)) t) ; сначала EQL
        ((null l) nil) ; затем NULL
        (t (MEMBER2 a (cdr l)))))
MEMBER2
(member2 'd '(a b c d))
; D является элементом списка
T ; результат верен
(member2 nil '(nil))
; NIL является элементом (NIL)
T ; результат верен
(member2 nil nil)
; NIL не является элементом
T ; пустого списка. Ошибка!
```

Ошибочный результат является следствием того, что в момент применения предиката EQL неизвестно, завершился ли список или проверяется список (NIL).

*хвост*). Причиной возникновения ошибки служит принятное в Лисп-системе соглашение, что головой пустого списка является NIL, а это приводит к тому, что как (CAR NIL), так и (CAR '(NIL . *хвост*)) возвращают в качестве результата NIL. В "чистом" Лиспе значение формы (CAR NIL) не определено, поскольку NIL – атом. Если бы функция MEMBER2 была определена так, как это сделано у нас, то вызов функции выдавал бы ошибочный результат во всех случаях, когда искомый элемент не входит в список. Если бы ветви EQL и NULL были расположены в том же порядке, как в MEMBER1, то функция работала бы правильно, поскольку ранее проверяемое условие (NULL L) исключало бы возможность применения CAR к пустому списку.

### Ошибка в условиях может привести к бесконечным вычислениям

Отсутствие проверки, ошибочное условие или неверный их порядок могут привести к бесконечной рекурсии. Это произошло бы, например, если в предикате

MEMBER значение аргумента L не укорачивалось на каждом шагу рекурсии формой (CDR L). То же самое случилось бы, если рекурсивная ветвь находилась в условном предложении перед условием окончания. Поэтому существенно, чтобы каждый шаг рекурсии приближал вычисления к условию окончания.



На практике рекурсивные вычисления не могут оказаться бесконечными, поскольку каждый рекурсивный вызов требует некоторого количества памяти (если только интерпретатор или транслятор не преобразовал рекурсию в цикл), а общий объем памяти ограничен. При простой рекурсии память заполняется быстро, но в более сложных случаях вычисления могут оказаться практически бесконечными, другими словами, до исчерпания памяти будет бесполезно истрачено много машинного времени.

Замечание: Предикат MEMBER в Коммон Лиспе определен в более общем виде, чем мы его представили. Ему можно вместо предиката сравнения EQL, являющегося умолчанием, задать при

вызове с помощью ключевого параметра :TEST другой предикат. Например, в следующем примере для сравнения элементов используется предикат EQUAL, который применим к спискам:

```
(member '(c d) '((a b) (c d))
         :test 'equal)
((C D))
```

Во многих системах (например, в Маклиспе) в MEMBER по умолчанию используется сравнение при помощи EQUAL.

На функциях, имеющих своим аргументом функцию, или на функционалах в последующем мы остановимся подробнее.

### APPEND объединяет два списка



Рассмотрим встроенную функцию Лиспа APPEND, объединяющую два списка в один новый список. Функция APPEND, подобно функции COPY-LIST (КОПИЯ), строит новый список из значений, сохраняемых на различных уровнях рекурсии:

```
(defun APPEND1 (x y)
  (cond ((null x) y)
        (t (cons (car x)
                  (APPEND1 (cdr x) y)))))
```

APPEND1

Приведем пример:

```
(append1 '(с л и) '(я н и е))
(С Л И Я Н И Е)
(append1 '(a b) nil)
(А В)
(append1 '(a b nil) '(nil))
(А В NIL NIL)
```

Идея работы функции состоит в том, что рекурсивно откладываются вызовы функции CONS с элементами списка X до тех пор, пока он не исчерпается, после чего в качестве результата возвращается указатель на

список Y и отложенные вызовы, завершая свою работу, формируют результат:

```
(trace APPEND1)
(APPEND1)
  (append1 '(a b) '(c d))
APPEND1:
X = (A B)
Y = (C D)
APPEND1:
X = (B)
Y = (C D)
APPEND1:
X = NIL
Y = (C D)
APPEND1 = (C D)
APPEND1 = (B C D)
APPEND1 = (A B C D)
(A B C D)
```

Обратите внимание, что результирующий список строится от конца первого списка к началу, поскольку вычислявшиеся в процессе рекурсии вызовы функции CONS начинают вычисляться из глубины наружу по мере того, как осуществляется процесс возврата из рекурсии.

В функции APPEND1 мы использовали рекурсию по аргументам. APPEND можно было бы определить и в форме рекурсии по значению:

```
(defun APPEND2 (x y)
  (cond ((null x) y)
        (t (APPEND2 (cdr x)
                     (cons (car x) y)))))

APPEND2
```

Это определение отличается тем, что результат теперь строится непосредственно во втором аргументе, а не где-то на стороне, как это осуществлялось в предыду-

щем определении APPEND<sup>1)</sup>). Трассировку вычислений пусть читатель попробует сам.

Как видно из предыдущих определений, APPEND копирует список, являющийся первым аргументом. Этую функцию часто так и используют в виде (APPEND *список* NIL), когда необходимо сделать копию верхнего уровня списка. Но все-таки лучше использовать форму (COPY-LIST *список*).

Во многих Лисп-системах функция APPEND может иметь переменное число параметров. Такую функцию APPEND можно определить через двуместную функцию APPEND1 следующим образом:

```
(defun APPEND3 (&rest args)
  (cond ((null args) nil)
        (t (append1 (car args)
                     (append3 (cdr args))))))

APPEND3
(append3 '(a b) '(c d) '(e f))
(A B C D E F)
```

Последним аргументом функции APPEND в Коммон Лиспе не обязательно должен быть список. В этом случае результатом будет точечное выражение.

### REMOVE удаляет элемент из списка

Все предыдущие определения функций содержали лишь один рекурсивный вызов. Рассмотрим в качестве следующего примера содержащую две рекурсивные ветви встроенную функцию REMOVE, которая удаляет из списка все совпадающие с данным атомом (EQL) элементы и возвращает в качестве значения список из всех оставшихся элементов. REMOVE можно определить через базисные функции и ее саму следующим образом:



<sup>1)</sup> Более существенное отличие этого определения состоит в том, что в объединенном списке нарушаются исходный порядок следования элементов. — Прим. перев.

```

_(defun REMOVE1 (a l)
  (cond ((null l) nil)
        ((eql a (car l))
         (REMOVE1 a ; убрали элемент a
                  (cdr l)))
        (t (cons (car l) ; a в CAR не было
                  (REMOVE1 a (cdr l))))))
REMOVE1
(remove1 'a '(c d e))
(C D E)
(remove1 'b '(a (b c)) ; элементы
(A (B C)) ; проверяются лишь в
            ; направлении CDR
(remove1 '(a b) '((a b) (c d)))
((A B) (C D)) ; сравнение EQUAL

```

Список L сокращается путем удаления всех идентичных A в смысле EQUAL элементов (вторая ветвь) и копирования в результирующий список (CONS) остальных элементов (третья ветвь) до тех пор, пока условие окончания (NULL) не станет истинным. Результат формируется в процессе возврата аналогично функции APPEND1.

Замечание: Во многих Лисп-системах функция REMOVE определена с использованием предиката EQUAL вместо EQUAL, с помощью такой функции можно удалять элементы, являющиеся списками. В Коммон Лиспе предикат сравнения для функции REMOVE можно задать непосредственно при ее вызове ключевым параметром :TEST таким же образом, как и для функции MEMBER. Приведем пример:

```

_(remove '(a b) '((a b) (c d))
          :test 'equal)
((C D))

```

### SUBSTITUTE заменяет все вхождения элемента

Функция SUBSTITUTE, заменяющая все вхождения данного элемента СТАРЫЙ в списке L на элемент НОВЫЙ, работает подобно функции REMOVE.

Обратите внимание, что и здесь замена производится лишь на самом верхнем уровне списка L, т.е.

```

(defun SUBSTITUTE1 (новый старый 1)
  (cond
    ((null 1) nil)
    ((eql старый (car 1))
      (cons новый ; замена головы
            (SUBSTITUTE1 новый
                           старый ; обработка хвоста
                           (cdr 1))))
    (t (cons (car 1) ; голова не меняется
              (SUBSTITUTE1 новый
                           старый ; обработка хвоста
                           (cdr 1))))))
SUBSTITUTE1
(substitute1 'b 'x '(a x x a))
(A B B A)

```

рекурсия осуществляется только по хвостовой части списка (в направлении CDR). Как и при копировании списка процедуру замены элемента можно обобщить так, чтобы список обрабатывался и в глубину, для этого в случае, когда голова списка является списком, нужно осуществить рекурсию и по головной части. К такому виду рекурсии мы вернемся позднее.

Замечание: Функции SUBSTITUTE в Коммон Лиспе можно через ключевой параметр :TEST передать предикат сравнения таким же образом, как и функции REMOVE. Мы сейчас не будем останавливаться на том, что эти функции, как это станет очевидным при рассмотрении типов данных, в Коммон Лиспе заданы в значительно более общем виде.

### REVERSE обращает список

В приведенных примерах мы просматривали список в соответствии с направлением указателей в списочных ячейках слева направо. Но что делать, если нужно обрабатывать список справа налево, т.е. от конца к началу?

Рассмотрим для примера функцию REVERSE, которая также является встроенной функцией Лиспа. REVERSE изменяет порядок элементов в списке (на верхнем уровне) на обратный.

Для обращения списка мы должны добраться до его последнего элемента и поставить его первым элементом

```

-(defun REVERSE1 (l)
  (cond ((null l) nil)
        (t (append (REVERSE1 (cdr l))
                   (cons (car l) nil)))))

REVERSE1
(reverse1 '(a b c))
(C B A)
(reverse1 '((A B) (C D))) ; обращается
((C D) (A B)) ; лишь верхний уровень

```



обращенного списка. Хотя нам непосредственно конец списка не доступен, можно, используя APPEND, описать необходимые действия. Идея определения состоит в следующем: берем первый элемент списка (CAR L), делаем из него с помощью вызова (CONS (CAR L) NIL) одноэлементный список и объединяем его функцией APPEND с перевернутым хвостом. Хвост списка сначала обращается рекурсивным вызовом (REVERSE1 (CDR L)). Попробуем проследить, как происходит такое обращение:

```

(trace reverse1)
(REVERSE1)
(reverse1 '(a b c))
REVERSE1:
L = (A B C)
REVERSE1:
L = (B C)
REVERSE1:
L = (C)
REVERSE1:
L = NIL
REVERSE1 = NIL
REVERSE1 = (C)
REVERSE1 = (C B)
REVERSE1 = (C B A)
(C B A)

```

Добраться до последнего элемента списка можно, лишь пройдя всю образующую список цепочку слева направо.

во. В функции REVERSE1 список проходится до конца и по пути подходящие элементы списка откладываются в аргументы незавершенных вызовов. Построение обращенного списка в порядке, противоположном следованию элементов исходного списка может начаться лишь после завершения рекурсии. Результат будет сформирован, когда исчерпается стек рекурсивных вызовов.

### Использование вспомогательных параметров



Список является несимметричной структурой данных, которая просто проходится слева направо. Во многих случаях для решения задачи более естественны вычисления, производимые справа налево. Например, то же переворачивание списка было бы гораздо проще осуществить, если бы был возможен непосредственный доступ к последнему элементу списка. Такое противоречие между структурой данных и процессом решения задачи приводит к трудностям программирования и может служить причиной неэффективности.

В процедурных языках программирования существует возможность использования вспомогательных переменных, в которых можно сохранять промежуточные результаты. Например, обращение списка можно осуществить простым переносом элементов списка друг за другом из списка L в результирующий список, используя функцию CONS:

```
(defun reverse2 (l)
  (do ((остаток l (cdr остаток))
        (результат nil
          (cons (car остаток)
                результат)))
      ((null остаток) результат))
REVERSE2
```

В функциональном программировании переменные таким образом не используются. Но соответствующий механизм можно легко осуществить, используя вспомо-

гательную функцию, у которой нужные вспомогательные переменные являются параметрами. Тогда для функции REVERSE мы получим такое определение:

```
_ (defun reverse3 (l)
  (ПЕРЕНОС l nil))
REVERSE3
_ (defun ПЕРЕНОС (l результат)
  (cond ((null l) результат)
         (t (ПЕРЕНОС (cdr l)
                (cons (car l)
                      результат))))))
ПЕРЕНОС
```

Вспомогательная функция ПЕРЕНОС рекурсивна по значению, так как результирующим выражением ее тела является либо непосредственно рекурсивный вызов, либо готовое значение. С помощью этой функции элементы переносятся таким образом, что на каждом шаге рекурсии очередной элемент переходит из аргумента L в аргумент РЕЗУЛЬТАТ. Обращенный список строится элемент за элементом функцией CONS в аргументе РЕЗУЛЬТАТ так же, как и в итеративном варианте. Вычисления производятся по списку L слева направо и соответствуют итеративным вычислениям.

### Упражнения

1. Определите встроенную функцию LAST, возвращающую последний элемент списка.
2. Определите функцию, удаляющую из списка последний элемент.
3. Определите предикат, проверяющий, является ли аргумент одноравневым списком.
4. Определите функцию (ЛУКОВИЦА n), строящую N-уровневый вложенный список, элементом которого на самом глубоком уровне является N.



5. Определите функцию ПЕРВЫЙ-АТОМ, результатом которой будет первый атом списка. Пример:

$\bar{A} \text{ (первый-атом '(((a b)) c d))}$

6. Определите функцию, удаляющую из списка первое вхождение данного элемента на верхнем уровне.

7. Что можно сказать о равенстве значений следующих выражений:

$(\text{append } x (\text{append } y z))$   
 $(\text{append } (\text{append } x y) z)$

8. Запишите выражение (REVERSE (APPEND X Y)), используя те же функции, в другом виде.

9. Заполните значениями функций следующую таблицу:

$fn$	$(fn '(A) NIL)$	$(fn NIL '(A))$
------	-----------------	-----------------

$\text{append:}$

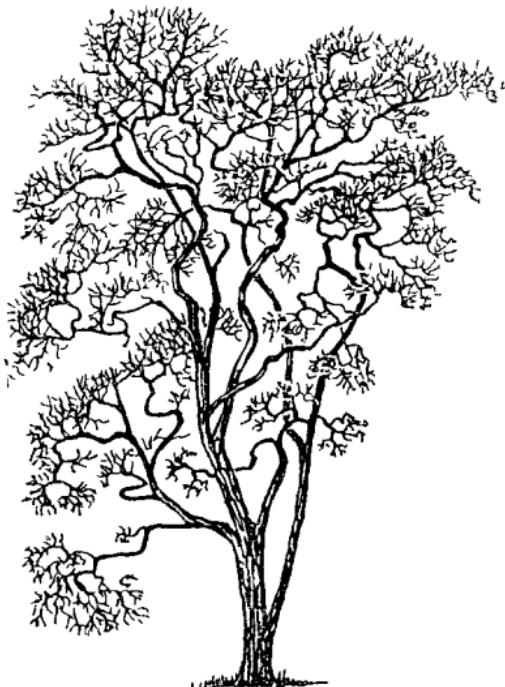
$\text{list:}$

$\text{cons:}$

10. Определите функцию, которая обращает список (a b c) и разбивает его на уровни (((c) b) a)<sup>1)</sup>.
11. Определите функции, преобразующие список (a b c) к виду (a (b (c))) и наоборот.
12. Определите функции, осуществляющие преобразования между видами (a b c) и (((a) b) c).

<sup>1)</sup> Разумеется, построенные функции в этой и последующих задачах должны работать и со списками, являющимися "естественным" обобщением приводимых частных случаев.— Прим. ред.

13. Определите функцию, удаляющую из списка каждый второй элемент.
14. Определите функцию, разбивающую список ( $a\ b\ c\ d\ \dots$ ) на пары  $((a\ b)\ (c\ d)\ \dots)$ .
15. Определите функцию, которая, чередуя элементы списков  $(a\ b\ \dots)$  и  $(1\ 2\ \dots)$ , образует новый список  $(a\ 1\ b\ 2\ \dots)$ .
16. Определите функцию, результатом которой будет выражение, являющееся факториалом числа, и в котором числа (сомножители) упорядочены в порядке возрастания.



*Все трудно до тех пор, пока не станет легким.*

*Т. Фуллер*

### 3.3 ДРУГИЕ ФОРМЫ РЕКУРСИИ

- Параллельное ветвление рекурсии
- Взаимная рекурсия
- Программирование вложенных циклов
- Рекурсия более высокого порядка
- Литература
- Упражнения



В определении функции (или чаще в определениях вызывающих друг друга функций) рекурсия может принимать различные формы. Ранее мы рассмотрели простую рекурсию, когда одиночный рекурсивный вызов функции встречается в одной или нескольких ветвях.

В этой главе мы продолжим изучение различных форм рекурсии, в том числе:

- 1) *параллельную рекурсию*, когда тело определения функции  $f$  содержит вызов некоторой функции  $g$ , несколько аргументов которой являются рекурсивными вызовами функции  $f$ :

```
(defun f ...
  ... (g ... (f ...) ... (f ...) ...)
  ...)
```

- 2) *взаимную рекурсию*, когда в определении функции  $f$  вызывается некоторая функция  $g$ , которая в свою очередь содержит вызов функции  $f$ :

```
(defun f ...
  ... (g ...) ...)
```

$$(defun g \dots  
... (f \dots) \dots)$$

- 3) *рекурсию более высокого порядка*, когда аргументом рекурсивного вызова является рекурсивный вызов:

$$(defun f \dots  
... (f \dots (f \dots) \dots)  
...)$$

### Параллельное ветвление рекурсии

Рекурсию называют *параллельной*, если она встречается одновременно в нескольких аргументах функции. Так выглядят повторяющиеся вычисления, соответствующие следующим друг за другом (текстуально) циклам в операторном программировании.

Рассмотрим в качестве примера копирование списочной структуры на всех уровнях. Ранее уже был представлен пример функции, копирующей список в направлении CDR на верхнем уровне: функция COPY-LIST (КОПИЯ) строила копию верхнего уровня списка. На возможные подсписки не обращалось внимания, и они не копировались, а брались как и атомы в том виде, как они есть (т.е. как указатель). Если нужно скопировать список целиком как в направлении CDR, так и в направлении CAR, то рекурсия должна распространяться и на подсписки. Таким образом, мы получим обобщение функции COPY-LIST Коммон Лиспа COPY-TREE. Слово TREE (дерево) в названии функции возникло в связи с тем, что в определении функции список трактуется как соответствующее точечной паре *бинарное дерево* (binary tree), у которого левое поддерево соответствует голове списка, а правое поддерево – хвосту:

<i>дерево</i> → NIL	; пустое дерево
<i>дерево</i> → атом	; лист дерева
<i>дерево</i> → ( <i>дерево</i> . <i>дерево</i> )	; точечная пара – дерево

```

(defun COPY-TREE1 (l)
  (cond ((null l) nil)
        ((atom l) l)
        (t (cons
              (COPY-TREE1 ; копия головы
                (car l))
              (COPY-TREE1 ; копия хвоста
                (cdr l))))))

COPY-TREE1

```

Функция COPY-TREE отличается от COPY-LIST тем, что рекурсия применяется как к голове, так и к хвосту списка. Отличие состоит также в использовании условия (ATOM L) для определения атомарного подвыражения (листа дерева). Поскольку рекурсивные вызовы представляют собой два аргумента вызова одной функции (CONS), то мы имеем дело с параллельной рекурсией. Заметим, что параллельность является лишь текстуальной или логической, но никак не временной, так как вычисление ветвей естественно производится последовательно.

Таким же образом рекурсией по голове и хвосту списка можно проверить логическую идентичность двух списков на основе сравнения структуры и атомов, составляющих список. Определим далее уже известный нам предикат EQUAL через применяемый к символам предикат EQ: (Ограничим наше рассмотрение лишь символами и списками, состоящими из списков.)

```

(defun EQUAL1 (x y)
  (cond ((null x)(null y))
        ((atom x)
         (cond ((atom y) (eq x y))
               (t nil)))
        ((atom y) nil)
        (t (and (EQUAL1 (car x) ; идентичные
                        (car y) ; головы
                        (EQUAL1 (cdr x) ; идентичные
                            (cdr y)))))) ; хвосты

```

Мы использовали предикат EQ, исходя из первоначального ограничения, т.е. исходя из предположения, что EQ определен лишь на атомарных аргументах. Практически вторую и третью ветви условного предложения можно было бы объединить более короткой ветвью ((EQ X Y) T).

Приведем еще один пример параллельной рекурсии. Рассмотрим функцию ОБРАЩЕНИЕ. Эта функция предназначена для обращения порядка следования элементов списка и его подсписков независимо от их места и глубины вложенности. Ранее определенная нами простой рекурсией функция REVERSE обличала лишь верхний уровень списка:

```

_ (defun ОБРАЩЕНИЕ (l)
  (cond
    ((atom l) l)
    ((null (cdr l))
     (cons (ОБРАЩЕНИЕ (car l)) nil))
    (t (append (ОБРАЩЕНИЕ (cdr l))
                (ОБРАЩЕНИЕ (cons (car l
                                      nil)))))))
ОБРАЩЕНИЕ
(ОБРАЩЕНИЕ '(а (б (с (д)))))
(((д) с) б) а)
_ (setq палиндром
      '((а
          (р о з а)
          (у п а л а)
          (н а)
          (л а п у)
          (а з о р а)))
      ((а) (р о з а) ...))
      (обращение палиндром)
      ((а р о з а) (у п а л) (а н) (а п а п у)
       (а з о р) (а)))

```

Функция ОБРАЩЕНИЕ обращает голову списка, формирует из него список и присоединяет к нему спереди обращенный хвост.

Применяя параллельную рекурсию, можно списочную структуру (двоичного дерева) ужать в одноуровневый список, т.е. удалить все вложенные скобки. Сделать это при помощи структурных преобразований довольно сложно, но с помощью рекурсии это осуществляется просто:

```

_(defun В-ОДИН-УРОВЕНЬ (l)
  (cond
    ((null l) nil)
    ((atom l) (cons (car l) nil))
    (t (append
          (В-ОДИН-УРОВЕНЬ ; сначала голову
          (car l))
          (В-ОДИН-УРОВЕНЬ ; потом хвост
          (cdr l))))))

В-ОДИН-УРОВЕНЬ
(в-один-уровень '(a (((((b))))) (c d) e))
(A B C D E)
_(equal (в-один-уровень палиндром)
        (в-один-уровень (обращение палиндром)))
T

```

Функция В-ОДИН-УРОВЕНЬ объединяет (функцией APPEND) ужатую в один уровень голову списка и ужатый хвост. Если голова списка является атомом, то из него формируется список, поскольку аргументы функции APPEND должны быть списками.

### Взаимная рекурсия

Рекурсия является *взаимной* (mutual) между двумя или более функциями, если они вызывают друг друга. Для примера можно представить ранее определенную нами функцию обращения или зеркального отражения в виде двух взаимно рекурсивных функций следующим образом:

```

_(defun ОБРАЩЕНИЕ (l)
  (cond ((atom l) l)
        (t (ПЕРЕСТАВЬ l nil))))

```

```
(defun ПЕРЕСТАВЬ (l результат)
  (cond ((null l) результат)
        (t (ПЕРЕСТАВЬ (cdr l)
                         (cons (ОБРАЩЕНИЕ (car l))
                               результат)))))
```



*Перевернутое дерево. Кабалистический знак.*

Функция ПЕРЕСТАВЬ используется в качестве вспомогательной функции с дополнительными параметрами таким же образом, как и ранее вспомогательная функция ПЕРЕНОС использовалась совместно с функцией REVERSE3. В процессе построения обращенного списка она заботится и о том, чтобы возможные подсписки были обращены. Она делает это не сама, а передает эту работу специализирующейся на этом функции ОБРАЩЕНИЕ. Результат получен взаимной рекурсией. Глубина и структура рекурсии зависят от строения списка L. Кроме участия во взаимной рекурсии функция ПЕРЕСТАВЬ рекурсивна и сама о себе.

### Программирование вложенных циклов

Соответствующие вложенным (nested) циклам процедурного программирования многократные повторения в функциональном программировании осуществляются обычно с помощью двух и более функций, каждая из которых соответствует простому циклу. Вызов такой рекурсивной функции используется в определении другой функции в качестве аргумента ее рекурсивного вызова. Естественно, аргументом рекурсивного вызова в определении функции может быть другой рекурсивный вызов. Это уже будет рекурсией более высокого порядка.

Рассмотрим сначала программирование вложенных циклов с помощью двух различных функций, а затем уже с помощью рекурсии более высокого порядка. Вложенный цикл можно выразить и с помощью циклических предложений (DO, LOOP и др.) или специализи-

рованных повторяющих функций (функции МАР, например). Такие явные способы мы сейчас использовать не будем.

Мы начнем рассматривать программирование вложенных циклов с сортировки списков. Определим сначала функцию ВСТАВЬ, которая добавляет элемент А в упорядоченный список L так, чтобы сохранилась упорядоченность, если порядок любых двух элементов задается предикатом РАНЬШЕ-Р:

```
(defun ВСТАВЬ (а l порядок)
  (cond ((null l) (list а))
        ((раньше-р а (car l)) порядок)
        (cons а l))
        (t (cons (car l)
                  (ВСТАВЬ а (cdr l)
                            порядок))))
```

ВСТАВЬ

Предикат РАНЬШЕ-Р проверяет, находится ли элемент А ранее элемента В, в соответствии с расположением определенным порядком следования элементов в списке ПОРЯДОК:

```
(defun РАНЬШЕ-Р (а b порядок)
  (cond
    ((null порядок) nil)
    ((eq а (car порядок)) t) ; А раньше
    ((eq b (car порядок)) nil)
    ; В раньше
    (t (РАНЬШЕ-Р а b (cdr порядок)))))

РАНЬШЕ-Р
(раньшер 'b 'e '(a b c d e))
 $\overline{T}$ 
(вставь 'b '(a c d) '(a b c d e))
(A B C D)
```

ВСТАВЬ и РАНЬШЕ-Р образуют двухуровневую вложенную итеративную структуру.

Неупорядоченный список можно отсортировать функцией СОРТИРУЙ; которая рекурсивно ставит

первый элемент списка на соответствующее место в предварительно упорядоченном хвосте списка:

```

_(defun СОРТИРУЙ (l порядок)
  (cond ((null l) nil)
        (t (вставь (car l)
                    (СОРТИРУЙ (cdr l) порядок)
                    порядок)))))

СОРТИРУЙ
(СОРТИРУЙ '(b a c) '(a b c d e))
(A B C)

```

Теперь рекурсия функций СОРТИРУЙ, ВСТАВЬ и РАНЬШЕ-Р образует уже трехуровневую вложенную повторяющуюся структуру.

Приведем еще функцию РАССТАВЬ, напоминающую своей структурой функцию СОРТИРУЙ и позволяющую элементы списка НОВЫЙ вставить в упорядоченный список L. Функция РАССТАВЬ повторяет процедуру вставки для каждого элемента списка НОВЫЙ:

```

_(defun РАССТАВЬ (новый l порядок)
  (cond
    ((null новый) l)
    (t (вставь (car новый)
                (РАССТАВЬ (cdr новый) l
                           порядок)
                порядок)))))

РАССТАВЬ
(расставь '(b c) '(a b d) '(a b c d e))
(A B B C D)

```

Эту рекурсивную по аргументам функцию можно записать и в виде функции, рекурсивной по значению:

```

(defun РАССТАВЬ (новый l порядок)
  (cond
    ((null новый) l)
    (t (РАССТАВЬ (cdr новый)
                  (вставь (car новый) l порядок)
                  порядок))))
```

## Рекурсия более высокого порядка

Выразительные возможности рекурсии уже видны из приведенных выше содержательных и занимающих мало места определений. Определения функций В-ОДИН-УРОВЕНЬ и ОБРАЩЕНИЕ в итеративном варианте не поместились бы на один лист бумаги (попробуйте!). С помощью рекурсии легко работать с динамическими, заранее неопределенными целиком, но достаточно регулярными структурами, такими как списки произвольной длины и глубины вложения.

Используя все более мощные виды рекурсии, можно записывать относительно лаконичными средствами и более сложные вычисления. Одновременно с этим, поскольку определения довольно абстрактны, растет сложность программирования и понимания программы.

Рассмотрим теперь программирование вложенных циклов в такой форме, при которой в определении функции рекурсивный вызов является аргументом вызова этой же самой функции. В такой рекурсии можно выделить различные *порядки* (order) в соответствии с тем, на каком уровне рекурсии находится вызов. Такую форму рекурсии будем называть *рекурсией более высокого порядка*. Функции, которые мы до сих пор определяли, были функциями с рекурсией нулевого порядка.

В качестве классического примера рекурсии более высокого порядка часто приводится известная из теории рекурсивных функций функция Аккермана, пользующаяся славой "плохой" функции:

```
(defun AKKERMAN (m n)
  (cond ((= m 0) (+ n 1))
        ((= n 0) (AKKERMAN (- m 1) 1))
        (t (AKKERMAN (- m 1)
                      (AKKERMAN m (- n 1))))))
```

АККЕРМАН  
(аккерман 2 2)

**(аккерман 3 2)**  
**27**

Функция Аккермана является функцией с рекурсией первого порядка. Ее вычисление довольно сложно, и время вычисления растет лавинообразно уже при малых значениях аргумента.

В качестве другого примера функции с рекурсией первого порядка приведем функцию В-ОДИН-УРОВЕНЬ, располагающую элементы списка на одном уровне, которую мы ранее определили, используя параллельную рекурсию:

```

_(defun в-один-уровень (l)
  (уравнять l nil))
В-ОДИН-УРОВЕНЬ
_(defun ВЫРОВНЯТЬ (l результат)
  (cond ((null l) результат)
        ((atom l) (cons l результат))
        (t (ВЫРОВНЯТЬ (car l)
                      (ВЫРОВНЯТЬ (cdr l)
                                  результат)))))

ВЫРОВНЯТЬ

```

В этом определении работа функции непосредственно сведена к базовым функциям и рекурсии первого порядка, где аргументом рекурсивного вызова является один рекурсивный вызов. В более раннем определении дополнительно к базовым функциям и рекурсии нулевого порядка мы использовали функцию APPEND. Применяя рекурсию более высокого порядка вычисления можно представить более абстрактно и с помощью более короткого определения, однако представить себе работу такой функции довольно сложно.

Функция ВЫРОВНЯТЬ работает следующим образом. Результат строится в списке РЕЗУЛЬТАТ. Если L – атом, то его можно непосредственно добавить в начало списка РЕЗУЛЬТАТ. Если L – список и его первый элемент является атомом, то все сводится к предыдущему состоянию на следующем уровне рекур-

сии, но в такой ситуации, когда список РЕЗУЛЬТАТ содержит уже вытянутый в один уровень оставшийся хвост.

В том случае, когда и голова списка L является списком, то его сначала приводят к одному уровню. Это делается с помощью рекурсивных вызовов, погружающихся в головную ветвь до тех пор, пока там не встретится атом, который можно добавить в начало вытянутой к этому моменту в один уровень структуры. Встречающиеся таким образом атомы добавляются по одному к вытянутому хвосту. На каждом уровне при исчерпании списка на предыдущий уровень выдается набранный к данному моменту РЕЗУЛЬТАТ.

Следующее определение функции REVERSE, использующей лишь базовые функции и рекурсию, является примером еще более глубокого уровня рекурсии:

```
(defun REV (l)
  (cond
    ((null l) l)
    ((null (cdr l)) l)
    (t (cons
          (car (REV (cdr l)))
          (REV (cons (car l)
                     (REV (cdr (REV (cdr l)
                                    )))))))))
```



В определении использована рекурсия второго порядка. Вычисления, представленные этим определением, понять труднее, чем прежние. Сложная рекурсия усложняет и вычисления. В этом случае

невозможно вновь воспользоваться полученными ранее результатами, поэтому одни и те же результаты приходится вычислять снова и снова. Обращение списка из пяти элементов функцией REV требует 149 вызовов. Десятиэлементный список требует уже 74 409 вызовов и заметное время для вычисления! Как правило, многократных вычислений можно избежать, разбив определение на несколько частей и используя подходя-

щие параметры для сохранения и передачи промежуточных результатов.

Обычно в практическом программировании формы рекурсии более высокого порядка не используются, но у них по крайней мере есть свое теоретическое и методологическое значение.

### Литература

1. McCarthy J. *Recursive Functions of Symbolic Expressions and Their Computation by Machine.* CACM, Vol. 3, No. 4, 1960.
2. McCarthy J., Abrahams P., Edwards D., Hart T., Levin M. *LISP 1.5 Programmer's Manual.* MIT Press, Cambridge, Massachusetts, 1962.
3. Henderson P. *Functional Programming, Application and Implementation.* Prentice-Hall, London, 1980.
4. Seppänen J. *Lisp kieleenopas.* Otadata, Espoo, 1972.



### Упражнения

1. Определите функцию, вычисляющую, сколько всего атомов в списке (списочной структуре).
2. Определите функцию, вычисляющую глубину списка (самой глубокой ветви).
3. Запрограммируйте интерпретатор ВЫЧИСЛИ, который преобразует инфиксную запись операций (например, +, -, \* и /) выражения в префиксную и возвращает значение выражения. Пример:

$\frac{6}{(ВЫЧИСЛИ'((-2 + 4) * 3))}$

4. Определите функцию (ПЕРВЫЙ-СОВПАДАЮЩИЙ  $x$   $y$ ), которая возвращает первый элемент, входящий в оба списка  $x$  и  $y$ , в противном случае NIL.

5. Определите предикат **МНОЖЕСТВО-Р**, который проверяет, является ли список множеством, т.е. входит ли каждый элемент в список лишь один раз.
6. Определите функцию **МНОЖЕСТВО**, преобразующую список в множество.
7. Определите предикат **РАВЕНСТВО-МНОЖЕСТВ**, проверяющий совпадение двух множеств (независимо от порядка следования элементов). Подсказка: используйте функцию **REMOVE**, удаляющую данный элемент из множества.
8. Определите функцию **ПОДМНОЖЕСТВО**, которая проверяет, является ли одно множество подмножеством другого. Определите также **СОБСТВЕННОЕ-ПОДМНОЖЕСТВО**.
9. Определите предикат **НЕПЕРЕСЕКАЮЩИЕСЯ**, проверяющий, что два множества не пересекаются, т.е. у них нет общих элементов.
10. Определите функцию **ПЕРЕСЕЧЕНИЕ**, формирующую пересечение двух множеств, т.е. множество из их общих элементов.
11. Определите функцию **ОБЪЕДИНЕНИЕ**, формирующую объединение двух множеств.
12. Определите функцию **СИММЕТРИЧЕСКАЯ-РАЗНОСТЬ**, формирующую множество из элементов не входящих в оба множества.
13. Определите функцию **РАЗНОСТЬ**, формирующую разность двух множеств, т.е. удаляющую из первого множества все общие со вторым множеством элементы.
14. Определите с помощью упоминавшегося в тексте предиката **РАНЬШЕ-Р** предикат (**АЛФАВИТ-Р**  $x\ y$ ), определяющий стоят ли слова, заданные списком

букв, в алфавитном (словарном) порядке. После этого определите функцию (**СЛОВАРЬ**  $x$ ), которая сортирует в алфавитном порядке неупорядоченное множество слов. Пример:

```
_ (словарь '((ф у н к ц и я)
              (цикл)
              (р е к у р с и я)))
((РЕКУРСИЯ) (ФУНКЦИЯ) (ЦИКЛ))
```

Определите с их помощью функцию, составляющую из данных слов обратный словарь, т.е. перечень слов, который упорядочен по буквам, начиная от конца слова к его началу:

```
_ (ословарь '((ф у н к ц и я)
              (цикл)
              (р е к у р с и я)))
((ЦИКЛ) (РЕКУРСИЯ) (ФУНКЦИЯ))
```

15. Упорядоченное бинарное дерево состоит из узлов вида:

(элемент левое-поддерево правое-поддерево)

В каждом узле выполнено следующее условие: все элементы из узлов его левого поддерева в некотором упорядочении (например, по числовой величине или в алфавитном порядке) предшествуют элементу из узла и соответственно элементы из узлов правого поддерева следуют за ним. Пример:

```
(5 (3 (1 nil nil)
      (4 nil nil))
  (7 (6 nil nil)
    (13 (11 nil nil)
        (15 nil nil))))
```

Определите функцию (**ИЩИ**  $a$  дерево), которая ищет в дереве элемент  $a$ .

16. Определите функцию (**ДОБАВЬ** *a дерево*), которая добавляет в упорядоченное дерево *дерево* элемент *a*. (Подсказка: копируйте дерево по пути поиска и подправляйте нужное поддерево).
17. Определите функции (**ПРЕДДЕРЕВО** *a дерево*) и (**ПОСЛЕДЕРЕВО** *a дерево*), которые выделяют в отдельное (упорядоченное) дерево из первоначального дерева все узлы, предшествующие данному элементу *a* и следующие за ним.
- Определите после этого с их помощью функцию (**ОБЪЕДИНИ** *p q*), объединяющую два упорядоченных дерева *p* и *q* в одно общее (упорядоченное) дерево.
18. Что вычисляет следующая рекурсивная функция первого порядка:

```
(defun бесполезная (l)
  (cond
    ((null l) nil)
    (t (cons (car l)
              (бесполезная (бесполезная (cdr l)))))))
```



*Все обобщения опасны,  
и это тоже.*

*А. Дюма мл.*

## 3.4 ФУНКЦИИ БОЛЕЕ ВЫСОКОГО ПОРЯДКА

- Функционал имеет функциональный аргумент
- Функциональное значение функции
- Способы композиции функций
- Функции более высокого порядка
- Литература

### Функционал имеет функциональный аргумент

До сих пор мы рассматривали функции, которые в качестве аргументов получали формы, представляющие собой выражения, являющиеся данными. Значения функций также считались относящимися по типу к данным, т.е. выражениям, трактуемым как данные.



Основываясь на едином представлении данных и программ, функции в качестве аргумента можно указать и функцию, или, другими словами, определение функции, либо представляющий функцию символ. Аргумент, значением которого является функция, называют в функциональном программировании *функциональным аргументом* (*functional argument*), а функцию, имеющую функциональный аргумент – *функционалом* (*functional*).

Различие между понятиями "данные" и "функция" определяется не на основе их структуры, а в зависимости от их использования. Если аргумент используется в функции лишь как объект, участвующий в вычислениях, то мы имеем дело с обычным аргументом, представляющим данные. Если же он используется как средство, определяющее вычисления, т.е. играет в вычислениях, например, роль лямбда-выражения, которое применяется к другим аргументам, то мы

имеем дело с функцией.

Одно и то же выражение может в связи с различными аспектами выступать, с одной стороны, как обычный аргумент, а с другой стороны, как функциональный. Роль и интерпретация выражения зависят от его синтаксической позиции. Приведем пример:

```
_ (car '(lambda (x)
                  (list x))) ; CAR - функция
LAMBDA      ; лямбда-выражение - данные
_ ((lambda (x)
      (list x)) 'car) ; CAR - данные
(CAR)        ; лямбда-выражение - функция
```

Переданное функции CAR лямбда-выражение не является функциональным аргументом, и CAR не становится функционалом. То же самое лямбда-выражение, стоящее в позиции функции, уже интерпретируется как функция в то время, как CAR интерпретируется как данные. Функциональный аргумент и функционал являются некоторым обобщением простого понятия функции: функциональным аргументом может быть любой подходящий объект, который используется в теле функционала в позиции функции и в роли функции.

Далее мы будем использовать понятия функции, вызова функции и значения функции в следующем смысле:

- 1) Функция сама есть изображение вычислений или определение.
- 2) Вызов функции есть применение этого изображения.
- 3) Значение функции есть результат такого применения.

Функция является функционалом, если в качестве ее аргумента используется лисповский объект типа (1), который интерпретируется как функция (2) в теле функционала. Таким функциональным объектом (functional object) может быть

- 1) символное имя, представляющее определение функции (системная функция или функция, определенная пользователем),
- 2) безымянное лямбда-выражение,
- 3) так называемое замыкание (рассматриваемое позже).

Фактический параметр для функционального аргумента функционала задается в виде формы, значением которой будет объект, который можно интерпретировать как функцию. Приведем примеры:

```
... '(lambda (x) (list x)) ...
... (list 'lambda '(x) (list 'list 'x)) ...
... (first '(car cdr cons)) ...
```

### Функциональное значение функции

Аргументом функции может быть функция, однако, функция может быть и результатом. Такие функции называют функциями с функциональным значением (function valued). Функционал также может быть с функциональным значением. Вызов такого функционала возвращает в качестве результата новую функцию, в построении которой, возможно, используются функции, получаемые функционалом в качестве аргументов.



Такие функционалы мы будем называть функционалами с функциональным значением.

Вызов функции с функциональным значением или другая форма с функциональным значением может в вызове функции находиться в двух различных позициях:

- 1) функционального аргумента в вызове функционала;
- 2) на месте имени функции в вызове функции (или функционала, или функции с функциональным значением).

В Коммон Лиспе предполагается, что в вызове функции на месте имени функции находится символ, определенный с помощью формы DEFUN как имя функции. Переданный в качестве параметра функциональный объект можно использовать лишь через явный вызов применяющих функционалов (FUNCALL, APPLY), которые мы рассмотрим подробнее в будущем. В некоторых других системах такого ограничения нет и в вызове функции на месте имени функции может быть любая форма, имеющая функциональное значение. Когда именем в вызове является атом без функциональной связи, то в качестве функционального объекта берется значение этого имени (SYMBOL-VALUE).

### Способы композиции функций

Все типы функций, которые до сих пор встречались, могут быть использованы в определениях (лямбда-выражениях) в следующих позициях:



1. Обыкновенный вызов:  
*(defun f ...  
... (g ...) ...)*

2. Рекурсивный вызов:  
*(defun f ...  
... (f ...) ...)*

3. Вложенный рекурсивный вызов:  
*(defun f ...  
... (f ... (f ...) ...) ...)*

4. Функциональный аргумент:  
*(defun f (... g ...)  
... (apply g ...) ...)*

Аргументами функций были данные, выражения, значением которых являются данные или, как в случае функций более высокого порядка, другие функции.

Объединяя использование рекурсии и функционалов, остановимся на еще одном способе использования, когда функция принимает саму себя в качестве функци-

онального аргумента:

### 5. Рекурсивный функциональный аргумент:

*(defun f (... f ...)  
... (apply f ... f ...) ...)*

Получающий себя в качестве аргумента функционал называют применяемым к самому себе или *автоаппликативной* (*self-applicative*, *auto-applicative*) функцией. Соответственно функцию, возвращающую саму себя, называют *авторепликативной* (*self-replicative*, *auto-replicative*). Часто говорят и о самовоспроизводящихся (*self-reproducing*) функциях.

Автоаппликативные и авторепликативные функции образуют класс *автофункций* (*auto-function*). Автофункции на самом деле не получают в качестве параметра и не возвращают в качестве результата буквально самих себя, а лишь используют или копируют себя.

## Функции более высокого порядка



Представленные ранее функционалы и функции с функциональным значением (типы 4 и 5) в отличие от обычных функций (1, 2, 3), получающих в качестве аргументов и возвращающих в качестве значения данные или выражения, значением которых являются данные, называются функциями более высокого порядка (*higher order*)<sup>1)</sup>. Обычные функции независимо от того, рекурсивные они или нет, являются функциями первого порядка.

Функции более высокого порядка открывают новые возможности для программистов, позволяя сложные вычисления записывать более коротко. Многие вещи, которые в традиционных языках или при традиционном подходе трудно или практически невозможно запрограммировать, можно определить яснее и короче, используя различные типы функционалов и функции с

---

<sup>1)</sup> Следует различать понятия порядка функции и порядка рекурсии (см. стр. 232). – Прим. перев.

**функциональным значением.**

Передача функции в качестве параметра другой функции и создание функции с помощью специальных форм составляет основу для новых технологий программирования, таких, например, как *программирование, управляемое данными* (data driven programming), и *объектно-ориентированное программирование* (object programming). Функции более высокого порядка тесно связаны с замыканиями и макросами, а также с *частичными* (partial evaluation) и *отложенными вычислениями* (lazy evaluation). К этим механизмам мы вернемся позднее.

Функции более высокого порядка тесно связаны с абстрактными понятиями и отображениями. Теоретическое и практическое владение ими предполагает умение учитывать типы функций, т.е. учитывать типы принимаемых ими аргументов и возвращаемых ими значений.

### Литература

- 
1. Brady J. *The Theory of Computer Science. A Programming Approach*. Chapman et Hall, London, 1977.
  2. Henderson P. *Functional Programming – Application and Implementation*. Prentice-Hall, London, 1980.
  3. Milner R. *Theory of Type Polymorphism in Programming*. JCSS, No. 17, 1978, pp. 348–375.



*Истинное знание состоит в  
знании того, что мы знаем то,  
что знаем, и в знании того,  
что мы не знаем то, чего не  
знаем.*

*Конфуций*

### 3.5 ПРИМЕНЯЮЩИЕ ФУНКЦИОНАЛЫ

- **APPLY** применяет функцию к списку аргументов
- **FUNCALL** вызывает функцию с аргументами
- Упражнения

Одним из основных типов функционалов являются функции, которые позволяют вызывать другие функции, иными словами, применять функциональный аргумент к его параметрам. Такие функционалы называют *применяющими* или *аппликативными функционалами* (*applicative functional*).

Применяющие функционалы родственны универсальной функции Лиспа **EVAL**. В то время как **EVAL** вычисляет значение произвольного выражения (формы), применяющий функционал вычисляет значение вызова некоторой функции. Интерпретатор Лиспа **EVAL** и на самом деле вызывает применяющий функционал **APPLY** при вычислении вызова, а **APPLY** в свою очередь вызывает **EVAL** при вычислении значения других форм.

Применяющие функционалы дают возможность интерпретировать и преобразовывать данные в программу и применять ее в вычислениях. Ниже мы рассмотрим применяющие функционалы интерпретатора Лиспа **APPLY** и **FUNCALL**, которые в дальнейшем будут использованы в программировании, управляемом данными.

## APPLY применяет функцию к списку аргументов



APPLY является (в своей исходной форме) функцией двух аргументов, из которых первый аргумент представляет собой функцию, которая применяется к элементам списка, составляющим второй аргумент функции APPLY:

(APPLY *fn* *список*)

$\Leftrightarrow$   
(*fn* 'x1 'x2 ... 'x*N*)

где *список* = (x1 x2 ... x*N*)

```

5 (apply '+ '(2 3))
5 (apply 'cons '(что (пожелаете))
5 (ЧТО ПОЖЕЛАЕТЕ)
5 (setq f '+)
5
5 (apply f '(2 3))
5 (apply 'eval '(+ 2 3))
5 (apply 'apply '(+ (2 3)))
5 ; APPLY применяет себя
5 (apply '(lambda (x y)(+ x y)) '(2 3))
5

```

Использование APPLY дает большую гибкость по сравнению с прямым вызовом функции: с помощью одной и той же функции APPLY можно в зависимости от функционального аргумента осуществлять различные вычисления.

## FUNCALL вызывает функцию с аргументами

Функционал FUNCALL по своему действию аналогичен APPLY, но аргументы для вызываемой функции он принимает не списком, а по отдельности:

**(FUNCALL  $f_n x_1 x_2 \dots x_N$ )**  
 $\Leftrightarrow$   
**( $f_n x_1 x_2 \dots x_N$ )**

Приведем пример:

```
(funcall '+ 2 3)
5
```

FUNCALL и APPLY позволяют задавать вычисления (функцию) произвольной формой, например, как в вызове функции, или символом, значением которого является функциональный объект. В обычном вызове функции можно использовать лишь символ, связанный DEFUN с лямбда-выражением. Интерпретация имени зависит от синтаксической позиции.

```
(setq сложение '+)
+
(funcall сложение 2 3)
5
(funcall (car '(+ - * /)) 2 3)
5
```

Таким образом появляется возможность использовать **синонимы** имени функции. С другой стороны, имя функции можно использовать как обычную переменную, например для хранения другой функции (имени или лямбда-выражения), и эти два смысла (значение и определение) не будут мешать друг другу:

```
(setq cons '+)
+
(funcall cons 2 3)
5
(cons 2 3)
(2 . 3)
```

Поскольку первый аргумент функционала FUNCALL вычисляется по обычным правилам, то на его

месте должно стоять выражение, значением которого будет функциональный объект. Функциональным аргументом может быть только "настоящая" функция. Такие специальные формы, как QUOTE и SETQ, либо рассматриваемые позже макросы для этого не подходят, даже если бы их значением был функциональный объект.

### Упражнения

1. Вычислите значения следующих вызовов:



- a) (apply 'list '(a b))
- b) (funcall 'list '(a b))
- c) (funcall 'apply 'list '(a b))
- d) (funcall 'list 'apply '(a b))
- e) (funcall 'funcall 'funcall 'list 'list '(a b))

2. Определите FUNCALL через функционал APPLY.



*Я люблю мудрость больше,  
чем мудрость любит меня.*

*Лорд Байрон*

### 3.6 ОТОБРАЖАЮЩИЕ ФУНКЦИОНАЛЫ

- Отображающие функции повторяют применение функции
- MAPCAR повторяет вычисление функции на элементах списка
- MAPLIST повторяет вычисление на хвостовых частях списка
- MAPCAN и MAPCON объединяют результаты
- MAPC и MAPL теряют результаты
- Композиция функционалов
- Итоговая таблица отображающих функций
- Упражнения

**Отображающие функции повторяют применение функции**



Важный класс функционалов в практическом программировании на языке Лисп образуют *отображающие функции* или *MAP-функции* (mapping function). MAP-функционалы являются функциями, которые некоторым образом отображают (пар) список (последовательность) в новую последовательность или порождают побочный эффект, связанный с этой последовательностью. Имена MAP-функций начинаются на *MAP*, и их вызов имеет вид

**(MAPx fn l1 l2 ... lN)**

Здесь *l1 ... lN* – списки, а *fn* – функция от *N* аргументов. Как правило, MAP-функция применяется к одному аргументу-списку, т.е. *fn* является функцией от одного аргумента:

**(MAPX fn список)**

Существуют два основных типа MAP-функций. Одни из них применяют функциональный аргумент *fn* таким образом, что его аргументами будут последовательные CAR аргумента-списка (иными словами, *fn* применяется к элементам списка). Другие применяют функциональный аргумент к последовательным CDR списка. Результатом этих повторяющихся вычислений будет список, состоящий из результатов последовательных применений функции. Кроме того, функции отличаются друг от друга способом формирования результата. Во всех случаях число аргументов-списков должно совпадать с числом аргументов применяемой для вычислений функции.

Рассмотрим основные типы MAP-функций.

**MAPCAR повторяет вычисление функции на элементах списка**


Значение функции MAPCAR вычисляется путем применения функции *fn* к последовательным элементам *xi* списка, являющегося вторым аргументом MAPCAR. Например, в случае одного списка получается (с небольшими ограничениями) следующее соответствие:

**(MAPCAR fn '(x1 x2 ... xN))**

↔

**(LIST (fn 'x1) (fn x2) ... (fn 'xN))**

В качестве значения функционала возвращается список, построенный из результатов вызовов функционального аргумента MAPCAR. Приведем пример:

```
(setq x '(a b c))
(A B C)
(mapcar 'atom x)
(T T T)
```

```

(mapcar '(lambda (y) (list (print y))) x)
A
B
C
((A) (B) (C))
(defun парал (x) (list x 1))
ПАРАЛ
(mapcar 'парал x)
((A 1) (B 1) (C 1))

```

Использованные выше функции (ATOM, ПАРАЛ) были функциями с одним аргументом, т.е. каждый раз они имеют дело с одним элементом. Если бы функции MAPCAR был передан функциональный аргумент *fn* с большим количеством параметров, то и у MAP-функционала было бы соответствующее количество параметров-списков, которые обрабатываются параллельно.

В приведенном ниже вызове, например, функциональным аргументом MAPCAR является функция CONS, которая требует двух аргументов. Вызов MAPCAR строит список пар из соответствующих элементов двух просматриваемых списков:

```

(mapcar 'cons x '(1 2 3))
((A . 1) (B . 2) (C . 3))
(mapcar '(lambda (u v) (list v u))
x '(1 2 3))
((1 A) (2 B) (3 C))

```

MAPCAR можно определить через используемый в Лиспе применяющий функционал. Для случая функционального аргумента с одним параметром функция MAPCAR выглядела бы следующим образом:

```

(defun mapcar1 (fn l)
  (if (null l) nil
      (cons (funcall fn (car l))
            (mapcar1 fn (cdr l)))))

```

Далее мы увидим, что это определение эквивалентно MAPCAR не во всех случаях.

### **MAPLIST повторяет вычисление на хвостовых частях списка**

MAPLIST действует подобно MAPCAR, но действия осуществляются не над элементами списка, а над последовательными CDR<sup>1)</sup> этого списка:

```
(maplist '(lambda (y) y) '(a b c))
((A B C) (B C) (C))
(maplist 'reverse '(a b c))
((C B A) (C B) (C))
(defun входит-ли (x)
  (if (member (car x) (cdr x)) 1 0))
ВХОДИТ-ЛИ
(setq последовательность '(В Х О Д И Т
                             П И З Н А К В Х В О С Т))
(В Х О Д И Т П И З Н А К В Х В О С Т)
(maplist 'входит-ли последовательность)
(1 1 1 0 1 1 0 0 0 0 0 0 1 0 0 0 0 0)
```



Функционалы MAPCAR и MAPLIST используются для программирования циклов специального вида и в определении других функций, поскольку с их помощью можно сократить запись повторяющихся вычислений.

### **MAPCAN и MAPCON объединяют результаты**

Функции MAPCAN и MAPCON являются аналогами функций MAPCAR и MAPLIST. Отличие состоит в том,

  
что MAPCAN и MAPCON не строят, используя LIST, новый список из результатов, а объединяют списки, являющиеся результатами, в один список, используя структуроразрушающую псевдофункцию NCONC. При использовании этих псевдофункций нужно соблюдать

<sup>1)</sup> Начиная с самого исходного списка (!). – Прим. ред.

осторожность. MAPCAN (с небольшими ограничениями) можно представить в виде вызова NCONC следующим образом (сравните с представлением MAPCAR через вызов LIST):

$$\begin{array}{l} (\text{MAPCAN } fn \ '(x1\ x2\ \dots\ xN)) \\ \Downarrow \\ (\text{NCONC } (fn \ 'x1) (fn \ 'x2) (fn \ 'x3) \dots (fn \ 'xN)) \end{array}$$

```
(марсан 'list '(a b c))
(А В С)
(марсон 'list '(a b c))
(А В С В С С)
(марсан 'cons '(a b c) '(1 2 3))
(А В С)
(марсон 'cons '(a b c) '(1 2 3))
((А В С) 1 2 3 (В С) 2 3 (С) 3)
```

Для того чтобы функции MAPCAN и MAPCON могли применить NCONC, их функциональные аргументы должны возвращать в качестве значения списки.

Длина списка, являющегося результатом объединяющих функционалов, не обязательно должна совпадать с длиной списка аргументов, поскольку если значением вызова функционального аргумента является NIL, то в объединенном списке его не видно. В функциях MAPCAR и MAPLIST значение NIL всегда остается в списке результатов:

```
(маркар 'numberp '(1 2 мимо 3))
(Т Т NIL Т)
(марсан '(lambda (x)
  (if (numberp x) '(t) nil))
 '(1 2 мимо 3))
(Т Т Т)
```

Исходя из этого свойства, объединяющие функционалы можно использовать как фильтры. В общем случае под фильтром понимается функция, которая оставляет или

удаляет элементы, удовлетворяющие заданному условию.



Например, функцией ПОСЛЕДНИЙ можно отсеять из списка буквы, оставив по одной в том порядке, в котором они входили в список в последний раз<sup>1)</sup>:

```
_ (defun последний (x)
  (if (eql (входит-ли x) 0)
    (list (car x))
    nil))
```

**ПОСЛЕДНИЙ**  
**последовательность**  
**(В Х О Д И Т П И З Н А К В Х В О С Т)**  
**(шарсон 'последний последовательность)**  
**(Д П И З Н А К Х В О С Т)**

MAPCAN и MAPCON можно определить через MAPCAR и MAPLIST следующим образом:

$$\begin{aligned} & (\text{MAPCAN } fn \ x_1 \ x_2 \dots x_N) \\ \Leftrightarrow & (\text{APPLY } 'NCONC (\text{MAPCAR } fn \ x_1 \ x_2 \dots x_N)) \\ & (\text{MAPCON } fn \ x_1 \ x_2 \dots x_N) \\ \Leftrightarrow & (\text{APPLY } 'NCONC (\text{MAPLIST } fn \ x_1 \ x_2 \dots x_N)) \end{aligned}$$

### MAPC и MAPL теряют результаты

Функции MAPC и MAPL также аналогичны функциям MAPCAR и MAPLIST, но отличаются от них и от функций MAPCAN и MAPCON тем, что не собирают и не объединяют результаты. Возникающие результаты просто теряются. В качестве значения возвращается значение второго аргумента функции:

**(MAPC** *fn* **список)**

<sup>1)</sup> Функция ВХОДИТ-ЛИ определена на стр. 252. – Прим. ред.

**(PROG2 (MAPCAR *fn* *snusok*) *snusok*)**

### **(MAPL fn cnucok)**

**(PROG2 (MAPLIST fn cnucok) cnucok)**

Псевдофункционалы MAPC и MAPL прежде всего используют для получения побочного эффекта:

```
(mapc '(lambda (u v) (set u v))
       '(a b c)
       '(1 2 3))
(A B C)
b
 $\frac{b}{2}$ 
```

**Замечание:** В более старых диалектах языка Лисп функция MAPL носила имя MAP. В Коммон Лиспе имя изменено, поскольку название MAP зарезервировано для рассматриваемой ниже более универсальной функции, которая используется и в литературе, посвященной функциональному программированию.

## Композиция функционалов

Вызовы функционалов можно объединять в более сложные структуры таким же образом, как и вызовы функций, а их композицию можно использовать в определениях новых функций.

Например, прямое произведение двух числовых множеств можно определить через два вложенных цикла, которые можно выразить с помощью композиции двух вложенных вызовов функционала **MAPCAR**:

```
(декартово '(a b c) '(1 2 3))
((A 1) (A 2) (A 3) (B 1) (B 2) (B 3) (C 1)
 (C 2) (C 3))
```

Функционалы могут, подобно функциям, комбинироваться с использованием различных видов рекурсии. Если вызов функционала встречается в его собственном определении, то функционал будет рекурсивным. Например, приведенное нами ранее определение MAPCAR рекурсивно.

В следующем определении декартова произведения внешний цикл выражен через рекурсию, а внутренний – через функционал MAPLIST:

```
(defun декартово (x y)
  (cond
    ((null x) nil)
    (t (append
          (maplist '(lambda (u)
                     (list (car x)
                           (car u)))
          y)
          (декартово (cdr x) y))))))
```

Определять и использовать функционалы нужно очень внимательно и следить за тем, чтобы число аргументов и их типы на различных уровнях были согласованы с типами функций, принимающих их в качестве параметра и возвращающих их в качестве результата.

### Итоговая таблица отображающих функций

**Σ** В качестве итога и с целью повторения приведем в виде таблицы МАР-функции, предназначенные для аргументов, записанных в виде списков<sup>1)</sup>.

У всех функций первый аргумент – функциональный. Если это функция более чем от одного аргумента, то после функционального аргумента

<sup>1)</sup> Учтите примечание на с. 252. – Прим. ред.

следует соответствующее количество аргументов-списков. Если списки различаются по длине, то количество повторений определяется длиной наиболее короткого списка.

Форма вызова	Применяется к	Значение
(MAPCAR <i>fn</i> &REST <i>li</i> )	голова	список из результатов функцией CONS
(MAPLIST <i>fn</i> &REST <i>li</i> )	хвост	
(MAPCAN <i>fn</i> &REST <i>li</i> )	голова	объединение результатов функцией NCONC
(MAPCON <i>fn</i> &REST <i>li</i> )	хвост	
(MAPC <i>fn</i> &REST <i>li</i> )	голова	первый аргумент-список
(MAPL <i>fn</i> &REST <i>li</i> )	хвост	

Мы рассмотрели традиционные функционалы, определенные для списков. Кроме них Коммон Лисп содержит еще целый набор функционалов, которые определены для более общего, чем списки типа данных, называемого *последовательностями* (sequence). Кроме списков последовательностями являются также векторы и строки. Функционалы над последовательностями можно применять ко всем типам последовательностей. К последовательностям в Коммон Лиспе мы вернемся подробнее в связи с рассмотрением типов данных.

Отображающие функционалы не усиливают вычислительной мощности Лиспа, но, без сомнения, являются удобными изобразительными средствами. Во многих случаях с их помощью можно существенно сократить запись по сравнению с повторяющимися вычислениями, выраженнымими рекурсией или итерацией.

### Упражнения

1. Определите функционал (**MAPLIST** *fn* *список*) для одного списочного аргумента.

- 2 Определите функционал (**APL-APPLY**  $f\ x$ ), который применяет каждую функцию  $f_i$  списка

$$f = (f_1 \ f_2 \ \dots \ f_N)$$

к соответствующему элементу  $x_i$  списка

$$x = (x_1 \ x_2 \ \dots \ x_N)$$

и возвращает список, сформированный из результатов.



3. Определите функциональный предикат (**КАЖДЫЙ** *пред список*), который истинен в том и только в том случае, когда, являющийся функциональным аргументом предикат *пред* истинен для всех элементов списка *список*.

Определите функциональный предикат (**НЕКОТОРЫЙ** *пред список*), который истинен, когда предикат истинен хотя бы для одного элемента списка.

4. Определите фильтры

a) (**УДАЛИТЬ-ЕСЛИ** *пред список*)

b) (**УДАЛИТЬ-ЕСЛИ-НЕ** *пред список*)

удаляющие из списка элементы, которые a) обладают или b) не обладают свойством, наличие которого проверяет предикат *пред*.



*Нет ничего более постоянно-го, чем временное.*

*О. д'Юрфе*

## 3.7 ЗАМЫКАНИЯ

- **FUNCTION** блокирует вычисление функции
- Замыкание – это функция и контекст ее определения
- Связи свободных переменных замыкаются
- Замыкания позволяют осуществлять частичное вычисление
- Генератор порождает последовательные значения
- Контекст вычисления функционального аргумента
- Литература
- Упражнения

### **FUNCTION** блокирует вычисление функции

С точки зрения эффективности работы транслятора было бы правильно, а с точки зрения вычисления функционала в некоторых случаях и необходимо, чтобы интерпретатор уже на этапе вызова функционала мог отличить функциональный аргумент от обычного. Функциональный аргумент можно пометить с помощью специальной, предотвращающей вычисления формы **FUNCTION**:

### (**FUNCTION** функция)

Например:

```
(function (lambda (x) (list x y)))
(function car)
```

В отличие от обычной блокировки вычислений с помощью **QUOTE** форму **FUNCTION** называют *функциональная блокировка* (*function quote*). Функциональ-

ную блокировку так же, как и QUOTE, можно записывать сокращенно:

$$\begin{array}{lcl} 'x & \Leftrightarrow & (\text{QUOTE } x) \\ #'f & \Leftrightarrow & (\text{FUNCTION } f) \end{array}$$

Если нужно передать функции данные в том виде, как они записаны, то используется обычная форма QUOTE. QUOTE достаточно и для передачи имени функции или лямбда-выражения, если в нем не используются свободные переменные. Работа со свободными переменными оказывается более сложной, чем с параметрами, поскольку значения свободных переменных зависят от контекста вычислений.

В системных функциях свободные переменные не используются, поэтому для функции +, например, следующие формы имеют одинаковое значение:

```
(+2 3)
(funcall '+ 2 3)
(funcall (function +) 2 3)
(funcall #''+ 2 3)
((lambda (x y) (+ x y)) 2 3)
(funcall (function (lambda (x y) (+ x y))) 2 3)
```

**Замыкание – это функция и контекст ее определения**

Сформированный на время вычисления функции вычислительный контекст после окончания ее вычисле-

 ния пропадает, и на него невозможно позже сослаться или вернуться к нему. Часто бывает полезным и необходимым, чтобы функция для продолжения вычислений могла запомнить связи и состояние более раннего контекста. Это

достигается с помощью таких функциональных объектов, в которых вместе с самим описанием вычислений сохраняется контекст момента определения функционального объекта, защищенный от более позднего контекста вызова. Ранее в Лиспе для пары, состоящей из функции (лямбда-выражения) и контекста, использовалось понятие *фунарг* (funarg). Позже ее стали

называть **замыканием** или **лексическим замыканием** (*lexical closure*). Замыкание, как и имя функции или лямбда-выражение, можно использовать в качестве функционального аргумента.

### Связи свободных переменных замыкаются

В Коммон Лиспе замыкание создается формой FUNCTION. Например:

```
(function (lambda (x) (+ x y))
  (LEXICAL-CLOSURE ...))
```

В замыкание из контекста определения функции включаются лишь связи свободных переменных функции (в приведенном примере глобальное значение *y*). Значения формальных параметров, в примере значение *x*, сохранять нет необходимости, поскольку они получают новые значения в момент применения такого замыкания. Если в замыкаемой функции нет свободных переменных, то форма FUNCTION ни чем не отличается от формы QUOTE.

Замыкание можно сохранять также, как любой лисповский объект, присваивая его какой-нибудь переменной. Сохраненное таким образом замыкание можно снова использовать и применять к аргументам так же, как функцию или лямбда-выражение, с тем лишь отличием, что значения свободных переменных замкнутой функции определяются в соответствии со связями из статического контекста момента создания замыкания, а не из динамического контекста момента вызова. Пример замыкания:

```
(setq y 10)
10
(setq s (function (lambda (x) (+ x y)))
  (LEXICAL-CLOSURE ...))
```

Здесь переменной *S* присваивается в качестве значения замыкание, в котором переменная *y* свободна. Значение переменной *y* сохраняется в замыкании, и поэтому возможные более поздние присваивания на это значе-

ние не повлияют. Теперь замыкание можно использовать как функциональный аргумент вместо имени функции или лямбда-выражения. Его можно, например, вызвать с помощью формы FUNCALL:

```
(funcall s 2) ; в замыкании у = 10
12
(setq y 100)
100
(funcall s 2) ; значения связей в
12             ; замыкании сохраняются
```

С помощью функции с функциональным значением можно создавать параметризованные замыкания, например, следующим образом:

```
(defun приплюсовывать (x)
  (function          ; замыкание в качестве
    (lambda (y)(+ x y))) ; значения
ПРИПЛЮСОВЫВАТЬ
  (setq плюс3 (приплюсовывать 3))
  (LEXICAL-CLOSURE ...)
  (funcall плюс3 5))
8
```

В различных Лисп-системах для создания замыканий используются различные механизмы. Например, в Франц Лиспе замыкание создается с помощью специальной формы CLOSURE, в которой можно перечислить переменные, значения которых будут сохраняться в составе замыкания.

Связи свободных переменных замыкания остаются в силе до следующего запуска замыкания, и этим переменным можно даже присваивать новые значения. В паре замыканий следующего примера переменная X свободна для каждого из приводимых замыканий и X можно присвоить значение вызовом функции ДВЕФУН.

Замыкание отличается от функции тем, что оно дает возможность сослаться на свободные переменные момента определения и избежать возможной путаницы

```

_(defun двефун (x)
  (cons (function      ; прибавляет x
             (lambda (y) (+ x y)))
        (function      ; присваивает
             (lambda (y) ; значение x
                 (setq x y)))))

ДВЕФУН
_(setq пара (двефун 10))
                  ; x присваивается 10
((LEXICAL-CLOSURE ...) ; в качестве
                         ; значения пара замыканий
(funcall (car пара) 2) ; 1-е замыкание
12                   ; прибавляет 10
(funcall (cdr пара) 100) ; 2-е
                         ; присваивает
100                  ; x=100
(funcall (car пара) 2) ; 1-е замыкание
102                  ; прибавляет 100

```

с переменными, имеющими такое же имя и могущими оказаться в контексте вызова. Форма FUNCTION переносит вычисление функции в другой контекст.

Происхождение термина "лексическое замыкание" объясняется тем, что замыкание запоминает статические (т.е. лексические) связи свободных переменных и не позволяет на них влиять внешним по отношению к данному замыканию функциям. Переменные замыкания невозможно использовать извне замыкания, и изменения их значений извне не видны (visibility).

### **Замыкания позволяют осуществлять частичное вычисление**

Замыкание можно трактовать как функцию, вычисление которой осуществлено лишь *частично* (partial evaluation), окончательное вычисление функции отложено на момент ее вызова. При создании замыкания из определения функции формируется частично вычисленный контекстный объект и, чтобы осуществить окончательные вычисления, ему следует передать значения недостающих параметров.

## Генератор порождает последовательные значения



Замыкание хорошо подходит для программирования *генераторов*. Под генератором (*generator*) понимают функциональный объект, из которого можно получать новые, отличные от прежних значения. При

каждом вызове генератор порождает значение некоторого типа, которое является в некотором смысле следующим по порядку. Генераторам присуща следующая особенность: значения порождаются только при необходимости (*demand generator*) и формирование следующего значения основано на предыдущем. Приведем пример:

```
_ (defun натуральное-число (x)
  (function (lambda nil
    (setq x (+ x 1))))
НАТУРАЛЬНОЕ-ЧИСЛО
  (setq следующий (натуральное-число 0))
(LEXICAL-CLOSURE ...)
```

Генератор чисел СЛЕДУЮЩИЙ порождает при каждом вызове целое число на единицу большее, чем порожденное в предыдущем вызове.

```
1 (funcall следующий)
2 (funcall следующий)
```

Можно создать различные *экземпляры* (*instance*) генератора, задав при вызове новое начальное значение и дав экземпляру имя:

```
(setq источник (натуральное-число 10))
(LEXICAL-CLOSURE ...)
(funcall источник)
11 (funcall следующий)
3
```

С помощью генераторов потенциально можно работать с неограниченными последовательностями и структурами, например с рекурсивными структурами данных. Такие структуры и последовательности фактически вычисляются лишь до необходимой в текущий момент глубины. Возникающие таким образом объекты данных называют *потоком* (*stream*). Примерами потоковых объектов являются рассмотренные нами ранее потоки ввода и вывода, в результате функционирования которых возникают файлы, окна и другие подобные объекты.

Понятие потока можно использовать и в самом языке как принцип осуществления вычислений. Например, стандартный интерпретатор Лиспа предварительно вычисляет формы, являющиеся аргументами вызова, еще до вычисления тела независимо от того, используются ли они фактически в теле. Однако интерпретатор можно было бы определить и так, чтобы каждая форма вычислялась лишь тогда, когда ее значение фактически требуется в ходе вычисления следующей формы.

Последний способ вычислений называют *отложенными* (*suspended*) или *ленивыми* вычислениями (*lazy evaluation*). Применяя принцип отложенных вычислений, можно решить многие задачи, связанные с неограниченными по длине списками или структурами. Этот же принцип позволяет осуществлять *параллельные* вычисления.

**Контекст вычисления функционального аргумента**  
С использованием функционалов и функций с функциональным значением связана так называемая *фунарг-проблема*<sup>1)</sup>. Мы рассмотрим эту проблему сначала на примере.

Ранее мы уже определили с помощью применяющего функционала функционал **MAPCAR** (упрощенный вариант) в следующем виде:

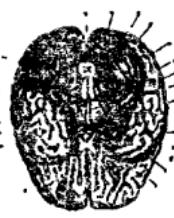
---

<sup>1)</sup> Фунарг – аббревиатура от слов функциональный аргумент.–  
Прим. ред.

```
(defun mapcar1 (fn l)
  (cond ((null l) nil)
        (t (cons (funcall fn (car l))
                  (mapcar1 fn (cdr l))))))
```

Используем это определение в следующей функции ПРОБА:

```
(defun проба (l)
  (mapcar1 '(lambda (x) (cons x l))
            '(a b c)))
```



У переменной  $L$  есть свободное вхождение в лямбда-выражение, являющееся функциональным аргументом в функции ПРОБА, и связанные вхождения в функциях ПРОБА и MAPCAR1. Не очевидно, какое значение  $L$  представляет каждое вхождение. Если мы вызовем, например, функцию ПРОБА с аргументом NIL, то статическим значением  $L$  функции ПРОБА будет NIL. Вызывая функцию MAPCAR, мы получим следующие значения для ее функционального аргумента FN и списочного аргумента L:

$$\begin{aligned} FN &= (\text{lambda} (x) (\text{cons} x l)) \\ L &= (A \ B \ C) \end{aligned}$$

Суть проблемы связана со свободной переменной  $L$  из лямбда-выражения, являющегося значением параметра FN. Определяется ли ее значение в момент применения лямбда-выражения статически в соответствии со значением параметра  $L=NIL$  функции ПРОБА либо динамически в соответствии со значением параметра  $L$  функции MAPCAR1? В последнем случае значениями  $L$  на различных уровнях рекурсии были бы список (A B C) и его последовательные хвосты.

Эти интерпретации дали бы два различных ответа:

- 1) ((A) (B) (C)) ; статическая интерпретация  
 2) ((A A B C) (B B C) (C C)) ; динамическая интерпретация

Различные результаты получаются из-за того, что контекст, в котором вычисляется функциональный аргумент, зависит от контекста момента вызова, и он может меняться от вызова к вызову. Проблемы могут возникнуть, когда в контекстах на момент определения и на момент вызова используются одинаковые имена, для переменных в действительности представляющих различные объекты. Значения свободных переменных функционального аргумента могут неожиданно измениться на некотором этапе вычислений. Чтобы избежать конфликта имен и возникновения этой проблемы, программист должен был бы при применении функционала знать имена его внутренних переменных. Однако такое решение было бы лишено изящества и ухудшало бы прозрачность программирования.

Рассмотренную выше проблему определения контекста при вычислении свободных переменных функционального аргумента в Лиспсе называют *функциональной проблемой*. Эта проблема получает разрешение путем использования в качестве функционального аргумента замыкания, в котором можно зафиксировать значения свободных переменных из контекста момента определения. При этом конфликты по именам будут исключены.

Например, если вместо QUOTE указанное выше лямбда-выражение будет передано как замыкание с помощью формы FUNCTION, то в функции ПРОБА значение переменной L из функционального аргумента MAPCAR1 будет всегда совпадать со значением статической переменной L функции ПРОБА:

```
(defun проба (l)
  (mapcar1 (function
    (lambda (x) (cons x l))) '(a b c))
ПРОБА
(проба nil)
((A) (B) (C))
```

Если бы функциональный аргумент был задан формой QUOTE, то в Коммои Лиспе мы получили бы ошибку с сообщением о том, что у L нет связи.

До того как фунарг-проблема прояснилась до конца, она успела создать много неприятностей как создателям Лисп-систем, так и программистам. Сначала казалось, что это – проявление ошибки в интерпретаторе Лиспа. После того как никакой ошибки не было обнаружено и положение так и оставалось неясным, в различных реализациях стали ограничивать использование функционалов, не рекомендуя использовать их в программах. В основе возникших проблем лежало то обстоятельство, что теоретические положения лямбда-вычисления не были в достаточной степени учтены при реализации языка: в самом языке Лисп или в формализме лямбда-исчисления никаких ошибок нет.

Вся совокупность вопросов получила окончательное разрешение после того, как Лэндин (Landin 1964) формализовал программные аспекты лямбда-исчисления, специально учитывая факторы, связанные с функционалами и контекстом вычислений. Построенный Лэндином формализм основывался на четырех структурах данных в виде списков: стек (stack), контекст (environment), управление (control) и дамп (dump), из которых происходит и имя его формализма – SECD-машина. Для решения проблемы контекста он предложил понятие замыкания, которое с тех пор занимает важное место в программировании, особенно в управлении параллельными процессами.

По сути дела, существуют два вида замыканий в зависимости от того, имеем ли мы дело с функциональным аргументом, передаваемым функционалу (downward funarg), или с возвращенным в качестве результата вычисления функциональным объектом (upward funarg). Первоначально фунарг-проблема осознавалась только для первого случая. Полученное для этого случая решение оказалось в какой-то мере применимо и ко второму случаю, и таким образом вся проблема получила свое решение.

Фунарг-проблема особенно неприятна для тех систем, в которых интерпретатор использует динами-

ческие вычисления, а отранслированные программы – статические<sup>1)</sup>. В них, использующие функциональный аргумент свободные переменные, работают по-разному в интерпретируемом и транслированном варианте.

Не все Лисп-системы допускают использование замыканий.

## Литература

- 
1. Curry H., Freys R. *Combinatory Logic*. Vol.1, North-Holland, Amsterdam, 1958.
  2. Friedman P., Wise D. *CONS Should not Evaluate its Arguments*. Automata, Languages and Programming, Third International Colloquium, Edinburgh University Press, 1976, pp. 257–284.
  3. Henderson P., Morris J. *A Lazy Evaluator*. ACM SIGPLAN, Third Symposium on Principles of Programming Languages, 1976.
  4. Landin P. *The Mechanical Evaluation of Expressions*. The Computer Journal, Vol. 6, No. 4, 1964, pp. 308–320.
  5. Turner D. *A New Implementation Technique for Applicative Languages*. Software Practice and Engineering, Vol. 9, 1979, pp. 31–49.

## Упражнения

- 
1. Напишите программу генератора, порождающего следующие последовательности:

- a) числа Фибоначчи 0,1,1,2,3,5 ...
- b) (A),(B A),(A B A),(B A B A) ...

<sup>1)</sup> Наиболее распространены, особенно на микромашинах, интерпретаторы языка Лисп, делающие интерактивный режим программирования особенно удобным. Трансляторы, повышающие вычислительную эффективность, применяются лишь на мини-ЭВМ и Лисп-машинах. Однако в версиях Коммон Лиспа для микромашин широко используются заранее отранслированные (на другой, более мощной ЭВМ) функции. Такая возможность обеспечивается высокой степенью унификации интерпретаторов Коммон Лиспа, созданных для разных ЭВМ.– Прим. ред.

*Мир создан для сотворения нового.*

*Ж. Дюамель*

### 3.8 АБСТРАКТНЫЙ ПОДХОД

- Обобщение функций, имеющих одинаковый вид
- Параметризованное определение функций
- Рекурсивные функции с функциональным значением
- Автоаппликация и авторепликация
- Порядок и тип функций
- Проблемы абстрактного подхода
- Литература
- Упражнения

Обыкновенные функции, работающие с определенными структурами данных, мы абстрактно представляли как отображения данных из области определения в область значения функции. Абстракция существенно основывалась на возможности представить отображения (функции) с помощью параметризированной лямда-выражением общей формы. Лямбда-механизм можно с точки зрения такого абстрагирования данных назвать *абстрацией отображения*. Соответствующая техника применима и для определения функций более высокого порядка. Поскольку в них в качестве параметров и значений используются (в том числе) функции, то можно с помощью механизма параметризации абстрагировать сами функции, другими словами, способ вычислений. Параметризация может касаться первого элемента вызова функции, т.е. осуществляемых вычислений, а не только аргументов, как это происходит в абстракции отображения. Такая *абстрация вычислений* качественно отлична от связанной с данными абстракции отображений.

Далее для иллюстрации более абстрактных вычислений мы рассмотрим два способа действий:

1. Определение вычислений обычным образом в виде функционала, получающего функциональный аргумент.
2. Определение вычислений с помощью параметризованной функции с функциональным значением.

В обоих способах можно свободно применять различные виды рекурсии. Способы определения 1 и 2 можно между собой объединить, что приводит к использованию функционалов с функциональным значением.

Для начала в качестве примера мы рассмотрим обобщение с помощью функционалов функций, определения которых с точки зрения абстракции вычислений сходны по строению, но которые довольно различны с точки зрения производимых действий. После этого мы рассмотрим функции с функциональным значением и тесно связанное с ними использование замыканий в абстракции вычислений.

### Обобщение функций, имеющих одинаковый вид

 Мы уже ранее определяли функцию Коммон Лиспа COPY-LIST, которая копировала верхний уровень списка, и функцию APPEND, которая последовательно объединяла два списка. Их определения были следующими:

```
(defun copy-list1 (x)
  (cond ((null x) nil)
        (t (cons (car x)
                  (copy-list1 (cdr x))))))

(defun append1 (x y)
  (cond ((null x) y)
        (t (cons (car x)
                  (append1 (cdr x) y)))))
```

Мы также определяли функцию СОРТИРУЙ, объединяющую два отсортированных списка в один отсортиро-

ванный список<sup>1)</sup>. Далее предположим, что элементами являются числа, в этом случае определения немного сократятся:

```
(defun сортируй (x y)
  (cond ((null x) y)
        (t (вставь (car x)
                    (сортируй (cdr x) y)))))

(defun вставь (a l)
  ;; добавляет элемент A в
  ;; упорядоченный список L
  (cond
    ((null l)(list a))
    ((< a (car l)) ; численное сравнение
     (cons a l))
    (t (cons (car l)
              (вставь a (cdr l))))))
```

Приведем пример:

```
(сортируй '(1 3 5 7) '(4 6 8))
(1 3 4 5 6 7 8)
```

Заметим, что функции COPY-LIST1, APPEND1 и СОРТИРУЙ отличаются лишь числом параметров и функцией, вызываемой в результирующем выражении второй ветви условного предложения приведенных определений. Отразив эти отличия с помощью параметров, мы можем определить следующий обобщенный функционал:

```
(defun обобщение (x y fn)
  (cond ((null x) y)
        (t (funcall fn (car x)
                    (обобщение (cdr x) y fn)))))
```

<sup>1)</sup> В разделе 3.3 эта функция называлась РАССТАВЬ. Упорядоченным должен в ней быть только второй аргумент.— Прим. перев.

Теперь можно очень коротко определить новые варианты функций COPY-LIST1, APPEND1 и СОРТИРУЙ с помощью функционала ОБОБЩЕНИЕ:

```
(defun copy-list2 (x)
  (обобщение x nil (function cons)))
(defun append2 (x y)
  (обобщение x y (function cons)))
(defun сортируй2 (x y)
  (обобщение x y (function вставить)))
```

Такое замещение функций одинакового вида функционалом является обобщением понятия подпрограммы. Если при использовании подпрограмм сходные элементы программы заменяются вызовом подпрограммы, то в случае функционального обобщения определения многих сходных подпрограмм заменяются функционалами. Далее в качестве второго примера мы определим очень полезный с точки зрения абстрактного построения функционал ИНДЕКС, который осуществляет над выражениями  $X = (X_1 \ X_2 \ \dots \ X_N)$  и  $Y$  действия по следующей схеме<sup>1)</sup>:

$$\begin{aligned} &(\text{индекс } x \ y \ fn) \\ &\Leftrightarrow \\ &(fn \ 'x_1 \ (fn \ 'x_2 \ \dots \ (fn \ 'x_N \ y) \dots)) \end{aligned}$$

Получим для функционала ИНДЕКС простое рекурсивное определение:

```
(defun индекс (x y fn)
  (cond ((null x) y)
        (t (funcall fn
                     (car x)
                     (индекс (cdr x) y fn))))))
```

<sup>1)</sup> На самом деле это тот же самый функционал ОБОБЩЕНИЕ.-  
Прим. перев.

```
(индекс '(a b c) '(ядро) (function cons))
(A B C ЯДРО)
(индекс '(1 2 3) 0 (function +))
6
```

Запрограммированное нами ранее в связи с отображающими функционалами декартово произведение можно определить и с помощью функционала ИНДЕКС:

```
(defun декартово (x y)
  (индекс x nil
    (function (lambda (u v)
      (индекс y v
        (function (lambda (p q)
          (cons (list u p) q))))))))
```

```
(декартово '(A B C) '(1 2 3))
((A 1) (A 2) (A 3) (B 1) (B 2) (B 3) (C 1)
 (C 2) (C 3))
```

Функционал ИНДЕКС очень полезен. С его помощью можно, например, вычислить и *множество всех подмножеств множества* (power set):

```
(defun всеподмножества (x)
  (cond ((null x) '(nil))
    (t (индекс
      (всеподмножества (cdr x))
      nil
      (function (lambda (u v)
        (cons (cons (car x) u)
          (cons u v))))))))
```

```
(всеподмножества '(a b c d))
((A B C D) (B C D) (A C D) (C D) (A B D)
 (B D) (A D) (D) (A B C) (B C) (A C) (C) (A
 B) (B) (A) NIL)
```

С помощью функции ВСЕПОДМНОЖЕСТВА можно определять все более сложные вычисления. Например, приведенная ниже функция с рекурсией более высокого порядка вычисляет множество из подмножеств множества всех подмножеств множества:

```
(defun всевсеподмножества (x)
  (union (всеподмножества x)
          (всеподмножества
           (всеподмножества x))))
```

```
(всевсеподмножества '(a))
(NIL ((A)) (NIL) ((A) NIL) (A))
(всевсеподмножества '(a b))
(NIL ((A B)) ((B)) ((A B) (B)) ((A)) ((A B) (A)) ((B) (A)) ((A B) (B)) ((A) (B)) ((A) NIL) ((B) NIL) ((A B) (B) NIL) ((A) (B) NIL) ((A) NIL) ((A B) (A) NIL) ((B) (A) NIL) ((A B) (B) (A)) ((A) (B) (B)) ((A) (B) (A) NIL) ((A) (B) (A) (B)))
```



Большие изобразительные возможности определения функции отражает тот факт, что для множества из трех элементов ответ будет состоять из 263 подмножеств, для записи которых потребуется 1677 атомов. Для вычисления всех подмножеств всех подмножеств множества из четырех элементов у используемой нами вычислительной машины VAX-11/780 не хватило адресного пространства.

Абстракция построения представляет собой специальный образ действий, сокращающий длину записи. Хотя программа становится более короткой и формально более простой, ее понимание все же усложняется из-за высокого уровня абстракции.

### Параметризованное определение функций

Наряду с функционалами для абстрагирования вычислений также подходят функции с функциональным значением. С их помощью в зависимости от параметров можно строить лямбда-выражения, подбирать применя-

емые функции или создать замыкание, соответствующее данному вычислению. Например, следующая функция с функциональным значением КОМПОЗИЦИЯ возвращает в качестве значения замыкание, которое применяет обе функции, бывшие аргументами. КОМПОЗИЦИЯ является одновременно как функцией с функциональным значением, так и функционалом:

```
(defun композиция (f g)
  (function
    (lambda (x)
      (funcall f (funcall g x)))))
```

```
_ (funcall
  (композиция 'car 'cdr) ; замыкание
  '(a b c)) ; x
B
```



Действие функции КОМПОЗИЦИЯ основано на том, что возвращаемое ею замыкание в момент его применения "помнит" заданные в лямбда-выражении свободные переменные функции (здесь CAR и CDR).  
Функция КОМПОЗИЦИЯ является функциональной абстракцией, с помощью которой можно определить функцию, являющуюся композицией двух функций:

```
(defun second1 (x)
  (funcall (композиция 'first 'rest) x))

(defun второй-атом (x)
  (funcall (композиция 'atom 'second1) x))
```

```
B (second1 '(a b c))
T (второй-атом '(a b c))
```

В этом случае композиция столь проста, что практической пользы от такого абстрагирования мало.



В качестве следующего примера рассмотрим функционал с функциональным значением ВЕРТУШКА. У него два функциональных аргумента F и G, которые являются функциями от двух переменных:

```
(defun вертушка (f g)
  (function (lambda (x y)
    (funcall f (funcall g x y)
              (funcall g y x)))))
```

ВЕРТУШКА применяет функцию, являющуюся вторым аргументом, к аргументам X и Y, взятым в прямом и обратном порядке, и к полученным результатам применяет функцию F. Приведем пример:

```
(setq x '(a b))
(A B)
(setq y '(1 2))
(1 2)
(funcall (вертушка 'list 'append) x y)
((A B 1 2) (1 2 A B))
(funcall (вертушка '* '+) 2 3)
25
(funcall (вертушка 'append 'декартово)
          x y)
((A 1) (A 2) (B 1) (B 2) (1 A) (2 A) (1 B)
 (2 B))
```

В последнем вызове мы получили в одном списке оба прямых произведения двух множеств ( $X \times Y$  и  $Y \times X$ ).

Рассмотрим еще функционал с функциональным значением ДВАЖДЫ, который дважды применяет функцию FN к аргументу X:

```
(defun дважды (fn)
  (function (lambda (x) ; fn с одним
             (funcall fn ; аргументом
                      (funcall fn x)))))
```

```
(defun делай-луковицу (ядро)
  ;; дважды применяет LIST
  (funcall (дважды 'list) ядро))
```

```
_ (делай-луковицу 'луковица)
((ЛУКОВИЦА))
_(defun чисть (луковица)
  ;; дважды применяется CAR
  (funcall (дважды 'car) луковица))
ЧИСТЬ
(чисть '((луковица)))
ЛУКОВИЦА
```

Лямбда-выражение, являющееся телом функционала с функциональным значением ДВАЖДЫ, получит фактический аргумент X как только замыкание будет вызвано с фактическим аргументом. Свободная переменная FN лямбда-выражения сохраняет значение заданной в момент определения функции (CAR или LIST), которая применяется дважды в момент вызова.

Из вызовов функций с функциональным значением можно строить иерархические структуры, либо выписывая их, либо вычислительным путем, используя содержащиеся в Лиспе средства компонирования, такие как композиция и рекурсия, например:

```
_ (funcall (дважды (дважды 'list))
           'луковица)
(((ЛУКОВИЦА)))
_(defun 8-раз (f)
  (дважды (дважды (дважды f))))
(funcall (8-раз 'list) 'луковица)
((((((ЛУКОВИЦА)))))))
(funcall (дважды (8-раз 'list))
        'луковица)
((((((((((ЛУКОВИЦА)))))))))))
```

Строя композицию, нужно быть внимательным, чтобы функции получали правильное количество параметров и соответствующих типов.

### Рекурсивные функции с функциональным значением



Функция с функциональным значением, предназначенная для абстрагирования повторяющихся вычислений, может, естественно, быть и рекурсивной. В качестве примера рекурсивной функции с функциональным значением мы приведем функцию MAPDEF, которую можно использовать в функциональных определениях многих функционалов, в том числе в представленных нами ранее MAP-функционалах (MAPCAR, MAPLIST, MAPCAN и MAPCON):

```
(defun MAPDEF (f g)
  (function (lambda (h y)
    (if (null y) nil
        (funcall f
                  (funcall h (funcall g y))
                  (funcall
                    (MAPDEF f g); рекурсия
                    h
                    (cdr y)))))))
```

В этой абстракции функция F выражает действия, с помощью которых осуществляется построение результата (CONS или NCONC), и G определяет объект, к которому применяется функциональный аргумент. Функция H в вызове MAP-функции является функциональным аргументом, а Y – списочным аргументом.

Для функционалов MAPCAR и MAPLIST мы, например, получим следующие определения, приведенные на следующей странице.

MAPDEF является одновременно функционалом и функцией с функциональным значением, поскольку функции используются как в качестве параметров, так и в качестве результата. В качестве функционального

```

;; ; функция построения результата
;; ; MAPCAR - CONS (F = CONS)
;; ; функциональный аргумент применяется
;; ; к головной части (G = CAR)

(defun mapcar1 (f l) ; L - аргумент-список
  (funcall (mapdef 'cons 'car)
           f l))

;; ; MAPCIST, как и выше, только F
;; ; применяется к хвосту

(defun maplist1 (f l)
  (funcall (mapdef 'cons
                    (function
                      (lambda (l) l)))
           f l))

```

аргумента используют различные функции, однако основные действия выполняет функция MAPDEF.

### Автоаппликация и авторепликация



Когда речь шла о функциях более высокого порядка, мы рассматривали автоаппликативные и авторепликативные функционалы как функции, которые получают или возвращают в качестве результата самих себя или точнее копии самих себя. Конечно не очевидно, что такие функции вообще существуют. Однако никаких принципиальных препятствий для их существования и определения не существует. В Лиспе их можно определить и использовать довольно просто.

Для начала рассмотрим автоаппликативный вариант факториала:

```

(defun ! (n) (факториал 'факториал n))

(defun факториал (f n)
  (if (zerop n) 1
      (* n (funcall f f (- n 1)))))

```

Вычисления, применяющие сами себя, возникнут, если функционалу ФАКТОРИАЛ в его вызове в качестве функционального аргумента передать его собственное определение. Такой вызов может возникнуть в процессе работы других функций или в ходе рекурсии.

Соответственно тому, как мы можем через функциональный аргумент определить автоаппликативную рекурсивную функцию, мы можем определить и возвращающую себя рекурсивно функцию с функциональным значением.

Возьмем в качестве примера простейшую возможную функцию, которая возвращает себя в качестве результата. Это лямбда-выражение, примененное к такому же лямбда-выражению с формой QUOTE, т.е. это лямбда-вызов:

```
((lambda (x)
  (list x (list (quote quote) x)))
 (quote (lambda (x)
  (list x (list (quote quote) x)))))
```

Форма является одновременно автоаппликативной! Она содержит изображение самой себя, но не целиком, поскольку такое изображение было бы бесконечно длинным. Изображение получается ограниченным благодаря механизму блокировки и лямбда-механизму, а также благодаря автоаппликации.

Для проверки присвоим форму переменной САМО:

```
(setq само
  '((lambda (x)
    (list x (list 'quote x)))
   '(lambda (x)
     (list x (list 'quote x)))))
 ((LAMBDA (X) (LIST X ...))
 ;;; значением лямбда-выражения - оно само
 _ (eval (eval (eval само)))
 ((LAMBDA (X) (LIST X ...)) ; само
```

Форму можно вычислять вновь и вновь, и все равно получается тот же результат: само определение функции (его копия). Кроме того значение функции совпадает со значением аргумента ее вызова. Такое значение называется *неподвижной точкой* (fixed point) функции.



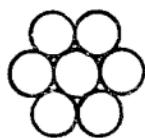
Автоаппликация как форма рекурсии с теоретической точки зрения очень интересна, но в практическом программировании, во всяком случае до настоящего времени, не находит особого применения. С нею связаны теоретические вопросы и проблемы, которые во многом еще не изучены. В формализмах и постановке проблем, которые основаны на похожих на автоаппликацию ссылках на себя (self-reference), очень легко приходят к логическим парадоксам, подобным известному из учебников по теории множеств парадоксу Рассела. Их разрешение предполагает учет типов объектов и кратности ссылок.



Однако автоаппликация может открыть новый подход к программированию. Возможными применениями могли бы быть, например, системы, сохраняющие неизменными определенные свойства, хотя какие-то их части изменяются. Такими свойствами могли бы быть наряду с применимостью и репродуцируемостью, например, способность к самоизменениям (self-modification), таким как приспособляемость, согласованность и обучаемость, самосознание (self-consciousness) и сознательность и т. д. Самосознание предполагает существование модели мира внутри такой системы, включая саму систему в эту модель.

Автофункции – это новый класс функций, которые отличаются от обычных рекурсивных функций так же, как рекурсивные функции отличаются от нерекурсивных. Чтобы их можно было использовать в программировании столь же удобно, как и рекурсию, нужно использовать механизмы, которые при программировании учитывают инвариантные свойства вычислений таким же образом, как механизмы определения, вызова и вычисления функций учитывают рекурсию.

## Порядок и тип функций



Подводя итоги, рассмотрим еще типы различных функций с функциональным значением. Понятие типа является центральным в теории функций более высокого порядка и создает предпосылки для их использования в практическом программировании.

Будем использовать следующие обозначения типов:

$N :$	Число
$T :$	Логическое значение
$S :$	Символ
$L :$	Список
$(L e) :$	Список из элементов типа $e$
$X, Y, Z :$	Произвольные типы

Для функций, получающих и возвращающих данные, таких как базовые функции, мы уже приводили следующие описания типов:

$car :$	$L \rightarrow X$
$cdr :$	$L \rightarrow L$
$cons :$	$X \times L \rightarrow L$
$atom :$	$X \rightarrow T$
$eq :$	$S \times S \rightarrow T$

Для описания функций более высокого порядка мы будем указывать типы функциональных аргументов (функционалов) или функциональных значений (функций с функциональным значением) в угловых скобках. В качестве следующего примера опишем типы рассмотренных нами ранее отображающих функционалов, работающих со списками:

$mapcar :$	$\langle X \rightarrow Y \rangle$	$\times (L X) \rightarrow (L Y)$
$maplist :$	$\langle (L X) \rightarrow Y \rangle$	$\times (L X) \rightarrow (L Y)$
$mapcan :$	$\langle X \rightarrow (L Y) \rangle$	$\times (L X) \rightarrow (L Y)$
$mapcon :$	$\langle (L X) \rightarrow (L Y) \rangle$	$\times (L X) \rightarrow (L Y)$

Тип представленного нами обобщения MAPDEF можно было бы интуитивно описать следующим образом:

$$\begin{aligned} \text{mapdef : } & \langle X \times (L Y) \rightarrow (L Y) \rangle \times \langle (L Z) \rightarrow ZI \rangle \\ & \quad \rightarrow \\ & \quad \langle ZI \rightarrow X \rangle \times (L Z) \rightarrow (L Y) \end{aligned}$$

MAPDEF является одновременно как функционалом, так и функцией с функциональным значением, поскольку в описании ее типа с обеих сторон встречаются функции.

Говорят, что функция является функцией второго порядка (order), если описание ее типа содержит функции (первого порядка). Соответствующим образом можно было бы определить описания типов и для функций более высокого порядка. Так легче представить себе действия функций.

Тип автоаппликативной функции рекурсивен и вообще не имеет порядка. Например, тип рассмотренной ранее автоаппликативной функции ФАКТОРИАЛ можно было бы описать следующим рекурсивным выражением:

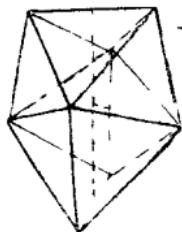
$$\text{факториал : } \langle \langle \dots \rangle \times N \rightarrow N \rangle \times N \rightarrow N$$

На практике порядок все-таки ограничен, как и порядок рекурсии в обычных функциях.

В теории типов большое значение придается обеспечению корректности программирования. Особен-но это существенно для "бестиповых" (typeless) языков программирования, таких как Лисп, Пролог, Смолтолк и другие. В них программисты могут без ограничений определять и использовать объекты различных типов, контроль типов которых целиком осуществляется лишь во время выполнения программы. Теория типов создает предпосылки для автоматической проверки правильности вычислений без выполнения функций. Это в свою очередь необходимо для создания эффективных и надежных интерпретаторов и трансляторов, а также для синтеза программ.

## Проблемы абстрактного подхода

В Лиспе функции с самого начала не определялись как отдельный тип данных, а в работе с ними опирались на другие решения подобно сохранению определения



функции в виде свойства символа. Отсюда происходят проблемы, возникающие при использовании функционалов, например связанная с контекстом вычисления фун-арг-проблема. Функции и функционалы в Лиспе не всегда являются "полноправными гражданами", иными словами, они не всегда трактуются совершенно равнозначно с другими типами данных. Даже не все Лисп-системы реализуют функционалы в общем виде.

Причины тому были частично исторические, частично технические. Теория типов данных и контекста вычислений, связанная с функциями более высокого порядка, не была известна в то время, когда создавался Лисп, и поэтому способы реализации по необходимости определились на основе технических и интуитивных представлений. Присущие ранним Лисп-системам решения и их влияние отражаются в современной Лисп-культуре, в том числе в виде различающихся и доведенных до различных степеней реализаций функционалов и функций с функциональным значением. Они, несомненно, будут прослеживаться и в будущих системах, хотя многие проблемы уже удалось решить теоретически. В новых Лисп-системах (например, Коммон Лисп и NIL) проблема с окружением решена или по крайней мере прояснена, но, например, функционалы не реализованы в их наиболее общем виде.

Значение и влияние функций более высокого порядка на программирование состоит в том, что они открывают новые возможности в обобщении и прояснении отображения вычислений. Развитие программирования и теории типов данных всегда происходило по направлению к более абстрактным формам и более мощным выразительным средствам. Это направление развития будет тем существеннее, чем большие и более сложные вычисления мы хотим запрограммировать и управлять ими.

Программирование с помощью функций более высокого порядка никак нельзя назвать простым, и его обширное использование затрудняет понимание программ, хотя они от этого и становятся короче. Определения формируются абстрактно, и на их основе не просто увидеть, сколько аргументов и каких типов передаются между функциями разного порядка. О различные стороны этого барьера сложности (complexity barrier) мы уже ударялись в связи с рекурсией более высокого порядка. Не следует слишком усложнять дело, увлекаясь функциональной абстракцией.

### Литература

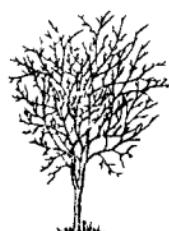
1. Boyer R. (ed.) *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*. Austin, Texas, Aug. 6–8, 1984.
2. Boyer R., Moore J. *A Computational Logic*. Academic Press, New York, 1979.
3. Allen J. *Anatomy of Lisp*. McGraw-Hill, New York, 1978.
4. Brady J. *The Theory of Computer Science. A Programming Approach*. Chapman et Hall, London, 1977.
5. Burge W. *Recursive Programming Techniques*. Addison-Wesley, Reading, Massachusetts, 1975.
6. Darlington J. (ed.) *Functional Programming and its Applications: An Advanced Course*. Cambridge University Press, Cambridge, England, 1982.
7. Henderson P. *Functional Programming – Application and Implementation*. Prentice-Hall, New Jersey, 1980.
8. Foerster H. *Notes on an Epistemology for Living Things*. Biological Computer Laboratory, University of Illinois, BCL Report No. 9.3, 1972.
9. Hofstadter D. *Gödel, Escher, Bach: An Eternal Golden Braid*. Vintage Books, New York, 1979.
10. Neumann J. *Theory of Self-Reproducing Automata*. In Burks A. (ed.), University of Illinois Press, Urbana, 1966. [Имеется перевод: фон Нейман Дж.].

- Теория самовоспроизводящихся автоматов. -М.: Мир, 1971.]
11. Perlis D., Languages with Self Reference. 1: Foundations. *Artificial Intelligence*, Vol. 25, No. 3, 1985.
  12. Scott D. Outline of a Mathematical Theory of Computation. *Proceedings of 4th Annual Princeton Conference on Information Sciences and Systems*, 1970, pp. 169-176.
  13. Stoyan H., Goerz G. *Lisp – Eine Einfuehrung in die Programmierung*. Springer-Verlag, Berlin, 1984.

### Упражнения

1. Определите функционал МНОГОФУН, который использует функции, являющиеся аргументами, по следующей схеме:
- 

$$\begin{array}{c} \text{(МНОГОФУН } '(f\ g\ \dots\ h)\ x) \\ \Leftrightarrow \\ (\text{LIST}\ (f\ x)\ (g\ x)\ \dots\ (h\ x)) \end{array}$$
2. Определите функции, которые возвращают в качестве значения
    - a) свой вызов
    - b) свое определение (лямбда-выражение)
    - c) форму своего определения (DEFUN)
  3. Приведите пример формы, для которой имеет место: (EQUAL x 'x)
  4. Какова неподвижная точка функции (lambda (x) (+ 1 (/ x 2))).



*Границы моего языка – это  
границы моего мира.*

*Л. Витгенштейн*

### 3.9 МАКРОСЫ

- Макрос строит выражение и вычисляет его значение
- Макрос не вычисляет аргументы
- Макрос вычисляется дважды
- Контекст вычисления макроса
- Пример отличия макроса от функции
- Рекурсивные макросы и продолжающиеся вычисления
- Тестирование макросов
- Лямбда-список и ключевые слова макроса
- Обратная блокировка разрешает промежуточные вычисления
- Образец удобно использовать для определения макросов
- Макросы с побочным эффектом
- Определение новых синтаксических форм
- Определение типов данных с помощью макросов
- Литература
- Упражнения

**Макрос строит выражение и вычисляет его значение**

Часто бывает полезно не выписывать вычисляемое выражение вручную, а сформировать его с помощью программы. Эта идея автоматического динамического программирования особенно хорошо реализуется в Лиспе, поскольку программа в этом языке также представляется в виде списка. При этом вычисление такого выражения или его части при необходимости можно предотвратить блокировкой (QUOTE), например, с целью преобразования выражения. Для вычис-



ления же сформированного выражения программист всегда может вызвать интерпретатор (EVAL). Предусмотренные в Лиспе специальные способы обработки аргументов открывают возможность для комбинирования различных методов.

Перечисленные возможности можно использовать в Лиспе и без специальных средств. Однако наиболее естественно программное формирование выражений осуществляется с помощью специальных макросов (macro). Внешне макросы определяются и используются так же, как функции, отличается лишь способ их вычисления. Вычисляя вызов макроса, сначала из его аргументов строится форма, задаваемая определением макроса. В результате вызова возвращается значение этой формы, а не сама форма, как было бы при вычислении функций. Таким образом, макрос вычисляется как бы в два этапа.

В предыдущих главах мы рассматривали различные виды функциональной абстракции. Макросы также представляют собой абстрактный механизм, хотя и не чисто функциональный. С его помощью можно определить формирование и вычисление произвольной (вычислимой) формы или целой программы.

Макросы дают возможность расширять синтаксис и семантику Лиспа и использовать новые подходящие для решаемой задачи формы предложений. Абстракции такого характера называют абстракциями *проблемной области* (domain abstraction), а определяемое ими расширение языка Лисп – *встроенным языком* (embedded language).

Макросы – это мощный рабочий инструмент программирования. Они дают возможность писать компактные, ориентированные на задачу программы, которые автоматически преобразуются в более сложный, но более близкий машине эффективный лисповский код. Однако с их использованием тоже связаны свои трудности и опасности. Создаваемые в процессе вычислений формы часто трудно увидеть непосредственно из определения макроса или из формы его вызова.

## Макрос не вычисляет аргументы

Синтаксис определения макроса выглядит так же, как синтаксис используемой при определении функций формы DEFUN:

**(DEFMACRO имя лямбда-список тело)**



Вызов макроса совпадает по форме с вызовом функции, но его вычисление отличается от вычисления вызова функции. Первое отличие состоит в том, что в макросе не вычисляются аргументы. Тело макроса вычисляется с аргументами в том виде, как они записаны. Такая трактовка аргументов соответствует применяемому в некоторых Лисп-системах вычислению NLAMBDA<sup>1)</sup>. В Коммон Лиспе нет функций, не вычисляющих предварительно свои аргументы, и при необходимости такие формы определяются с использованием макросов.

## Макрос вычисляется дважды



Второе отличие макроса от функции связано со способом вычисления тела макроса. Вычисление вызова макроса состоит из двух последовательных этапов. На первом этапе осуществляется вычисление тела определения с аргументами из вызова таким же образом, как и для функции. Этот этап называют этапом *расширения* (expansion) или раскрытия макроса, поскольку возникающая форма, как правило, больше и сложнее исходной формы вызова. Часто говорят и о *трансляции* (translation) макросов, поскольку на этапе расширения макровызов транслируется в некоторое вычислимое лисповское выражение. На втором этапе вычисляется полученная из вызова раскрытая форма, значение которой возвращается в качестве значения всего макровызова.

---

<sup>1)</sup> Например, в весьма распространенном диалекте Лиспа – Интерлисп.– Прим. ред.

Определим, например, макрос SETQQ, который действует наподобие SETQ, но блокирует вычисление и второго своего аргумента:

```
(defmacro setqq (x y)
  (list 'setq x (list 'quote y)))
SETQQ
(setqq список (a b c)) ; SETQQ не
(A B C)      ; вычисляет свои аргументы
список
(A B C)
```

После этапа расширения макровызова значением тела макроса было

**(SETQ СПИСОК (QUOTE (A B C)))**

На втором этапе эта программно созданная форма вычисляется обычным образом и ее значение возвращается в качестве значения вызова макроса. В этом случае у возникшей формы есть побочный эффект.

Итак, макрос – это форма, которая во время вычисления заменяется на новую, обычно более сложную форму, которая затем вычисляется обычным образом.

### Контекст вычисления макроса

Макросы отличаются от функций и в отношении контекста вычислений. В этом смысле этап расширения



*Магическое кольцо Фауста.* макроса аналогичен вычислению функции, а этап последующего вычисления – нет. Во время расширения макроса доступны синтаксические связи из контекста определения. Вычисление же

полученной в результате расширения формы производится вне контекста макровызова, и поэтому статические связи из макроса не действуют. Следовательно содержащее макровызов выражение эквивалентно выражению, в котором вызов заменен на его расширенную форму. Из-за такого определения контекста этапа вычисления макрос не во

всех случаях можно заменить функцией, в теле которой

имеется дополнительный вызов EVAL, даже если бы Лисп-система содержала функции типа NLAMBDA.

### Пример отличия макроса от функции

В качестве примера отличий функций и макросов в Коммон Лиспе рассмотрим операцию добавления элемента в стек, называемую PUSH, запрограммированную сначала обычными средствами, а потом посредством макроса. Пользователь применяет PUSH следующим образом:



**(PUSH элемент стек)**

Вызов PUSH добавлял бы в начало списка, являющегося значением переменной стека *стек*, новый элемент – *элемент*, обновляя значение переменной соответствующим образом:

```
(setq стек '(b c))
(B C)
(push 'a стек)
(A B C)
стек
(A B C)
```

Определим форму PUSH сначала как функцию:

```
(defun push1 (a p)
  (setq p (cons a (eval p))))
PUSH1
(setq стек '(b c))
(B C)
(push1 'a 'стек) ; имя стека с апострофом
(A B C)
стек
(A B C)
```

Это определение не в полной мере соответствует желаемому, поскольку его использование предполагает употребление в вызове апострофов для того, чтобы имя стека можно было передать функции. Однако функция

не во всех случаях работает верно. Если переменная А или Р определена как динамическая (special) переменная и функция PUSH1 применяется к стеку, имя которого совпадает с именем формального параметра (А или Р), то связь, содержащая значение стека, становится в окружении функции PUSH1 закрытой. Например, следующий вызов PUSH1 приведет к ошибке, поскольку значение переменной Р динамически становится равным Р:

```
(setq p '(b c))
(B C)
(defvar p)          ; определяется
P                   ; динамически
(push1 'a 'p)       ; во время вызова
(A . P)             ; значением Р является Р
p                   ; присваивание пропадет
(B C)
(setq a '(b c))   ; присваивается значение
(B C)               ; А
(defvar a)          ; А определяется
A                   ; динамически
(push1 'a 'a)       ; значение стека А во
(A . A)             ; время вызова равно А
```

Ошибочные результаты получаются из-за того, что значения динамических переменных А и Р внутри функции PUSH1 определяются по их последним связям и динамическое значение (В С) переменной Р вне функции недоступно. В Коммон Лиспе эта проблема проявляется лишь в случае динамических специальных переменных, а в системах, использующих динамическое вычисление, возникновение такого рода проблем является правилом, а не исключением.

При статическом вычислении, т.е. когда Р и А не являются динамическими, путаницы не возникает, поскольку в момент вычисления вызовов SET и EVAL переменные Р и А могут ссылаться на два различных лисповских объекта: статическую переменную из списка параметров и одноименную, находящуюся вне функции, глобальную переменную. Во время вычисле-

ния значением статической переменной является динамическая переменная. Поскольку SET вычисляет значение первого аргумента, то новое значение в качестве побочного эффекта вычисления присваивается глобальной переменной, а не статической переменной P. Следовательно, значение P в этом случае будет определено верно и новое значение останется значением глобальной переменной.

Один из способов обойти проблемы, связанные со сменой окружения вычислений, состоит в использовании в качестве имен параметров символов с необычными именами, которые не могут быть случайно использованы, например %A и %P. Лучшим способом является, однако, использование макроса:

```
_ (defmacro push2 (a p)
  (list 'setq p (list 'cons a p)))
```

Апостроф при передаче стека теперь не нужен, поскольку макровызов не вычисляет аргументы, а формы, являющиеся аргументами, связываются с формальными параметрами в том виде, как они записаны. При вычислении вызова значением параметра A будет (QUOTE A), значением параметра P – P из вызова. В качестве макрорасширения тела макроса будет получено выражение:

**(SETQ P (CONS (QUOTE A) P))**

Вычислив его, мы получим желаемый эффект без проблем, связанных с контекстом вычислений, поскольку связи формальных параметров действительны только на этапе расширения макроса.

### **Рекурсивные макросы и продолжающиеся вычисления**

Возникающее во время расширения макроса новое выражение может вновь содержать макровызовы, возможно рекурсивный. Содержащиеся в расширении макровызовы приводят на втором этапе вычислений к

 новым макровычислениям. С помощью рекурсивного макроса можно осуществить расширение, динамически зависящее от параметров, и продолжающееся макровычисление. Например, копирование верхнего уровня списка можно было бы определить следующим рекурсивным макросом:

```

_(defmacro copy-listq (x) ; аргументы не
  (cond ((null x) nil) ; вычисляются
         (t (list 'cons ; вызов CONS в
                  ; качестве расширения
                  (list 'quote (car x))
                  (list 'copy-listq
                        (cdr x))))))

COPY-LISTQ
(copy-listq (A B C))
(A B C)

```

В результате первого расширения формируется выражение, содержащее новый макровызов:

**(CONS (QUOTE A) (COPY-LISTQ (B C)))**

Таким образом, расширение на втором этапе вычислений приводит к рекурсивному макровызову. Рекурсия заканчивается на вызове (COPY-LISTQ NIL), значением расширения которого является NIL.

### Тестирование макросов

 Макросы представляют собой эффективный инструмент, которым, однако, тоже можно неверно воспользоваться. Из неправильно определенных макросов в результате расширения возникают неожиданные выражения, которые могут увести вычисления на неверный путь. Поиск ошибки в этом случае сложнее, чем обычно, поскольку раскрытие выражения сами по себе уже не видны. В этих целях для тестирования макросов существует специальная функция, которая осуществляет лишь расширение макровызова:

## (MACROEXPAND макровызов)

Форма возвращает в качестве значения результат макрорасширения вызова, которое теперь можно изучать:

```
(macroexpand '(copy-listq (a b c))
  (CONS (QUOTE A) (COPY-LISTQ (B C))))
```

### Лямбда-список и ключевые слова макроса

Как при определении и вызове функции, в случае макросов в лямбда-списке можно использовать те же ключевые слова &OPTIONAL, &REST, &KEY и &AUX.

Например, функцию LIST можно было бы определить как рекурсивную функцию и макрос следующим образом:

```
; ; LIST как функция
(defun LIST1 (&rest args)
  (if (null args) nil
      (cons (car args)
            (apply 'LIST1 (cdr args)))))

; ; LIST как макрос
(defmacro LIST2 (&rest args)
  (if (null args) nil
      (cons 'cons
            (cons (car args)
                  (cons (cons 'LIST2
                               (cdr args))
                        nil)))))
```

В данном случае определение функции яснее и короче макроопределения. В макроопределении особенно мешают многочисленные вызовы функции CONS, которые используются для построения вычисляемого выражения. Мы в дальнейшем вернемся к решению этой проблемы.

Параметр `&REST` использовался для указания на заранее неопределенное количество аргументов. Этот механизм применяется и для определения форм, не все аргументы которых нужно вычислять или в которых аргументы желательно обрабатывать нестандартным образом. Например, в обычной форме `COND` предикаты вычисляются лишь до тех пор, пока не будет получено первое значение, отличное от `NIL`. Для определения таких форм функции не подходят.

Зададим простую форму `COND` с помощью следующего рекурсивного макроса:

```
(defmacro COND1 (&rest ветви)
  (if (null ветви) nil
      (let (ветвь (car ветви)))
        (if (eval (car ветвь))
            (cadr ветвь)
            (cons 'COND1 (cdr ветви))))))
```

В лямбда-списке макроса кроме ключевых слов, входящих в определения функций, можно использовать еще некоторые, относящиеся лишь к макросам ключевые слова. Например, `&WHOLE` описывает параметр, который связывается со всей формой макровызова. Таким образом, можно из тела макроса кроме аргументов получить и имя самого макроса. Использование параметра `&WHOLE` в Коммон Лиспе соответствует в более старых Лисп-системах случаю, когда на месте лямбда-списка был один символьный параметр, который связывался со всем выражением вызова.

Другой, свойственной лишь макросам, особенностью лямбда-списка является то, что структуру параметра для определения более сложных форм вызова можно более детально изобразить с помощью подсписков (*destructuring facility*). Например, приведенное ниже макроопределение имеет четыре параметра, из которых первые три задаются в вызове списками из двух элементов. Второй элемент третьего параметра состоит еще из двух элементов, а четвертый параметр является необязательным:

```

_(defmacro герой ((имя x)
                  (кличка y)
                  (встречается (место1 место2))
                  &optional остаток)
  ...); тело
ГЕРОИ
_(герой (имя Zippy) ; макровызов
         (кличка Pinhead)
         (встречается (лisp-машины комиксы))
         (язык английский))
...
...; значение

```

### Обратная блокировка разрешает промежуточные вычисления

При раскрытии макроса обычно используется большое количество вложенных друг в друга вызовов функций CONS, CAR, CDR, LIST, APPEND и других. Поэтому при построении расширения можно легко ошибиться, а само макроопределение становится менее прозрачным. Для облегчения написания макросов в Лиспе принят специальный механизм блокировки вычислений, который называют *обратной* блокировкой и который помечается в отличие от обычной блокировки (quote) наклоненным в другую сторону (обратным) апострофом "" (back quote).

Внутри обратно блокированного выражения можно по желанию локально отменять блокировку вычислений, иными словами внутри некоторого подвыражения опять осуществлять вычисления. Отсюда происходит и название обратной блокировки. Отмена блокировки помечается запятой (,) перед каждым предназначенным для вычисления подвыражением. Запятая дает возможность на время переключиться в нормальное состояние вычислений. Пример:



```

'(не вычисляется (+ 3 4)) ; обычновенный '
'(не вычисляется (+ 3 4)) ; не отменяется

```

```
'(можно вычислить (+ 3 4)); ' действует
'(можно вычислить (+ 3 4)) ; как обычная
; блокировка
'(желательно вычислить , (+ 3 4)); , перед
'(желательно вычислить 7) ; выражением
; приводит к его вычислению
```

В расширяемом выражении при обратной блокировке выражения с предваряющей запятой заменяются на их значения. Использование предваряющей запятой можно назвать *замещающей отменой* блокировки. В Коммон Лиспе кроме запятой можно использовать запятую вместе со знаком @ (at-sign) присоединения подвыражения. Выражение, перед которым стоит признак присоединения ",@", вычисляется обычным образом, но полученное выражение присоединяется (splice) к конечному выражению таким же образом, как это делает функция APPEND. Так внешние скобки списочного значения пропадут, а элементы станут элементами списка верхнего уровня. Такую форму отмены блокировки называют *присоединяющей*. Приведем пример:

```
(setq x '(новые элементы))
(НОВЫЕ ЭЛЕМЕНТЫ)

'(включить ,x в список) ; замещающая
(ВКЛЮЧИТЬ (НОВЫЕ ЭЛЕМЕНТЫ) В СПИСОК)
; отмена
'(включить ,@x в список) ; присоединяющая
(ВКЛЮЧИТЬ НОВЫЕ ЭЛЕМЕНТЫ В СПИСОК)
; отмена
```

Блокировка вычислений и знаки отмены определены в Лиспе как макросы чтения. Способы блокировки и ее отмены в конце концов сводятся к иерархическим вызовам функции CONS. Изучить способы формирования выражений можно, задавая интерпретатору выражения, в которых внутри QUOTE содержится обратная блокировка. Например:

```
_ '(a (b ,c))
```

В качестве значения получается иерархия вызовов системных функций, вычислению значения которой препятствует апостроф.

### **Образец удобно использовать для определения макросов**

Обратная блокировка является удобным средством формирования макроса. С ее помощью макрос можно определить в форме, близкой к виду его раскрытоого выражения. Например, ранее определенный нами и казавшийся довольно сложным макрос LIST2 можно теперь определить короче:

```
(defmacro LIST3 (&rest args)
  (if (null args) nil
      '(cons ,(car args)
              ,(cons 'LIST3 (cdr args)))))
```



Обратная блокировка дает возможность определить раскрытое выражение в виде *образца* (template, skeleton), в котором динамически заполняемые формы помечены запятой. Это помогает избежать сложной комбинации вызовов функций CONS, LIST и других.

Обратная блокировка используется не только в макроопределениях. Например, построение результатов для функции PRINT часто приводит к использованию вызовов CONS, LIST и других функций. Обратной блокировкой мы вновь приближаемся к окончательному выводимому виду:

```
_ (defun добавь-и (x y z)
  (print ',x ,y и ,z))
ДОБАВЬ-И
```

```
(добавь-и 'ниф 'наф 'нуф)
(НИФ НАФ И НУФ)
(НИФ НАФ И НУФ)
```

ДОБАВЬ-И можно было бы определить и следующим макросом, для которого не нужны апострофы перед аргументами:

```
(defmacro добавь-и1 (x y z)
'(print '(,x ,у и ,z))
Добавь-И1
(добавь-и1 ниф наф нуф)
(НИФ НАФ И НУФ)
(НИФ НАФ И НУФ)
```

### Макросы с побочным эффектом

Интересный тип макросов образуют макросы с *побочным эффектом*, или макросы, меняющие определение. Поскольку, изменения, они одновременно что-то уничтожают, то их еще называют *структуроразрушающими* (*destructive*) макросами.

С помощью подходящего побочного эффекта часто можно получить более эффективное решение как в макросах, так и в работе со структуроразрушающими функциями. Однако использовать их надо очень внимательно, иначе получаемый эффект станет для программы разрушающим в буквальном смысле этого слова.

Обычно интерпретатор Лиспа расширяет макровызовы при каждом обращении заново, что в некоторых ситуациях может быть слишком неэффективным. Альтернативным решением может стать программирование такого макроса с привлечением побочного эффекта, который заменяет с помощью псевдофункций (RPLACA, RPLACD, SETF и др.) макровызовов на его расширение. Такие макросы нет надобности расширять каждый раз, так как при следующем вызове макроса на месте макровызыва будет полученное при первом вызове расширение. Например:

```
(defmacro первый (&rest x) (cons 'car x))
ПЕРВЫЙ
_ (первый '(a b c))
A
_(defun выведи-первый (x)
  (print (первый x)))
ВЫВЕДИ-ПЕРВЫЙ
```

По этим определениям при каждом вызове функции **ВЫВЕДИ-ПЕРВЫЙ** производится расширение макроса **ПЕРВЫЙ**:

```
_(выведи-первый '(a b c))
A
```

Превратим теперь макрос **ПЕРВЫЙ** в структуроразрушающий макрос:

```
(defmacro первый (arg &whole вызов)
  (rplaca вызов 'car))
```



Расширение макроса происходит теперь только при первом вызове функции **ВЫВЕДИ-ПЕРВЫЙ**, которое в качестве побочного эффекта преобразует форму (**ПЕРВЫЙ** x) в теле **ВЫВЕДИ-ПЕРВЫЙ** в форму (**CAR** x). Вычисления соответствуют ситуации, в которой **ВЫВЕДИ-ПЕРВЫЙ** с самого начала была бы определена в виде:

```
(defun выведи-первый (x)
  (print (car x)))
```

Изменяющие структуру макросы могут быть полезны особенно при работе в режиме интерпретации. В оттранслированных программах пользы от них меньше. Трансляторы Коммон Лиспа, например, способны (обычно) раскрывать и транслировать макросы уже на этапе трансляции, и не возникает необходимости в их раскрытии даже в момент первого вызова. Раскрытие можно производить и в процессе чтения выражений из

файла<sup>1)</sup>. Оттранслированные макросы можно вычислить весьма эффективно.

Макрос ПЕРВЫЙ является примером физически изменяющего вызов макроса. Макрос может физически менять и такие внешние структуры, как функции. Примером такого макроса является обобщенная форма присваивания Коммон Лиспа SETF. Ее можно определить в виде макроса, который на основе своего первого аргумента определяет, над каким типом данных (или над какой структурой) осуществляется операция присваивания, и расширяется в зависимости от этого в действия, необходимые для данного случая. Так пользователь не должен помнить об особенностях форм присваивания значений различным видам данных.

В качестве примера такой техники рассмотрим определение функции присваивания, являющейся частным случаем функции SETF, которую можно применить для присваивания значений переменным и полям списочной ячейки.

```
(defmacro := (p a) ; p = место
  (cond ; a = значение
    ((symbolp p) ; присваивание переменной
     '(setq ,p ,a))
    ((get (car p) 'метод-присваивания)
     ; поля списочной ячейки
     (cons (get (car p)
                  'метод-присваивания)
           (append (cdr p) (list a))))
    (t (format t "Неизвестная :=-форма: ~S"
               '(:= ,p ,a)))))

;; Функции присваивания - свойство имени
(setf (get 'car 'метод-присваивания)
      'rplaca)
(setf (get 'cdr 'метод-присваивания)
      'rplacd)
...
...
```

<sup>1)</sup> При распечатке макроса в системе Голден Коммон Лисп, например, макрос выдается на экран дисплея уже в раскрытом, развернутом виде.— Прим. ред.

```
(:= x '(a b c)) ; присваивание переменной
(A B C)
(:= (car (cdr x)) 'второй)
(ВТОРОЙ C) ; присваивание в поле CAR
x
(A ВТОРОЙ C)
```

### Определение новых синтаксических форм

Макросы особенно полезны при создании специальных транслирующих и трансформирующих программ. С



помощью макросов легко осуществляется определение новых синтаксических форм и структур выражений. Это необходимо, например, при добавлении в язык новых абстракций более высокого уровня, таких как структуры управления или типы данных, или при реализации интерпретатора для совершенно нового проблемно-ориентированного языка.

Определения новых форм предложений осуществляются как макросы, которые транслируют их в известные интерпретатору или ранее определенные формы. Например, предложение IF на русском языке

**(ЕСЛИ условие ТО p ИНАЧЕ q)**

можно было бы определить следующим образом:

```
(defmacro если (условие то p иначе q)
  '(if ,условие ,p ,q))
```

```
(setq x '(alea iacta est))
(ALEA IACTA EST)
_если (atom x)
  то 'орел
  иначе 'решка)
РЕШКА
```

В данном выше определении не проверяется, правильно ли указаны и на своих ли местах вспомогательные слова (ТО, ИНАЧЕ), но такую проверку можно было бы добавить к определению достаточно просто.

### Определение типов данных с помощью макросов

Кроме синтаксического определения форм макросы применяются и для определения новых абстрактных типов данных и используемых для работы с ними (зависящих от типа) функций доступа (access function). Мы уже познакомились с основными типами данных Лиспа: числами, символами и списками. Далее мы в качестве простейшего примера применения макросов для работы с типами данных зададим макрос DEFДОСТУП, который определяет предназначенную для чтения свойства символа простейшую функцию доступа, именем которой является само это свойство, а аргументом – символ. С помощью такой функции доступа можно читать свойство символа в более простой форме, чем с помощью формы GET, а именно

**(свойство символ)**

```
;;; Макрос определения функции доступа
(defmacro defдоступ (свойство)
  '(defun ,свойство (символ)
    (get символ ',свойство)))
```

В результате вызова макроса мы в качестве побочного эффекта получим определение новой функции с помощью DEFUN.

```
(setf (get 'яблоко 'цвет) 'красный)
КРАСНЫЙ ; присваивание свойства
(defдоступ цвет) ; определение с помощью
ЦВЕТ ; макроса функции доступа
(цвет 'яблоко) ; чтение свойства
КРАСНЫЙ ; теперь стало проще
```

При рассмотрении типов данных мы увидим, что, например, форма определения структур DEFSTRUCT в Коммон Лиспе является похожим по принципам, но более сложным макросом, который в качестве побочного эффекта приводит к определению многих функций доступа, зависящих от типа.

Подведем итоги:

1. С помощью макросов можно легко написать программы, которые формируют другие программы и сразу же вычисляют их.
2. С помощью макросов можно определить формы и языки более высокого уровня, которые Лисп-система транслирует в Лисп и тем самым их реализует.
3. С помощью макросов можно абстрагировать как программы, так и данные. К абстрагированию данных мы вернемся подробнее, когда будем рассматривать типы данных.

### Литература

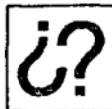
1. Brown B. *Macro Processors and Techniques for Portable Software*. John Wiley, New York, 1974.
2. Queinnec C. *LISP – Mode d'emploi*. Eyrolles, Paris, 1984.
3. Steele G. *Common LISP – The Language*. Digital Press, Hanover, Massachusetts, 1984.
4. Stoyan H., Goerz G. *Lisp – Eine Einführung in die Programmierung*. Springer-Verlag, Berlin, 1984.
5. Stratchey C. A General Purpose Macrogenerator. *Computer Journal*, Vol. 8, No. 3, 1965, pp. 225–241.
6. Waite M. *Implementation Software for Non-numerical Applications*. Prentice-Hall, Englewood Cliffs, New Jersey, 1973.
7. Weinreb D., Moon D. *Lisp Machine Manual*. MIT, Cambridge, Massachusetts, 1981.
8. Wilensky R. *LISPcraft*. W.W. Norton et Co., London, 1984.



9. Winston P., Horn B. *Lisp*. Addison-Wesley, Reading, Massachusetts, 1984.

### Упражнения

1. Что делает следующий макрос:



**(defmacro аргумент (форма)  
(list '(lambda (x) x) (саг форма)))**

2. Определите макрос, который возвращает свой вызов.

3. Что делает следующий макрос:

**(defmacro вот (&whole форма &rest args)  
форма)**

4. Определите макрос (POP стек), который читает из стека верхний элемент и меняет значение переменной стека.

5. Определите лисповскую форму (*IF условие р q*) в виде макроса.

6. Определите форму, соответствующую синтаксису оператора IF Фортрана (*FIF тест отр нуль полож*):

**3(setq x 3)  
\_ (fif x (print 'отрицательное)  
    (print 'нуль)  
    (print 'положительное))  
ПОЛОЖИТЕЛЬНОЕ**

7. Определите в виде макроса форму (REPEAT *e* UNTIL *p*) паскалевского типа.

8. Определите логическое действие ИЛИ для переменного количества аргументов (ИЛИ *x1 x2 ... xN*) так, чтобы вычисления заканчивались, когда найдено *xi*, значение которого не NIL.

# **4 ТИПЫ ДАННЫХ**

## **4.1 ПОНЯТИЯ**

## **4.2 ЧИСЛА**

## **4.3 СИМВОЛЫ**

## **4.4 СПИСКИ**

## **4.5 СТРОКИ**

## **4.6 ПОСЛЕДОВАТЕЛЬНОСТИ**

## **4.7 МАССИВЫ**

## **4.8 СТРУКТУРЫ**

До настоящего момента мы имели дело только с простыми типами данных Лиспа: атомарными объектами (числами, символами, логическими значениями), списками и средствами их внутреннего представления (списочными ячейками). Теоретически можно ограничиться лишь этими основными типами данных, однако при этом представление и использование знаний в форме, адекватной решаемой проблеме, на практике было бы столь же трудно достижимым, как и хорошее программирование с использованием только базовых функций.

Нужны типы данных более высокого уровня и другие средства абстракции данных, с помощью которых можно было бы без труда задать в каждом конкретном случае уместные понятия и подходящие объекты данных. Структуры и механизмы более высокого уровня дадут возможность отделить и скрыть детали более низкого уровня, которые с точки зрения осуществляемых действий и решаемых проблем являются несущественными. Прибегнув к ним, можно думать над задачей и программировать ее, пользуясь одинаковыми терминами, и, таким образом, сблизить языки, используемые машиной и человеком.

В этой главе мы рассмотрим наиболее важные типы данных, предусмотренные в Коммон Лиспе.

*Какая нам польза от всех наших знаний? Мы даже не знаем, какая завтра будет погода.*

*Б. Ауэрбах*

## 4.1 ПОНЯТИЯ

- Явное и неявное определение
- Абстракция данных
- Составные типы и процедуры доступа
- В Лиспе тип связан со значением, а не с именем
- Проверка и преобразование типов
- Иерархия типов
- Определение новых типов

### **Явное и неявное определение**

*Типы данных* (data type) – это подклассы используемых в языках программирования *объектов данных* (data object). Простыми типами данных являются, например, целые (integer) и вещественные (real) числа, строки (string), логические значения (boolean) и т. д. Из них можно далее формировать *составные* (compound) типы.

Более точно под типом данных понимается множество значений, которые могут иметь одинаковые объекты или *экземпляры* (instances) данного типа. Необходимые объекты можно определить *явно* (explicit), перечислив их или их имена. Второй возможностью является определение объектов *неявно* (implicit) в виде класса, описывая их структуру и свойства. В последнем случае обрабатываемые объекты создаются по мере необходимости.

В Лиспе используется как явный, так и неявный способы определения. Например, символы как таковые не требуют определения или описания перед их исполь-



зованием. Их тип определяется автоматически на основании формы и позиции, в которой они встречаются. Однако в Коммон Лиспе символам, используемым как переменные, при желании можно с помощью специальной формы **DECLARE** назначить явным образом некоторый тип, после чего такой символ может представлять лишь объекты, относящиеся к этому типу.

### Абстракция данных

Первоначально основой для выделения различных типов данных служил, внутренний формат представления данных (*word format*) в памяти машины. Поскольку данные различных типов обрабатывались разными командами, то оказалось полезным объединить формальное описание типа с описанием применяемых действий. Получаемая от этого выгода



становится тем более очевидной, чем больше используется новых различных типов. В связи с проектированием языка Алгол-68 стало ясным, сколь важным было точное определение как типов данных, так и действий над ними. Это положило начало развитию специальной области исследований, названной теорией абстрактных типов данных.

Под абстракцией данных (*data abstraction*) понимают отделение свойств данных от самих объектов данных. Под *абстрактным типом данных* (*abstract data type*) понимается тип данных, который определен через применимые к данному типу операции независимо от того, как представлены значения объектов данного типа.

Например, списки образуют абстрактный тип данных. Его объекты можно реализовать с помощью списочных ячеек и указателей или какой-нибудь другой структуры или типа данных. Базовые функции образуют множество операций (аксиомы типа). Их можно применять к спискам независимо от того, как последние реализованы на более низком уровне абстракции.

Пользуясь абстракцией типов данных, программист может забыть о технической реализации (значений

объектов) типа (*information hiding*), которая с точки зрения представления данных из проблемной области несущественна, и сосредоточиться на решении самой проблемы. В то же время вычисления в системе естественным образом распределяются между различными типами данных.

Абстрактные типы данных предлагают естественный способ разбиения программы на модули, спецификации и верификации модулей. С помощью абстракции данных можно упрощать программирование, уменьшая количество ошибок, и создавать более читаемые и легко модифицируемые программы, в которых можно менять структуры данных, не внося больших изменений в программы.

Абстракция данных позволяет определить универсальные (*generic*) действия, выглядящие с точки зрения пользователя одинаково, но реализующиеся специфическим образом в зависимости от типов объектов. Это упрощает программирование и сопровождение программ. Изменяя программу, не нужно делать изменения во всех местах, где используется данный тип. Достаточно внести изменения в определение типа. Это обстоятельство особенно важно в том случае, когда программы велики по размерам или когда над ними работают несколько программистов.

В рамках исследований по искусственноому интеллекту в течение нескольких десятилетий были предложены и отработаны многочисленные структуры данных и формы их представления. Некоторые из них уже с самого начала входили в состав Лиспа (например, так называемые ассоциативные списки и списки свойств). Некоторые более новые разработки вошли в Лисп-системы позже (например, структуры и объекты). Кроме них в Лисп-системы были включены типы данных и структуры, использованные в других языках программирования, — это различные виды чисел, строки, векторы, массивы и т. д.

Благодаря естественной расширяемости и открытости Лиспа в языке было просто определять новые типы данных и средства абстракции данных. Ни в одном другом языке программирования не найдется столь

широкого и многостороннего выбора готовых типов данных и средств для определения новых типов, как в Лиспе. Однако списки и работа с ними образуют базовый механизм, посредством которого можно использовать и объединить в единое целое лисповские объекты различных типов и с помощью которого можно реализовать новые способы представления данных.

### Составные типы и процедуры доступа

При работе с типами данных различают три основных действия: *построение* (construction) объекта данных,



*выборку* (selection) его элемента или компоненты и ее *обновление* (update, mutation). Процедуры, осуществляющие соответствующие действия, называют конструктором (constructor), селектором (selector) и мутатором (mutator) и *процедурами доступа* (access operators, accessors).

Тип данных, составленный из базовых или ранее определенных типов, называют *составным типом* (composite type). Если новый составной тип и связанные с ним действия определены, то



принадлежащие ему объекты данных можно использовать, не зная их внутренней структуры, способа представления и процедуры доступа более низкого уровня.

### В Лиспе тип связан со значением, а не с именем

В объекте данных можно различить его *имя*, сам объект, или *значение*, и *тип* объекта. Именем объекта является символ, который используется как переменная или как представитель объекта. Тип объекта определяет его общие свойства.



Монограмма  
Карла Великого.

В Лиспе тип объекта данных связан не с его именем, а с самим объектом. Именно этим Лисп отличается от многих других языков программирования. Например, в Паскале именам переменных присваивается определенный тип, и переменные нельзя использовать для

представления объектов другого типа. В Фортране в свою очередь за частью имен переменных тип закреплен уже в определении языка с 1950-х годов (имена, начинающиеся на буквы I, J и K)<sup>1)</sup>. Закрепление типа обычно осуществляется при написании программы в момент описания переменной, и его нельзя изменить в ходе работы программы.

В Лиспе в качестве значения символу можно присвоить объект произвольного типа (число, символ, список, строку, массив, структуру, поток, объект, функцию, макрос и т. д.), не заботясь о типе переменной. Одна и та же переменная может в различные моменты времени и в различной обстановке представлять лисповские объекты различных типов. Соответствие значения реальной ситуации, например аргумента функции, проверяется лишь в ходе конкретного вычисления.

Исходя из того, что переменные не имеют типов, Лисп называют *бестиповым* (typeless) языком. Подобный термин не означает того, что в языке вообще нет типов данных. Динамичность типов совместно с абстракцией данных и универсальным определением действий позволяет записывать программы в более простой и более короткой форме, чем в традиционных языках.

При желании в Коммон Лиспе можно определить и тип переменной. Это может быть целесообразным, например, в случае, когда надо ограничить область значений переменной с целью повышения эффективности или обнаружения ошибки на возможно более раннем этапе.

Тип переменной можно при необходимости определить предложением **DECLARE**. Такое определение не влияет на характер использования переменной или работу всей программы (если только форма **DECLARE** не объявляет эту переменную как специальную динамическую переменную (**SPECIAL**)). Также при определе-

<sup>1)</sup> Напомним, что такие имена используются для хранения целочисленных данных. — Прим. ред.

ни структур можно для отдельных полей структуры задать ограничения на их тип.

### Проверка и преобразование типов

Для объектов, имеющих различные типы, нужны свои функции на базовом уровне. Кроме этого, для каждого типа нужен предикат, проверяющий принадлежность к этому типу. Например, (ATOM *x*) проверяет, является ли *x* атомом. Наряду с предикатами, различающими типы (ATOM, CONSP, LISTP, NUMBERP, STRINGP, ARRAYP и другие), в Коммон Лиспе используется и обобщенный предикат проверки типа:

**(TYPEP *объект тип*)**

Значением TYPEP будет истина, если объект является объектом данного типа:

```
(typep 'атом 'atom)
T
(typep (list 'a) 'consp)
T ; является ли списочной ячейкой
```

Тип объекта можно определить и с помощью функции TYPE-OF, которая возвращает в качестве значения тип аргумента:

```
(type-of 'атом)
АТОМ
```

Кроме того, нужны функции преобразования, с помощью которых можно преобразовывать объекты данных из одного типа в другой. Во многих случаях, например при использовании чисел различных типов, эти преобразования производятся автоматически.

Наряду с множеством отдельных функций для преобразования одного типа в другой в Коммон Лиспе используется и обобщенная функция преобразования:

**(COERCE *объект тип*)**

Вызов COERCE преобразует объект, являющийся первым аргументом, к типу, задаваемому вторым аргументом. COERCE может также осуществлять преобразования между списками, строками и одномерными массивами, преобразовывать символы, состоящие из одного знака, строки и числа в соответствующие знаки и осуществлять преобразования между различными типами чисел:

```
(coerce "abc" 'list)      ; из строки в
 (#\a #\b #\c)            ; список знаков
 (coerce "a" 'character)   ; из строки в
 #\a                      ; знак
```

Встречающийся выше знак № и следующий за ним знак \ отображают тип выводимого значения. Ранее мы уже познакомились с записью

#'fn

которая изображает замыкание. Соответственно комплексные числа представляются в виде

#C(x y)

где x – действительная, а у – мнимая компоненты числа. По префиксу #C можно отличить комплексное число от обычного двухэлементного списка. В качестве примера далее приведены изображения различных типов в Коммон Лиспе:

(...)	; список (cons, list)
"..."	; строка (string)
#\x	; знак (character)
#(...)	; вектор (vector)
#*...	; битовый вектор (bit-vector)
#S(...)	; структура (structure)

В различных Лисп-системах набор типов данных, способы их представления и функции, предназначенные

ные для работы с ними, значительно отличаются друг от друга.

### Иерархия типов



Типы данных языка Коммон Лисп образуют иерархию понятий (*conceptual hierarchy*): расположенные ниже классы типов автоматически входят в расположенный выше более абстрактный класс данного типа. Например, наиболее общим типом для представления чисел является тип NUMBER, который подразделяется на следующие типы: рациональные числа, числа с плавающей запятой и комплексные числа (типы RATIONAL, FLOAT и COMPLEX). Из них, например, рациональные числа разбиваются на целые (INTEGER) и дробные числа (RATIO).

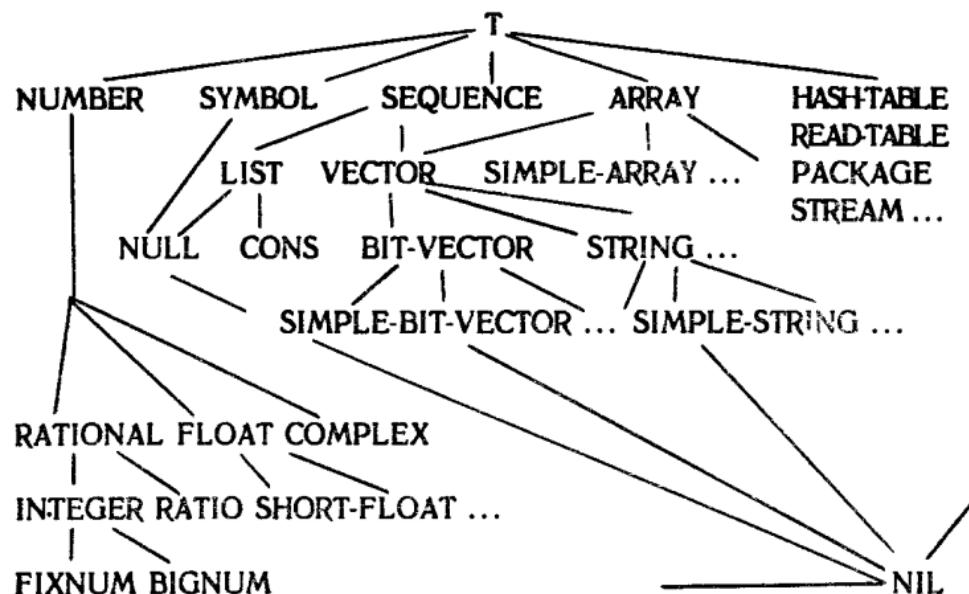
Встроенные функции, определенные для типа, расположенного выше по иерархии, можно применять и для объектов, типы которых принадлежат более низким уровням. Например, основные арифметические действия над числами +, -, \* и / можно также применять к рациональным числам, числам с плавающей запятой и комплексным числам.

Соответственно списки, строки (STRING) и векторы (одномерные массивы) принадлежат общему классу последовательностей (SEQUENCE), подтипами которого являются различные упорядоченные подмножества. Определенные для последовательностей универсальные функции, такие как LENGTH, REVERSE и т. д. можно применять как к спискам, так и к векторам и строкам.

Заметим, что если бы не было такой иерархии типов, то пришлось бы для обозначения схожих действий для строк и списков использовать функции с различными именами.

### Определение новых типов

Наряду с уже существующими типами программист может сам определить формой DEFSTRUCT новые



*Рис. 4.1.1 Иерархия наиболее важных типов данных Коммон Лиспа. Т является надтипом всех типов, а NIL – подтипом. Обратите внимание, что у некоторых типов есть несколько надтипов.*

ориентированные на задачу составные структуры, или записи. Для структур можно далее определить иерархию, для полей задать значения по умолчанию и типы и т. д.

Разграничение между системными лисповскими и используемыми в решении конкретной задачи типами часто столь же сложно, как между системными и пользовательскими функциями. Новые типы данных и другие свойства были перенесены в Лисп из связанных с практическими задачами решений и идей, про которые было замечено, что они полезны и в более общем случае. Например, объекты объектно-ориентированного программирования первоначально разрабатывались как системный тип данных для программ управления окнами Лисп-машин. Лишь позже была осознана их более общая применимость и



понятие объекта было включено во многие Лисп-системы.

Концепция объектов оказывается особенно полезным и разносторонним средством для реализации новых, особенно "активных" типов. Объектное мышление просматривается в том числе и в иерархии типов Коммон Лиспа, хотя объекты как таковые в момент написания этой книги (1985) еще не были включены в рамки разработанного стандарта этого языка. К объектам и вытекающему из этого объектно-ориентированному программированию мы подробнее вернемся во второй части книги.

Ниже мы более детально рассмотрим основные типы данных Коммон Лиспа.



*Упаси нас Господь от того,  
чтобы истина ограничивалась  
лишь математически доказу-  
емым.*

У. Блейк

## 4.2 ЧИСЛА

- Лисп умеет работать с числами
- Целые числа
- Дробные числа
- Числа с плавающей запятой
- Комплексные числа

Хотя первоначально Лисп был создан лишь для работы с символами и списками, современные системы и по вычислительной эффективности и по своим свойствам вполне годятся и для численных вычислений. Создаваемый современными трансляторами машинный код по своей эффективности в общем сравним с кодом, получаемым трансляторами с более традиционных языков. Например, для Маклиспа в свое время был разработан особенно эффективный оптимизирующий транслятор для численных вычислений. Транслятор с Коммон Лиспа для суперкомпьютера S-1 тоже производит столь же эффективный код, как и транслятор с Фортрана. Для многих Лисп-машин с целью эффективного выполнения численных операций можно приобрести специальное арифметическое устройство, или *ускоритель вычислений с плавающей точкой* (*floating point accelerator*).

### Лисп умеет работать с числами

Как правило, Лисп-системы обладают всеми средствами, необходимыми для представления численных данных. Известно, что одной из целей Коммон Лиспа было стать гибкой и эффективной альтернативой

Фортрану в области численного научно-технического программирования.

Наиболее общим числовым типом стандарта Коммон Лисп является NUMBER. Для него определены проверяющие знак и вид числа общеприменимые предикаты:

(PLUSP *число*) ; положительное  
 (MINUSP *число*) ; отрицательное  
 (ZEROP *число*) ; нуль

...

Вызов функции вычитания с одним аргументом возвращает число с измененным знаком, а функции ABS – абсолютную величину числа:

(- *x*) ; перемена знака  
 (ABS *x*) ;  $\Leftrightarrow (\text{IF } (\text{MINUSP } x) (- x) x)$

Для сравнения чисел используются следующие операции:

(= *x* &REST *xi*) ; числа равны  
 (/= *x* &REST *xi*) ; числа не равны  
 (< *x* &REST *xi*) ; числа возрастают  
 (> *x* &REST *xi*) ; числа убывают  
 (<= *x* &REST *xi*) ; возрастают или равны  
     ; (не убывают)  
 (>= *x* &REST *xi*) ; убывают или равны  
     ; (не возрастают)

...

Операции сравнения допускают произвольное число аргументов<sup>1)</sup>:

**T** (*< 1 2 3 4 5 6*)

Арифметическими операциями являются:

<sup>1)</sup> Об этом говорит ключевое слово &REST.– Прим. ред.



$(+ x \&REST xi)$	; сумма
$(- x \&REST xi)$	; вычитание
$(* x \&REST xi)$	; умножение
$(/ x \&REST xi)$	; деление
$(MIN x \&REST xi)$	; наименьшее $X_i$
$(MAX x \&REST xi)$	; наибольшее $X_i$

Удобной особенностью Коммон Лиспа являются обобщенные формы обновления INCF (increment function) и DECF (decrement function). Их можно выразить с помощью вызова SETF следующим образом:

$\Leftrightarrow$   
**(INCF место приращение)**  
 $\Leftrightarrow$   
**(SETF место (+ место приращение))**  
  
**(DECF место приращение)**  
 $\Leftrightarrow$   
**(SETF место (- место приращение))**

С их помощью можно увеличивать или уменьшать численные значения в ячейках памяти, например в переменных, элементах массива и т.д.:

```
(setq x 0)
0
(incf 'x) ; по умолчанию добавляется 1
1
(decf 'x 2)
-1
```

Формы обновления по своему эффекту и общности отличаются от вызовов  $1+$  и  $1-$ , увеличивающих и уменьшающих значение:

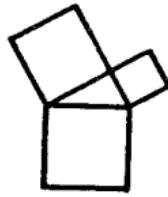
$(1+ x)$	; $\Leftrightarrow (+ 1 x)$
$(1- x)$	; $\Leftrightarrow (- x 1)$

В Коммон Лиспе имеется большой выбор определенных для всех чисел иррациональных и трансцендентных

функций, таких как экспонента, логарифм и тригонометрические функции:

<b>(EXP x)</b>	; $e$ в степени $x$
<b>(EXPT x y)</b>	; $x$ в степени $y$
<b>(LOG x &amp;OPTIONAL (y e))</b>	; логарифм $x$ по основанию $y$ ; по умолчанию – основание $e$
<b>(SQRT x)</b>	; квадратный корень

Аргументы следующих тригонометрических функций задаются в радианах и могут быть комплексными:



<b>(SIN x)</b>	и	<b>(ASIN x)</b>
<b>(COS x)</b>	и	<b>(ACOS x)</b>
<b>(TAN x)</b>	и	<b>(ATAN x)</b>

Число  $\pi$  является значением специальной переменной PI. Имеются гиперболические и обратные им функции:

<b>(SINH x)</b>	и	<b>(ASINH x)</b>
<b>(COSH x)</b>	и	<b>(ACOSH x)</b>
<b>(TANH x)</b>	и	<b>(ATANH x)</b>

Генератор случайных чисел (random number generator) Коммон Лиспа вызывается в форме:

**(RANDOM x)**

Она генерирует строго меньшее, чем  $x$  неотрицательное случайное число<sup>1)</sup>.

Рассмотренные ранее функции применимы для чисел различного типа. Кроме того для определенных типов имеются свои специфические функции.

<sup>1)</sup> Точнее генерируемое случайное число принадлежит равномерному распределению, заданному на интервале  $[0,x]$ . – Прим. ред.

## Целые числа

В стандарт Коммон Лиспа входят следующие типы чисел: *рациональные числа (RATIONAL)*, *числа с плавающей запятой (FLOAT)* и *комплексные числа (COMPLEX)*. Рациональные числа являются либо целыми числами (INTEGER), либо дробными числами (RATIO). Целые числа можно представлять с неограниченной точностью (BIGNUM) или довольствоваться целыми числами обыкновенного диапазона значений (FIXNUM), работа с которыми с вычислительной точки зрения более эффективна. Перед целым числом может стоять знак "+" или "-" и в его конце может быть десятичная точка:

**+0 ; то же, что и 0 или -0**

**1984. ; =1984 (точку в конце можно опустить)**

**-373**

Вот несколько функций и предикатов, определенных и используемых лишь для целых чисел:

**(EVENP n)** ; проверяет, четное ли число

**(ODDP n)** ;  $\Leftrightarrow (\text{NOT} (\text{EVENP } x))$

**(GCD n &REST ni)** ; наибольший общий делитель

**(LCM n &REST ni)** ; наименьшее общее кратное

Целые числа можно использовать и в логических функциях битового уровня наряду с двоичными данными.

## Дробные числа

Коммон Лисп содержит отсутствующую в традиционных языках программирования возможность использовать дробные числа без преобразования их в числа с плавающей запятой, что обычно лишь уменьшает точность их представления. Дробные числа изображаются знаком и двумя положительными числами, между которыми стоит дробная черта "/":

1/5  
 2/10 ; = 1/5  
 10/2 ; = 5  
 -31/79

Если в результате вычислений получается сокращаемое число, то автоматически происходит сокращение числа до его *канонической формы* (canonical form).

### Числа с плавающей запятой

В Коммон Лиспе используется много различных типов чисел с плавающей запятой, которые отличаются обеспечиваемой точностью. Использование более точных чисел приводит к более точному результату, но за счет увеличения времени вычисления.

Число с плавающей запятой состоит из знака (+ можно опустить), десятичного числа, записанного с использованием десятичной точки (мантийса) и отдельной знаком экспоненты экспоненциальной части, которая отображает порядок числа в терминах степени десяти:

0.0	; нуль
2.0E3	; = 2000.0
2.0E-3	; = 0.002

Вместо обычновенного формата Е в Коммон Лиспе можно использовать признаки, соответствующие следующим точностям:



S	<b>SHORT-FLOAT</b>	укороченное число
F	<b>SINGLE-FLOAT</b>	одинарная точность
D	<b>DOUBLE-FLOAT</b>	двойная точность
L	<b>LONG-FLOAT</b>	удлиненное число

Их точность зависит от реализации системы. Е по умолчанию соответствует типу с плавающей запятой с одинарной точностью, т.е. типу **SINGLE-FLOAT**. Тип определяется на основе значения, присваиваемого

пользователем глобальной системной переменной **\*READ-DEFAULT-FLOAT-FORMAT\***.

### Комплексные числа

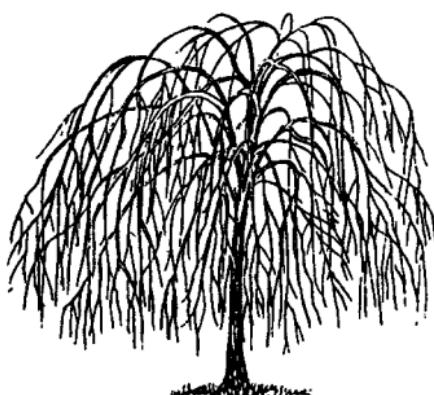
В некоторых реализациях Коммон Лиспа можно использовать и комплексные числа. Они представляются парой некомплексных чисел в форме

**#C(действительная-часть мнимая-часть)**

Например:

**#C(2.0 -3)** ; 2.0-3i  
**#C(0 1)** ; мнимая единица i

Арифметические и математические функции различных систем значительно различаются между собой как по действиям, так и по записи. Коммон Лисп современен в своей арифметической части. Более старые Лисп-системы по набору типов и по идейному подходу к ним более ограничены.



*Имена – это не только то, чем они кажутся. Обыкновенное уэльское имя *Bxixxllloшер* произносится как Джексон.*

*М. Твен*

## 4.3 СИМВОЛЫ

- Системные свойства символа
- Специальные знаки в символах
- Обращение с внешним видом символа
- GENTEMP создает новый символ

### Системные свойства символа

Ранее мы уже указывали, что с символом или с его именем могут быть связаны значение, определение функции и список свойств. Таким образом символ является структурным объектом, состоящим из четырех компонент:

КОМПОНЕНТА	ФОРМА ПРЕДСТАВЛЕНИЯ
Имя (print name)	Строка
Значение	Произвольный лисповский объект
Определение функции	Лямбда-выражение (список)
Список свойств	Список

Кроме этого, с каждым символом связаны данные о том, какому пространству имен он принадлежит.

Для чтения и проверки значений различных системных свойств существуют свои предикаты и функции, приведенные в таблице на следующей странице.

Другими функциями, соответствующими системным свойствам символа, являются MACRO-FUNCTION, которая возвращает в качестве значения макроопределение, связанное с символом, и SPECIAL-FORM-P, с

помощью которой можно проверить, есть ли у символа признак специальной формы. Предикат `FBOUNDP` возвращает значение `T`, если с именем связано определение функции, специальная форма или макрос, иначе – значение `NIL`.

СВОЙСТВО	ПРОВЕРОЧНАЯ ФУНКЦИЯ	ФУНКЦИЯ ЧТЕНИЯ
Имя	–	<code>SYMBOL-NAME</code>
Значение	<code>BOUNDP</code>	<code>SYMBOL-VALUE</code>
Функция	<code>FUNCTIONP</code>	<code>SYMBOL-FUNCTION</code>
Свойства	–	<code>SYMBOL-PLIST</code>

В предыдущих главах мы уже познакомились с различными свойствами символов, позволяющими использовать их в качестве переменной, в качестве имени функции и связывать с ним список свойств. В этой главе мы еще немного остановимся на обработке имени символа или его внешнего вида и на создании новых символов.

### Специальные знаки в символах

С каждым символом связана строка, о которой говорят как о его внешнем представлении, или о печатном



имени (`print name`, `prname`). Внешний облик символа хранится в виде строки, физически связанной с каждым символом. При выводе символа на дисплей выводится его имя. Внешний облик списков определяется его атомами и разделяющими их пробелами (или переводами строки) и ограничивающими скобками.



В начале книги говорилось, что имя символа при желании может содержать пробел, скобку или другой специальный символ, если воспользоваться знаком отмены – "обратной" косой чертой `\` (backslash), который ставится перед специальным знаком. (На терминалах со скандинавскими



Тавровые знаки.

буквами знак \ соответствует прописной букве ё.) Другой способ включения в печатное имя атома ограничивающих или других специальных символов состоит в написании всего атома между вертикальными чертами : (bar). (Снабженные скандинавскими буквами терминалы используют в качестве знака : строчную букву ё.) Это эквивалентно записи, в которой каждый знак атома специально выделен с помощью обратной косой черты:

**один атом: « \о\д\и\н\ \ а\т\о\м**

```
(setq x 'это\ один\ атом
      ЭТО\ ОДИН\ АТОМ
      (setq скобка '\))
      \)
      скобка
      \)
```

Знак отмены \ может быть виден (PRINT, PRIN1) или не виден (PRINC) в выводе на экран или печать в зависимости от используемой для вывода функции:

```
(princ скобка)
( ; вывод (PRINC)
  \ ; значение (PRINT)
```

### Обращение с внешним видом символа

Внешний вид (имя) символа можно изучить, превратив его в строку, например, функцией STRING, которая возвращает в качестве значения строку имени атома:

```
(string 'H2O)
" H2O"
```

К обращению со строками мы вернемся позднее. Имя можно разбить на знаки, составив из них список функцией COERCE:

```
(coerce (string 'H2O) 'list)
(#\H #\2 #\O)
```



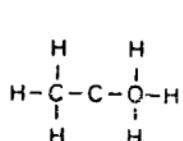
В новой форме представления можно работать с одиночными знаками, с ними можно осуществлять вычисления и строить из них новые символы. Это необходимо, например, для работы со словами естественного языка.

Для создания символа можно использовать функцию INTERN, которая включает в систему (intern), вносит в список объектов и в пространство имен символ с именем, заданным строкой.

```
(intern "C2H5OH")
C2H5OH
```

Символы можно создать и не включая их в систему. Для этого нужно использовать функцию MAKE-SYMBOL:

```
(make-symbol "C2H5OH")
C2H5OH
```



Невключенный символ является уникальным объектом данных, на который можно сослаться только лишь через ссылку.

Например, сравнение двух одноименных, но различных символов C2H5OH дает:

```
(eq 'C2H5OH (make-symbol "C2H5OH"))
NIL
```

В этом вызове EQ значением первого аргумента является включенный в систему символ C2H5OH, а значением второго – невключенный. Заменив вызов MAKE-SYMBOL на вызов INTERN, мы получим результатом EQ уже значение T:

```
(eq 'C2H5OH (intern "C2H5OH"))
T
```

Обычно в программировании на Лиспе используются лишь включенные в систему символы, чтобы применяемые в программе символьные имена имели одинаковый смысл. Если символы не включаются в систему, то с логически одинаковым именем из-за наличия физически различных символов могут быть связаны несколько различных значений, определений функции или других данных. Однако в некоторых случаях использование невключенных в систему символов может в некоторой мере сократить вычисления.

### GENTEMP создает новый символ

Для создания символов можно использовать и лисповский генератор символов (GENTEMP *тело*), который может порождать новые символы. Имя символа составляется из задаваемого в качестве необязательного аргумента начала и из порядкового номера порожденного символа в качестве суффикса:

```
(gentemp "СИМВОЛ-")
СИМВОЛ-1
(gentemp "СИМВОЛ-")
СИМВОЛ-2
(gentemp)
T1 ; "T" - начало по умолчанию
```



*Пальма – древнеиндийский символ со-зидящего духа.*

Создаваемые GENTEMP символы автоматически включаются в систему. Кроме того, гарантируется, что возвращаемый функцией символ всегда является новым: если, например, символ СИМВОЛ-2 был бы по какой-нибудь причине уже в списке объектов, то GENTEMP возвратил бы в качестве значения следующий по СИМВОЛ-3 (если бы он не был уже в употреблении).

Основой для GENTEMP является соответствующая более примитивная функция GENSYM, которая лишь возвращает новое имя, не включая его в систему.

*Все хорошие принципы уже записаны. Теперь нужно только использовать их.*

*Б. Паскаль*

## 4.4 СПИСКИ

- Ассоциативный список связывает данные с ключами
- PAIRLIS строит список пар
- ASSOC ищет пару, соответствующую ключу
- ACONS добавляет пару в начало а-списка
- PUTASSOC изменяет а-список

В предыдущих главах книги мы уже рассматривали списки и точечные пары. В этой главе мы коротко повторим эти понятия и подробнее опишем основанную на списках и точечных парах структуру данных – ассоциативный список (association list), или список пар.

Рассматривая физическую форму представления списков, мы уже отмечали, что точечная пара, или списочная ячейка, состоит из двух частей: полей CAR и CDR. Значением поля может быть произвольный лисповский объект (атом, знак, другой список, массив, структура и т. д.). Список можно определить через NIL и списочную пару следующим образом:

- 1) пустой список (), или NIL, является списком;
- 2) каждая точечная пара, поле CDR которой список, является списком.

Таким образом список – это связанная через поле CDR цепочка списочных ячеек, которая в поле CDR последней ячейки содержит NIL. Поля CAR этих ячеек представляют элементы списка.

Списки образуют в Коммон Лиспе тип LIST. Его подтипы являются зарезервированный для пустого

списка NIL особый тип данных NULL и тип непустых списков CONS.

Набор действий со списками образует основной механизм представления данных в Лиспсе, с помощью которого из объектов различных типов можно строить большие комплексы данных. Элементом списка так же, как и значением переменной, может быть произвольный лисповский объект. Используя списки, можно определять новые типы и структуры данных.

### **Ассоциативный список связывает данные с ключами**

*Ассоциативный список* или просто *a-список* (*a-list*) – это структура данных, часто используемая в Лиспсе и основанная на списках и точечных парах, для работы с которой существуют готовые функции. Ассоциативный список состоит из точечных пар, поэтому его также называют *списком пар*:

$((a_1 . t_1) (a_2 . t_2) \dots (a_N . t_N))$



Первый элемент пары (CAR) называют *ключом* (key) и второй (CDR) – связанными с ключом данными (datum). Обычно ключом является символ. Связанные с ним данные могут быть символами, списками или какими-нибудь другими лисповскими объектами. С помощью а-списка можно объединить разнотипные, поименованные ключами компоненты данных в единый комплекс данных.

В работе со списками пар нужно уметь строить списки, искать данные по ключу и обновлять их. В дальнейшем коротко коснемся этих действий.

### **PAIRLIS строит список пар**



С помощью встроенной функции PAIRLIS можно строить а-список из списка ключей и списка, сформированного из соответствующих им данных. Третьим аргументом является старый а-список, в начало которого добавляются новые пары:

**(PAIRLIS ключи данные а-список)**

Приведем пример:

```
(setq 'словарь (один . one))
(ОДИН . ONE)
_(setq словарь
      (pairlis '(три два) '(three two)
                словарь))
((ТРИ . THREE) (ДВА . TWO) (ОДИН . ONE))
```

PAIRLIS можно было бы определить следующим образом:

```
(defun pairlis (ключи данные а-список)
  (if (null ключи) а-список
    (cons (cons (car ключи)
                 (car данные))
          (pairlis (cdr ключи)
                    (cdr данные) а-список))))
```

**ASSOC ищет пару, соответствующую ключу**

Ассоциативный список можно считать отображением из множества ключей в множество значений. Данные можно получить с помощью функции

**(ASSOC ключ а-список)**

которая ищет в списке пар данные соответствующие ключу, сравнивая искомый ключ с ключами пар слева направо. ASSOC можно было бы (упрощенно) определить следующим образом:

```
(defun assoc (a а-список)
  (cond ((null а-список) nil)
        ((eql (caar а-список) a)
         (car а-список))
        ; значение - пара целиком
        (t (assoc a (cdr а-список)))))
```

Например:

```
(assoc 'два словарь)
(ДВА . TWO)
_(defun переведи (слово словарь)
  (cdr (assoc слово словарь)))
ПЕРЕВЕДИ
(переведи 'два словарь)
TWO
```

Заметьте, что в качестве значения ASSOC возвращает пару целиком, а не только искомые данные, или NIL, если ключа нет. В Коммон Лиспе есть еще функция RASSOC (обратная ASSOC), которая ищет ключ по заданным данным.

Функции ASSOC в Коммон Лиспе можно задать с помощью ключевого слова :TEST функциональный аргумент, определяющий характер операции сравнения ключей, таким же образом как для MEMBER, REMOVE и других функционалов.

### **ACONS добавляет новую пару в начало списка**

Ассоциативный список можно обновлять и использовать в режиме стека. Новые пары добавляются к нему только в начало списка, хотя в списке уже могут быть данные с тем же ключом. Это осуществляется функцией ACONS:

**(ACONS *x y a-список*)  $\leftrightarrow$  (CONS (CONS *x y*) *a-список*)**

Поскольку ASSOC просматривает список слева направо и доходит лишь до первой пары с искомым ключом, то более старые пары как бы остаются в тени более новых. Используемые таким образом а-списки хорошо решают, например, проблему поддержки меняющихся связей переменных и контекста вычисления. С такой целью они используются и в самом интерпретаторе Лиспа, к программированию которого мы вернемся во втором томе.

Преимуществом такого использования является простота изменения связей и возможность возврата к

значениям старых связей. Недостатком является то, что поиск данных замедляется пропорционально длине списка.

### **PUTASSOC изменяет а-список**



Если старое значение больше не потребуется, то а-список можно изменить, физически изменив данные, связанные с ключом. Это можно, например, сделать изменяющей значение поля CDR псевдофункцией RPLACD:

**(RPLACD (ASSOC *ключ* а-список) *новое-значение*)**

С тем же успехом можно было бы использовать и обобщенную псевдофункцию SETF:

**(SETF (CDR (ASSOC *ключ* а-список)) *новое-значение*)**

Значением первого аргумента вызова будет поле CDR пары, соответствующей ключу, которое затем заменяется на данные из второго аргумента.

В других Лисп-системах часто используется специальная псевдофункция PUTASSOC (put association), которую можно было бы определить следующим образом:

```
(defun putassoc (ключ данные а-список)
  (cond
    ((null а-список) nil)
    ((eql (caar а-список) ключ)
     (rplacd (car а-список)
              данные)
     данные) ; новое значение
    ((null (cdr а-список)) ; новая пара
     (rplacd а-список
              (list (cons ключ данные))))
    (t (putassoc ключ данные
                  (cdr а-список)))))
```

Псевдофункция PUTASSOC работает со списком не на логическом, а на физическом уровне. Это означает,

что если мы использовали старый а-список в каком-нибудь другом значении, то это значение может измениться в результате побочного эффекта обновления из-за действия PUTASSOC:

```

_(setq вода '((кислород . 1)
              (водород . 2)))
((КИСЛОРОД . 1) (ВОДОРОД . 2))
_(setq лед (cons '(т-ра ниже 0) вода))
(Т-РА НИЖЕ 0) (КИСЛОРОД . 1) (ВОДОРОД .
2))
(putassoc 'состояние 'жидкость вода)
ЖИДКОСТЬ
вода
((КИСЛОРОД . 1) (ВОДОРОД . 2) (СОСТОЯНИЕ .
ЖИДКОСТЬ))
лед
((Т-РА НИЖЕ 0) (КИСЛОРОД . 1) (ВОДОРОД .
2) (СОСТОЯНИЕ . ЖИДКОСТЬ)) ; ошибка

```

Связанные с символами ВОДА и ЛЕД а-списки – это физически один и тот же список. Если теперь список ВОДА меняется, то изменится и список ЛЕД, вызывая коварную ошибку.



Алхимический  
знак воды.

Проблему можно обойти, например, используя в вызове CONS вторым аргументом вместо символа ВОДА вызова функции (COPY-LIST ВОДА). Копирование, однако, добавит работы управлению памятью, и его нельзя использовать не задумываясь. По этой причине в процессе копирования обычно избегают бессмысленного копирования всех частей списка.



*Где отсутствуют мысли, там  
слова бесполезны.*

*И. Гёте*

## 4.5 СТРОКИ

- **Знаки и строки**
- **Преобразования строк**
- **Работа со строками**
- **Наследуемые функции**

*Строки (string),* подобно символам, числам, спискам и другим простым типам, являются основным типом данных, в работе с которыми используются свои элементарные примитивные функции и предикаты. Кроме того, строки наследуют весь набор действий и свойств своих родительских типов, как, например, последовательностей (sequence).

### Знаки и строки

Строка состоит из последовательности **знаков** (*character*). Знак также является типом данных. Его объ-

мечены

екты в отличие от атомов, имя которых состоит из одного знака, изображаются в виде

Ушиные метки – над-  
резы, которые хо-  
зяин делает на ушах  
своих овец и оленей.

#\x

В этой записи *x* является изображаемым данной записью знаком или словом в случае специального знака, обычно не имеющего печатного изображения. Например:

```
#\e      ; маленькое e
#\T      ; большое t
#\Return ; перевод строки
#\Tab    ; табуляция
```

Значением знака, как и у констант, является сам знак:

```
#\e  
#\e
```

Является ли лисповский объект к знаком, можно проверить предикатом (**CHARACTERP x**).

В строке знаки записываются в последовательности друг за другом, для ограничения которой с обеих сторон в качестве ограничителя используется знак ", например:

```
"Это строка"  
"(+ 2 3)"  
"""; строка, состоящая из одного ограничителя
```

Если строку ввести в интерпретатор, то в качестве результата получим ту же строку. Стока не может представлять что-либо, кроме самой себя. Она является такой же константой, как числа и логические значения:

```
"Это строка"  
"Это строка"  
"(+ 2 3)"  
"(+ 2 3)"  
(list "abc" "def")  
("abc" "def"); список строк
```

Функцией, проверяющей, имеет ли объект тип строки, является (**STRINGP x**), значением которой будет истина, если *x* – это строка.

### Преобразования строк

Для работы со строками существует целое множество готовых системных функций. Во многих из них предполагается, что аргументом является строка. В некоторых случаях функция при необходимости автоматически преобразует аргумент в строку. В разных системах используются различные функции для преобразования атомов и списков в строки и наоборот. Преобразования

в Коммон Лиспе можно осуществлять общей функцией COERCE или функцией STRING.

## Работа со строками



В работе со строками нужны функции по крайней мере для чтения строк и изменения их элементов и для сравнения двух строк.

Произвольный элемент строки можно прочитать (т.е. сослаться на него с помощью индекса) функцией CHAR:

**(CHAR строка n)**

Приведем пример:

```
(char "Кот" 0) ;индексация начинается с 0
#\K
```

Элемент строки можно изменить с помощью обобщенной функции присваивания SETF, например:

```
(SETF (char "Кот" 0) #\P)
"Рот"
```

Строки можно сравнивать с помощью предиката STRING=:

**(STRING= x y)**

Этот предикат проверяет логическую идентичность двух строк. Вместо предиката STRING= можно использовать и предикат проверки логического равенства EQUAL, который мы ранее использовали для сравнения списков, символов и однотипных чисел. Пример на следующей странице.

Если интерпретатор Лиспа встречает атом, то происходит его сравнение с уже ранее известными атомами. Если атом новый, то это приводит к резервированию памяти под внешний вид, список свойств и другие системные свойства и к записи результата в

```

(string= "Кот" "кот") ; в отличие от
NIL                      ; атомов
(string= "Кот" "Кот")   ; равны
T
(equal "Кот" "Кот")    ; проверка более
T                         общего логического равенства

```

память. При чтении строки все эти действия не выполняются, поэтому к ним прибегают, когда данным, представленным последовательностью букв, не планируется присваивать значения, связывать с ними определения функций и т. п. Такая ситуация типична, например, для выдачи сообщений.

### Наследуемые функции

Для создания и изменения строк используется множество функций. Кроме этого, много операций наследуется из свойств более общих типов данных. Стока, в частности, определена как одномерный массив, или вектор, элементами которого являются знаки. Функции, определенные для массивов (например, действия по чтению и по сравнению элементов), можно использовать и для строк.

Векторы совместно с классом списков образуют класс упорядоченных множеств, или последовательностей (sequence). Определенные для них общие действия можно применять как к строкам, так и к спискам. Мы вернемся подробнее к действиям над последовательностями и векторами в следующих разделах.



*Обобщения указывают путь в великое царство мысли.*

*R. Эмерсон*

## 4.6 ПОСЛЕДОВАТЕЛЬНОСТИ

- **Последовательности являются списками или векторами**
- **Основные действия с последовательностями**
- **Мощные функционалы**
- **Упорядочивание последовательности**

**Последовательности являются списками или векторами**

В представлении данных часто естественно составлять из объектов данных упорядоченные множества, или

последовательности. На элементы последовательности можно сослаться через их расположение в последовательности, ничего не зная о самих элементах. С другой стороны, представленные в виде множества элементы данных можно просто подвергнуть единообразным действиям на основе их содержания, например таким, как удаление определенных элементов из множества, упорядочивание множества по какому-либо критерию и так далее.

В Коммон Лиспе для упорядоченных множеств, или *последовательностей*, определен свой общий тип данных **SEQUENCE**. Его подтипами являются

- списки,
- векторы, или одномерные массивы.

Векторы наподобие списков содержат последовательность лисповских объектов, таких как атомы, числа, списки или другие векторы. Внешней формой представления векторов является

**#(x1 x2 ... xN) ; N-элементный вектор**

В отличие от списков векторы можно обрабатывать не только с помощью представленных здесь операций над последовательностями, но также и операциями над массивами, но не функциями, действующими над списками. К элементам вектора можно, например, эффективно обращаться напрямую, выбирать и присваивать им значения, задавая номер элемента. Мы подробнее рассмотрим массивы в следующей главе.

Несмотря на представление, основанное на выделении кавычками, строки являются векторами, элементы которых – знаки. Исходя из этого, функции, этой главы, предназначенные для работы с последовательностями, можно без оговорок применять и к строкам. То же самое относится и к определенным в следующей главе действиям над массивами, которые применимы и к векторам.

Последовательность, таким образом, является довольно общим типом данных, содержащим в качестве подтипов основные типы данных Лиспа. Поэтому работа с последовательностями занимает в Коммон Лиспе важное место. Система содержит большое количество весьма разнообразных функций и функционалов работы с последовательностями.

### Основные действия с последовательностями

Для последовательностей определено около тридцати основных функций и функционалов. Многие из них допускают использование в различных вариантах, которые отличаются обычно добавлением в конец названия формы -IF и -IF-NOT. Многие действия можно гибко варьировать с помощью необязательных ключевых аргументов, которые к тому же часто являются функциональными. Мы сначала представим наиболее важные основные функции работы с последовательностями и затем некоторые функционалы, обладающие большой изобразительной мощью.



Последовательность можно создать с помощью функции MAKE-SEQUENCE:

**(MAKE-SEQUENCE *тип количества*  
&KEY :INITIAL-ELEMENT)**

У функции MAKE-SEQUENCE два обязательных и один необязательный ключевой параметр :INITIAL-ELEMENT. С его помощью при необходимости можно задать начальное значение для элементов создаваемой последовательности. Например:

```
-(setq ox
      (make-sequence 'list ; создается список
                      4 ; длиной 4
                      :initial-element 'ox))
(0X 0X 0X 0X)
```

На новый элемент можно сослаться вызовом:

**(ELT *последовательность n*)**

```
(elt ox 0) ; индексация начинается с нуля
0X
```

Для того чтобы изменить элемент, используется обобщенная функция присваивания SETF:

```
(setf (elt ox 0) 'ой)
ой
ox
(ой 0X 0X 0X)
```

Функция ELT обобщает определенную для строк функцию CHAR и определенную для списков – NTH. Определенные для подтипов специальные функции эффективнее обобщенных форм.

Подпоследовательность последовательности можно построить вызовом:

**(SUBSEQ последовательность начало &OPTIONAL конец)**

**(subseq ox 0 1)  
(сий ox)**

Кроме этого, для последовательностей обобщены многие функции, упоминавшиеся в связи с работой над списками, это – LENGTH, REVERSE, REMOVE, DELETE, RPLACA, SOME и EVERY, которые можно применять и для векторов, и для строк:

**(reverse "abc")  
"cba"  
(remove #\a "abba")  
"bb"**

### Мощные функционалы

Последовательности, как и списки, очень хорошо подходят для обработки их функционалами. Для ранее рассмотренных нами MAP-функционалов предусмотрено обобщение MAPCAR – функции MAP, которую можно применять к любому объекту в форме последовательности. Форма вызова и выполняемые MAP действия соответствуют вызову MAPCAR с тем лишь отличием, что первым аргументом формы можно задать тип последовательности результата:

**(MAP тип-результата fn &REST последовательности)**

Приведем пример:

```
(map 'list  
'(lambda (x) (if (numberp x) 'n 's))  
#(1 2 b 3)) ; вектор  
(N N S N) ; в результат – список
```

```

-(map 'string
      '(lambda (x) (if numberp x) #\n #\s)
      '(1 2 b 3)) ; список
"nnsn"          ; результат - строка

```

С помощью функционалов **EVERY** и **SOME** можно проверить, выполняется ли заданное функциональным аргументом условие на всех элементах последовательности или только на некоторых ее элементах:

```

(every 'atom '(a b c))
·t
(some 'numberp '(a b c))
nil

```

Для работы с последовательностями существуют еще более мощные формы, чем **MAP** и **SOME**. Мы перечислим их назначение в следующей таблице. У основного типа функционала из таблицы существуют версии, оканчивающиеся на **-IF** и на **-IF-NOT**:

ФУНКЦИОНАЛЫ	НАЗНАЧЕНИЕ
REMOVE(-IF-NOT)	Удаление
SUBSTITUTE(-IF-NOT)	Замена
FIND(-IF-NOT)	Поиск
POSITION(-IF-NOT)	Поиск места
COUNT(-IF-NOT)	Подсчет



Эти формы тоже обрабатывают последовательность подобно функции **MAP**, просматривая ее и осуществляя над элементами действия, задаваемые функциональным аргументом(ами): проверку, сравнение, изменение значения и тому подобное.

Способ и порядок обработки элементов задается по умолчанию, однако как порядок обработки последовательности, так и способ обработки можно определить и в месте вызова, передавая функции необязательные ключевые параметры. Таким образом

мы можем реализовать и довольно специальные фильтры.

Далее рассмотрим наиболее типичные возможности уточнения и ключи параметров:

1. :TEST и :TEST-NOT. Применяемый к элементу тест выбора. Элементы, над которыми осуществляется действия функционал, выбираются с помощью некоторого предиката. Например, по умолчанию функция REMOVE удаляет из последовательности все элементы, которые находятся в отношении EQL ко второму аргументу. Тест по умолчанию можно определить в месте вызова. Приведенный ниже вызов REMOVE удаляет из списка все числа:

```
_remove '(2 a 8 b 1 2 c)
          :test 'numberp)
(A B C)
```



Обратите внимание, что в вызове REMOVE задан не удаляемый элемент, а только предикат. Несмотря на это, система признает вызов осмысленным, поскольку предикат NUMBERP имеет один аргумент, т.е. удаляемый элемент для сравнения не нужен. В следующем вызове предикат с двумя аргументами EQL, который берется по умолчанию, заменен на более общий предикат EQUALP:

```
_remove (/ 30 10) '(3.0 a 3 b 0.3e1)
          :test 'equalp)
(A B)
```

2. :KEY. Выбор части или характеристики исследуемого элемента. С помощью этой функции можно перед проверкой выбрать некоторую характеристику или часть элемента. Например, приведенный ниже вызов REMOVE удаляет из списка все списки, начинающиеся с символа +:

```
(remove '((- 2 3) (+ 5 6) (- 4))
         :test '(lambda (x) (eq x '+))
         :key 'car)
((- 2 3) (- 4))
```

3. :START и :END. Расположение обрабатываемых элементов. Область, к которой применяется функционал, можно ограничить аргументами :START и :END. В примере ниже элементы удаляются только из позиций 2, 3 и 4:

```
(remove 'a '(a b a b a b a)
         :end 4 :start 2)
(a b b a)
```

4. :COUNT. Количество обрабатываемых элементов. Аргумент :COUNT задает число выполняемых функционалом действий:

```
(remove #\a "aaaaa" :count 3)
"aa"
```

5. :FROM-END. Порядок просмотра элементов. Элементы просматриваются по умолчанию от начала к концу. Обратный порядок можно задать, присвоив аргументу :FROM-END значение T. Для получения более сложных вычислений аргументы можно комбинировать. Например:

```
(remove-if '(lambda (x) (eq x 'кошка))
            '(кошка собака кошка овца кошка)
            :from-end t ; начиная с конца
            :count 2) ; два раза
(кошка собака овца)
```

### Упорядочивание последовательности

Часто необходимо упорядочить или отсортировать множество элементов по какому-нибудь критерию, например расположить в алфавитном порядке. Для этого существует функция:

**(SORT последовательность предикат)**

SORT упорядочивает последовательность так, что все последовательные пары элементов удовлетворяют заданному вторым аргументом двуместному предикату. Например:

```
(sort '(4 1 3 2 5) '>)  
(5 4 3 2 1)
```

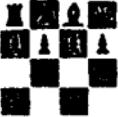


*В действительности геометрия является лишь плохой шуткой.*

*Вольтер*

## 4.7 МАССИВЫ

- Типы массивов
- Работа с массивами
- Хэш-массив ассоциирует данные
- Хэш-массивы обеспечивают быстродействие



Во многих применениях *массивы* или матрицы являются естественными структурами данных. С помощью массивов можно собрать и упорядочить данные в целом, причем отдельные элементы данных можно с помощью механизма индексирования доставать и изменять безотносительно содержания данных.

Например, точки на экране дисплея в ходе обработки изображения можно представить элементами двумерного массива, такая форма представления данных наглядна, и к ней можно непосредственно применить различные матричные операции.

Массивы создаются формой:

**(MAKE-ARRAY (*n1 n2 ... nN*) режимы)**

Функция возвращает в качестве значения новый объект – массив. *n1, n2, ..., nN* – целые числа, их количество *N* отражает *размерность* (dimensionality) массива, а значения – *размер* (size) по каждой размерности. Необязательными аргументами (режимами) можно задать тип элементов массива, указать их начальные значения или придать самому массиву динамический размер. Общий размер массива в этом случае заранее знать и закреплять не обязательно. За подробностями мы отсылаем в справочное руководство.

Например: двумерный массив, представляющий точки дисплея с экраном 1024×1024, элементами которого являются числа с плавающей запятой одинарной точности и начальным значением 0.0, можно было бы создать следующим выражением:

```
(setq экран (make-array '(1024 1024)
                         :element-type 'single-float
                         :initial-element 0.0))
```

### Типы массивов

Массивы подразделяются на *универсальные массивы* (general arrays), элементы которых могут содержать произвольный лиспovskyй объект, и *специализированные массивы* (specialized arrays), про элементы которого известно, что они всегда одного типа (битовые, численные, строковые и т. п.).

 Массив может иметь раз мерность 0, 1, 2, ... . 0-мерный массив состоит ровно из одного элемента. Одномерные массивы, или *векторы* (vectors), образуют важный специальный класс массивов, с которым мы уже познакомились в связи с последовательностями. Для него кроме общих действий над массивами и последовательностями определены свои собственные действия.

### Работа с массивами



Для вычислений, осуществляемых с массивами, наряду с функцией создания массива используются функции для выборки и изменения элементов массива. Ссылка на элемент N-мерного массива осуществляется с помощью вызова:

**(AREF массив n1 n2 ...nN)**

*n1, n2, ..., nN* – координаты, или *индексы* (index), элемента, на который ссылаются. В качестве функции

присваивания используется обобщенная функция присваивания SETF. Например, следующая директива присваивает элементу (512, 279) массива ЭКРАН число 1.0:

```
(setf (aref экран 512 279) 1.0)
```

В следующем вызове строка трактуется как массив:

<pre>(aref "Кошка" 1) #\o</pre>	<p><b>; строка в качестве</b> <b>массива</b></p>
-------------------------------------	--

Кроме того, в Коммон Лиспе существует множество функций для проверки типа и параметров массивов, для их преобразования, например для увеличения общего размера. Ограничимся ссылкой на справочное руководство.

### Хэш-массив ассоциирует данные

Кроме привычных массивов с численными индексами можно также работать с *хэш-массивами* (hash arrays). Хэш-массивы родственны обыкновенному одномерному массиву и ассоциативному списку. Если с помощью массива можно связать лисповские объекты с числами, являющимися индексами, а с помощью ассоциативного списка — с символами, являющимися ключами, то хэш-массив позволяет связать два произвольных лисповских объекта (атомы, списки, строки, массивы, хэш-массивы и т. д.).

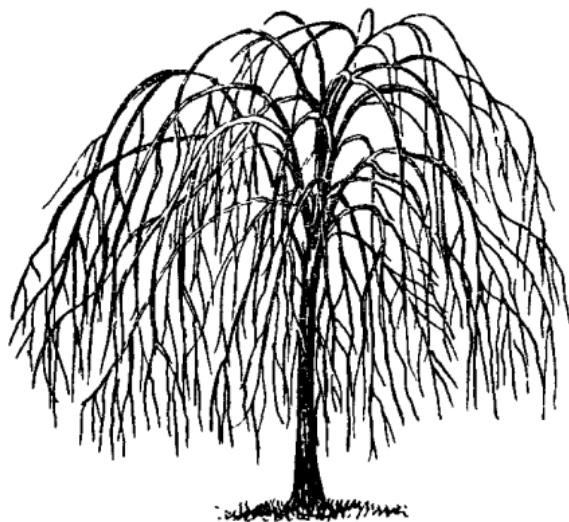
Работа с хэш-массивом напоминает работу с обычным (одномерным) массивом с тем лишь отличием, что в позиции индекса при выборке и записи в массив в качестве индекса используется указатель на лисповский объект.

### Хэш-массивы обеспечивают быстродействие

Преимуществом хэш-массивов является быстрота вычислений. Поиск данных, соответствующих ключу, например в ассоциативном списке, предполагает последовательный просмотр ключей до тех пор, пока искомый ключ не найден. Вместо этого в хэш-массиве



место хранения, соответствующее ключу, вычисляется непосредственно по виду ключа с помощью специальной хэш-функции. До искомого элемента, таким образом, добираются за определенное время независимо от объема массива<sup>1)</sup>.



---

<sup>1)</sup> С теоретической точки зрения вопрос о константности времени поиска более тонкий.— Прим. перев.

*Выясните сначала факты, а потом можете извращать их по своему усмотрению.*

*М. Твен*

## 4.8 СТРУКТУРЫ

- Структуры представляют собой логическое целое
- Определение структурного типа и создание структуры
- Функции создания, доступа и присваивания
- Действия не зависят от способа реализации

Многие факты реального мира нельзя отобразить, используя лишь ранее представленные обычные типы и структуры данных. Например, для отображения состояния шахматной партии недостаточно просто массива, а нужны еще данные об очередности хода, взятых фигурах, предыдущих ходах, уровне сложности игры и т. д.

Для объединения основных типов данных (чисел, символов, строк, массивов) в комплексы, отображающие предметы, факты, ситуации предметной области, используется составной тип, который называется *структурой*, или записью (*structure, record*). Лисповские структуры напоминают структуры Паскаля и ПЛ/1, но для их определения, создания и работы с ними в Лиспе существуют значительно более разнообразные средства.

**Структуры представляют собой логическое целое**  
В структурах к одиночным данным и различным характеристикам можно обращаться по их логическим именам независимо от внутреннего строения структуры. Мы уже ранее встречались с подобным типом данных, а именно со списком свойств символа. Свойства символа можно было присваивать, изменять и

читать по имени свойства, ничего не зная об их расположении и порядке в списке свойств. С точки зрения функций SETF, GET и REMPROP свойства могли бы с тем же успехом быть записаны в хэш-массиве или в а-списке.

### **Определение структурного типа и создание структуры**

Предположим, что наши программы имеют дело с судами. Будем интересоваться следующими характеристиками судна: двумерные координаты ( $x, y$ ) места его расположения, составляющие его скорости по каждой из координат и водоизмещение судна.

Отображающий класс судов универсальный (generic) тип структуры можно определить с помощью макроса DEFSTRUCT, формой которого (упрощенно) является

**(DEFSTRUCT** класс-структур  
поле1  
поле2  
...**)**

Например:

```
_ (defstruct судно
      х у х-скорость у-скорость тоннаж)
СУДНО
```



Символ СУДНО теперь определен как простой тип структуры, состоящий из компонент с именами Х, У, Х-СКОРОСТЬ, У-СКОРОСТЬ и ТОННАЖ.

**Функции создания, доступа и присваивания**

DEFSTRUCT – это сложный макрос. Его вызов имеет много побочных эффектов, из которых наиболее существенными являются:

1. Определение функции создания структуры. Для каждого нового типа данных генерируется начинающаяся с **MAKE**- функция создания структуры данного типа. Например, объект типа **СУДНО** можно создать и присвоить переменной **СУДНО1** следующим вызовом:

```
(setq судно1 (make-судно))
```

Полю можно с помощью ключевого слова, которым является имя поля с двоеточием перед ним, присвоить при создании начальное значение:

```
(setq судно2 (make-судно :тоннаж 8.9
                           :x 0 :x-скорость 3.0)
#S(СУДНО :X 0 :Y NIL
   :Х-СКОРОСТЬ 3.0
   :Y-СКОРОСТЬ NIL :ТОННАЖ 8.9)
; внешний вид структуры
```

Вызов **MAKE-СУДНО** возвращает в качестве значения созданную структуру. Префикс **#S** обозначает, что объект является структурой, а не простым списком.

Присваивание полю значения можно факультативно ограничить некоторым типом (режим **:TYPE**), например числами с плавающей запятой. Поле можно определить и так, чтобы нельзя было в дальнейшем его изменять (режим **:READ-ONLY**).

2. Для копирования структуры генерируется функция, начинающаяся с **COPY-** (**COPY-СУДНО**).
3. Автоматическое определение предиката проверки типа. **DEFSTRUCT** автоматически определяет новый предикат проверки типа, имя которого состоит из имени типа и следующего за ним **-P**. Например, для типа **судно** сгенерирован предикат:

```
(СУДНО-P x)
```

Значением предиката будет **T**, если лисповский объект **x** есть объект типа **СУДНО**.

4. Новый тип данных добавляется в иерархию типов данных Коммон Лиспа в качестве подтипа общего типа данных **STRUCTURE**. Таким образом, такой предназначенный для проверки встроенных типов общий предикат **TYPEP** можно использовать и для проверки типов данных пользователя.
5. Создание функций доступа для полей. Для каждого поля определяемой структуры создается функция доступа, имя которой образуется написанием после имени типа через тире имени поля, например:

**(СУДНО-ТОННАЖ x)**

Вызов возвращает тоннаж судна, задаваемого структурой x, функция СУДНО-У-СКОРОСТЬ – его у-скорость и т.д.

Для присваивания значений полям структуры используется обобщенная функция присваивания **SETF**:

```
(судно-тоннаж судно2) ; судно2
8.9 ; определено выше
(setf (судно-тоннаж судно2) 9.2)
9.2
(судно-тоннаж судно2)
9.2
```

### Действия не зависят от способа реализации

Удобством использования структур является то, что на ее различные поля можно сослаться по символьному имени поля, заданному в определении типа структуры.

Например, приведенное выше представление судна в виде структуры более естественно, чем, скажем, в виде пятиэлементного списка или вектора, первым элементом которого была бы координата x, вторым элементом – координата у и т.д. У взаимного расположения элементов нет ничего общего с собственно проблемой описания судна. Искусственные соглашения о конструкциях лишь усложняют програм-



мирование, заставляя программиста помнить, была ли у-скорость в изображающем судно списке компонентой CADDR или CADDDR. Проще сослаться на у-скорость по ее имени Y-СКОРОСТЬ. Программируя с использованием структур, можно строить структуры данных все более высокого уровня, не заботясь о том, как реализованы структуры более низкого уровня.

Детали строения структур и способы их записи в разных Лисп-системах различны.



# **5 РЕШЕНИЯ**

*Не существует ответов, есть  
лишь ссылки.*

*Библиотечный закон Винера*

## **2.1 СИМВОЛЫ И СПИСКИ**

1. Сколько элементов на верхнем уровне:
  - a) 1
  - b) 3
  - c) 4
  - d) 2
2. Пустые списки:
  - a) (NIL)
  - b) (NIL (NIL))
  - c) ((NIL (NIL (NIL)) NIL))
3. Чем отличаются: а) Кроме символов атомами являются еще числа, а также Т и NIL. б) Ничем (в Коммон Лисп), названия отражают лишь различные свойства символов. с) Выражениями наряду со списками являются еще атомы.
4. Логика высказываний: Высказывания логики можно, например, представить в виде форм:

(НЕ  $x_1$ )  
(ИЛИ  $x_1 \ x_2 \dots \ x_N$ )  
(И  $x_1 \ x_2 \dots \ x_N$ )  
(=>  $x_1 \ x_2$ )  
(<=>  $x_1 \ x_2$ )

$\text{Xi}$  здесь может быть другим высказыванием или символом, представляющим логическое значение. Например:

**(или (и t (не nil)) nil nil)**

5. Электрические цепи: представим основные типы связей в следующем виде:

**(R величина) ; Одиночное сопротивление**  
**(ПОСЛЕДОВАТЕЛЬНО k1 k2 ... kN) ; Последовательное соединение**  
**(ПАРАЛЛЕЛЬНО k1 k2 ... kN) ; Параллельное соединение**

Например:

**(последовательно  
 (г 0.52)  
 (параллельно  
 (последовательно (г 0.9)  
 (г 0.43))  
 (г 0.78))  
 (г 0.12))**

## 2.2 ПОНЯТИЕ ФУНКЦИИ

1. Значения функций: значение функции на некотором аргументе всегда одно, но для разных аргументов может быть одинаковое значение. Например, абсолютная величина числа определена однозначно, но абсолютные величины числа и соответствующего ему отрицательного числа одинаковы.
2. Попробуйте следующие вызовы:

- a) **(\* 3.234 (+ 4.56 21.45))**
- b) **(+ 5.67 (+ 6.342 12.97))** или  
**(+ 5.67 6.342 12.97)**
- c) **(/ (- 454 214 675)  
 (+ 456 (\* 678 3)))**

3. Среднее арифметическое: (/ (+ 23 5 43 17) 4)

- #### 4. Значения выражений:

- a) (+ 2 (\* 3 4))
  - b) Ошибочное выражение.
  - c) Ошибка: перед именем функции не должно быть апострофа.
  - d) 24
  - e) (QUOTE QUOTE)
  - f) 2
  - g) (QUOTE NIL)

## **2.3 БАЗОВЫЕ ФУНКЦИИ**

- ## 1. Базовые функции:

**car:** *список* → *s-выражение*

**cdr:** *спусок* → *спусок*

**cons:** *s-выражение* × *список* → *список*

**атом:** *s-выражение* → логическое значение

**eq:**    *символ x символ* → логическое-значение

- ## 2. Выделение атома ЦЕЛЬ:

a)  $(\text{car} (\text{cdr} (\text{cdr} x))) = (\text{caddr} x)$

$$b) (\text{car} (\text{cdr} (\text{car} (\text{cdr} x)))) = (\text{cadadr } x)$$

c)  $(\text{car} (\text{cdr} (\text{cdr} (\text{car} (\text{cdr} (\text{car} (\text{cdr} (\text{car} x)))))))) = (\text{caddar} (\text{cdadar} x))$

- ### 3. Значения выражений:

- а) (NIL СУТЬ ПУСТОЙ СПИСОК)

- b) (NIL)

- c) ((NIL) NIL)

- d) (A B)

- e) CAR

- f) Ошибка, так как В – атом.

- g) ((A B) (CAR (C D))))

- 4** Вызовы, возвращающие значение T: c), d), h), i) и j).

## 2.4 ИМЯ И ЗНАЧЕНИЕ СИМВОЛА

1. Значения X и Y:

- a) X=Y, Y=Y
- b) X=Y, Y=X
- c) X=Y, Y=X

2. Значение A: a) B b) C

3. Значения выражений:

- a) (CAR (QUOTE (A B C)))
- b) A
- c) Ошибка, так как у A нет значения.
- d) QUOTE
- e) (EVAL (QUOTE (QUOTE QUOTE))))

## 2.5 ОПРЕДЕЛЕНИЕ ФУНКЦИЙ

1. Определение функции:

(lambda (x y) (- (+ x y) (\* x y))))

2. Лямбда-вызовы:

- a) (Y)
- b) Ошибка: у аргумента Y нет значения.
- c) ((NIL))
- d) (((NIL)))

4. Определения функций:

- a) (defun null1 (x) (eq x nil))
- b) (defun caddr1 (x) (car (cdr (cdr x))))
- c) (defun LIST1 (x1 x2 x3)  
      (cons x1 (cons x2 (cons x3 nil)))))

5. Функция НАЗОВИ:

(defun назови (x y)  
      (eval (cons 'defun (cons x (cdr y))))))

**Определение функций сумма квадратов и НАЗОВИ:**

(назови 'суммаквадратов  
'(lambda (x y) (+ (\* x x) (\* y y))))

(назови 'назови  
'(lambda (x y)  
(eval (cons 'defun  
(cons x (cdr y)))))

Лямбда-вызовы:

- a) 19
- b) (:A 1 6 8)
- c) (1 3 NIL 1 NIL)
- d) (1 6 7 1 (:C 7))

## 6 ПЕРЕДАЧА ПАРАМЕТРОВ И ОБЛАСТЬ ИХ ДЕЙСТВИЯ

Различия переменных: а) Значение статической переменной определяется по текстуальному окружению места определения функции. Статическая переменная лишь представляет значение и сама не является символом. б) Значение динамической переменной определяется по состоянию вычислительного окружения в момент вызова. с) Специальная переменная = динамическая переменная. д) Глобальная переменная – это внешняя по отношению ко всем функциям динамическая переменная. е) Свободная переменная – это переменная, использованная в теле функции, но не упомянутая в лямбда-списке функции.

Вычисления не отличаются.

Значения вызовов:

- а) Ошибка: Y не имеет значения
- б) (Y Y)
- с) Ошибка: Y не имеет значения

- d) (Y Y)  
e) (X X) (X X)

## 2.7 ОСУЩЕСТВЛЕНИЕ ВЫЧИСЛЕНИЙ

1. Формы LET:

- a)  $\lambda(\text{let } ((x ' (+ 2 3)) (y ' c))$   
 $\quad (\text{list } x y))$   
 $\quad ((+ 2 3) C)$
- b)  $\lambda(\text{let } ((x ' a) (y ' b))$   
 $\quad (\text{let } ((z ' s))$   
 $\quad \quad (\text{list } x y z)))$
- c)  $\lambda(\text{let } ((x (\text{let } ((z ' a)) z))$   
 $\quad \quad (y ' b))$   
 $\quad \quad (\text{list } x y))$   
 $\quad (A B)$

2. Столица государства:

```
(defun столица (страна)
  (case страна
    (Финляндия 'Хельсинки)
    (Швеция 'Стокгольм)
    ...))
```

3. Среднее из трех:

```
(defun среднее (x y z)
  (cond ((> x y)
         (cond ((> y z) y)
               ((> x z) z)
               (t x)))
        (t (среднее у x z))))
```

4. Предложение IF с помощью предложения COND:  
Соответствие между предложениями IF и COND  
следующее:

$(\text{IF } p x y) \Leftrightarrow (\text{COND } (p x) (t y))$

Можно было бы предположить, что подходит следующее определение:

```
(defun if (p x y)
  (cond (p x)
        (t y)))
```

Однако определить форму IF в виде функции невозможно, поскольку функция вычисляет все свои аргументы перед выполнением тела. Предложение IF вычисляет же в зависимости от условия лишь одно из результирующих выражений.

5. Факториал с помощью предложения DO:

```
(defun ! (n)
  (do ((результат n (* результат n)))
      ((> 2 n) (if (= n 0) 1 результат))
      (setf n (- n 1))))
```

6. Произведение двух целых чисел:

```
(defun произведение (n m)
  (if (= n 1) m
      (+ m (произведение (- n 1) m))))
```

7. Длина списка:

```
(defun length1 (l)
  (cond ((null l) 0)
        (t (+ 1 (length1 (cdr l))))))
```

```
(defun length2 (l)
  (prog (результат)
        (setq результат 0)
        A   (if (null l) (return результат)
              (setq результат (+ результат 1))
              (setq l (cdr l))
              (go A))))
```

8. Функция Фибоначчи:

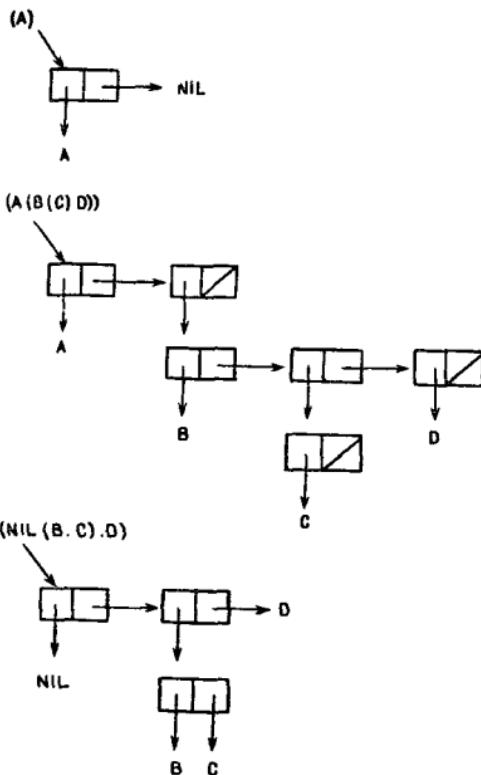
```
(defun fib (n)
  (cond ((= n 0) 1)
        ((= n 1) 1)
        (t (+ (fib (- n 1))
                (fib (- n 2)))))))
```

9. Определение функции ДОБАВЬ:

```
(defun добавь (l a)
  (cond ((null l) nil)
        (t (cons (+ a (car l))
                  (добавь (cdr l) a))))))
```

## 2.8 ВНУТРЕННЕЕ ПРЕДСТАВЛЕНИЕ СПИСКОВ

1. Рисунки списочных ячеек:



2. Списочные ячейки выражений не совпадают, хотя их поля ссылаются на одни и те же атомы.
  3. Значения выражений:
    - a)  $((A \ B) \ B)$  и  $(A \ A \ B)$
    - b)  $((A))$  и  $(A \ A)$

Если аргументы образованы физически теми же ячейками, то возникнут циклические структуры. Результат вызова RPLACA будет в обоих случаях выводится как

((((((( ...

а результат RPLACD – как

(A A A A A A A A A A A A A A ...)

4. Значения вызовов RPLACx:

  - a) (A . B)
  - b) (B)
  - c) (X . B)
  - d) (NIL)
  - e) (NIL NIL)

5. **БЕСПОЛЕЗНАЯ** возвращает в качестве значения X (как логически, так и физически), так как **RPLACD** имеет результатом свой первый аргумент.

## **2.9 СВОЙСТВА СИМВОЛА**

1. Изменение свойств: Статическая переменная не является лисповским объектом и поэтому не может иметь списка свойств. Свойства динамической переменной глобальны и не зависят от вычислительного окружения.
  2. Расстояние между городами:

```
(defun расстояние (a b)
  (sqrt (+ (expt (- (get a 'x) (get b 'x)) 2)
    (expt (- (get a 'y) (get b 'y)) 2)))))
```

3. Родители и сестры-братья:

```
(defun родители (x)
  (list (get x 'мать) (get x 'отец)))
```

```
(defun сестры-братья (x y)
  (or (eq (get x 'мать) (get y 'мать))
    (eq (get x 'отец) (get y 'отец))))
```

4. Удаление свойств:

```
(defun remprops (x)
  (cond ((symbol-plist x)
    (remprop x (car (symbol-plist x)))
    (remprops x))
    (t 'готово)))
```

5. Проверка свойства:

```
(defun hasprop (x свойство)
  (ищи-свойство свойство (symbol-plist x)))
```

```
(defun ищи-свойство (свойство список)
  (cond ((null список) nil); свойство не найдено
        ((equal свойство (cadr список))
         t) ; свойство найдено
        (t (ищи-свойство свойство
          (cddr список)))))
```

Свойство ищется путем просмотра списка свойств символа. На каждом шагу рекурсии список свойств становится на два элемента короче, чтобы на следующем шаге можно было проверить, является ли искомое свойство головой списка свойств. К рекурсивным определениям мы вернемся позднее. Нужный цикл можно легко запрограммировать и с помощью предложения DO.

## 2.10 ВВОД И ВЫВОД

1. Чтение фразы:

```
(defun читай-фразу ()
  (let ((слово (read)))
    (cond ((eq слово '?) '(?))
          (t (cons слово (читай-фразу)))))))
```

2. Ввод в режиме EVALQUOTE:

```
(defun читай ()
  (eval (cons (read) (read)))))
```

3. Проведение черты:

```
(defun черта (n)
  (cond ((= n 0) t)
        (t (princ "*")
           (черта (- n 1))))))
```

4. Заполнение прямоугольника:

```
(defun прямоугольник (ширина высота)
  (cond ((= высота 0) t)
        (t (черта ширина)
            (terpri)
            (прямоугольник ширина
              (- высота 1))))))
```

5. Соответствующая функция:

```
(defun начали ()
  (let ((имя))
    (terpri)
    (princ "Введите Ваше имя: ")
    (setq имя (read))
    (format t
      "Прекрасно, ~S, программа работает!"  
имя)))
```

### 3.2 ПРОСТАЯ РЕКУРСИЯ

1. Последний элемент списка:

```
(defun last1 (l)
  (cond ((null l) nil)
        ((null (cdr l)) (car l))
        (t (last1 (cdr l))))))
```

2. Удаление последнего элемента списка:

```
(defun droplast (l)
  (cond ((null l) nil)
        ((null (cdr l)) nil)
        (t (cons (car l) (droplast (cdr l)))))))
```

3. Проверка одноуровневости списка:

```
(defun atomlist (l)
  (cond ((null l) t)
        ((atom l) nil)
        ((atom (car l))
         (atomlist (cdr l)))
        (t nil))))
```

4. Построение списка глубиной N:

```
(defun луковица (n &optional (ядро n))
  (if (= n 0) ядро
      (cons (луковица (- n 1) ядро)
            nil))))
```

5. Первый атом списка:

```
(defun первый-атом (l)
  (cond ((atom l) l)
        (t (первый-атом (car l))))))
```

6. Удаление из списка первого вхождения:

```
(defun удалить-первый (a l)
  (cond ((null l) nil)
        ((equal (car l) a) (cdr l))
        (t (cons (car l)
                  (удалить-первый a (cdr l)))))))
```

7. Логически значения выражений равны, но физически они отличаются. В первом выражении копируется сначала Y, а затем X. Во втором копируется сначала X, затем объединение X и Y. См. определения APPEND1 и APPEND2 и разд. 2.8 "Внутреннее представление списков".
8. Другой вид выражения (REVERSE (APPEND X Y)):  

$$(\text{append} (\text{reverse } y) (\text{reverse } x))$$
9. Сравнение функций APPEND, LIST и CONS:

<i>fn</i>	<i>(fn '(A) NIL)</i>	<i>(fn NIL '(A))</i>
append:	(A)	(A)
list:	((A) NIL)	((NIL (A))
cons:	((A))	((NIL A))

10. Переворачивание и разбиение списка:
- $$(\text{defun переверни-и-разбей} (l)
 (\text{cond ((null l)} nil)
 ((null (cdr l)) l)
 (t (list (переверни-и-разбей (cdr l))
 (car l))))))$$
11. Взаимное преобразование форм (A B C) и (A (B (C))):
- $$(\text{defun разбей} (l)
 (\text{cond ((null l)} nil)
 ((null (cdr l)) l)
 (t (list (car l)
 (разбей (cdr l)))))))$$

```
(defun выровняй (l)
  (cond ((null l) nil)
        (t (cons (car l)
                  (выровняй (cadr l)))))))
```

12. Взаимное преобразование форм (A B C) и (((A) B) C):

```
(defun распредели (l)
  (слой l (list (car l)))))
```

```
(defun слой (l результат)
  (cond ((null l) результат)
        ((null (cdr l)) l)
        (t (распредели (cddr l)
                         (list результат
                               (cadr l)))))))
```

```
(defun подними (l)
  (if (atom (car l)) l
      (append (подними (car l)) (cdr l)))))
```

13. Каждый второй элемент списка:

```
(defun каждый-второй (l)
  (cond ((null l) nil)
        ((null (cdr l)) l)
        (t (cons (car l)
                  (каждыйвторой (cddr l)))))))
```

14. Разбиение элементов списка на пары:

```
(defun пары (l)
  (cond ((null l) nil)
        ((null (cdr l)) nil)
        (t (cons (list (car l) (cadr l))
                  (пары (cddr l)))))))
```

15. Чередование элементов двух списков:

```
(defun чередуй (x y)
  (cond ((null x) y)
        (t (cons (car x)
                  (чередуй y (cdr x)))))))
```

16. Факториал в виде выражения:

```
(defun факториал1 (n)
  (cond ((< n 2) '(1))
        (t (append (факториал1 (- n 1))
                  (list '* n))))))
```

### 3.3 ДРУГИЕ ФОРМЫ РЕКУРСИИ

1. Количество атомов в списочной структуре:

```
(defun атомов (x)
  (cond ((null x) 0)
        ((atom (car x))
         (+ 1 (атомов (cdr x))))
        (t (+ (атомов (car x))
               (атомов (cdr x)))))))
```

2. Глубина списка:

```
(defun глубина (x)
  (cond ((atom x) 1)
        (t (max1 (+ 1 (глубина (car x)))
                  (глубина (cdr x)))))))
```

(defun max1 (x y) ; встроенная функция MAX  
 (if (> x y) x y))

3. Преобразование записи из инфиксной в префиксную:

```
(defun вычисли (l)
  (eval (в-префиксную l))))
```

```
(defun в-префиксную (l)
  (cond ((atom l) l)
```

```

((= (length l) 2) ; унарная операция
 (cons (first l)
        (в-префиксную (second l))))
 ((= (length l) 3) ; бинарная операция
 (list (second l)
        (в-префиксную (first l))
        (в-префиксную (third l))))
 (t (format t
             "Ошибкачная форма: ~S~%" l)))

```

4. Первый совпадающий элемент в двух списках:

```

(defun первый-совпадающий (x y)
  (cond ((null x) nil)
        ((member (car x) y) (car x))
        (t (первый-совпадающий (cdr x) y))))

```

5. Предикат, проверяющий, является ли список множеством:

```

(defun множество-р (l)
  (cond ((null l) t)
        ((member (car l) (cdr l)) nil)
        (t (множество-р (cdr l)))))

```

6. Преобразование списка в множество:

```

(defun множество (l)
  (cond ((null l) nil)
        ((member (car l) (cdr l))
         (множество (cdr l)))
        (t (cons (car l)
                  (множество (cdr l)))))))

```

7. Предикат проверяющий совпадение двух множеств:

```

(defun равенство-множеств (x y)
  (cond ((null x) (null y))
        ((member (car x) y)
         (равенство-множеств (cdr x)))
        (t nil)))

```

(remove (car x) y)))

(t nil)))

8. Проверка подмножества:

```
(defun подмножество (x y)
  (cond ((null y) (null x))
        ((null x) t)
        ((member (car x) y)
         (подмножество (cdr x) y)))
        (t nil)))
```

```
(defun собственное-подмножество (x y)
  (cond ((null x) (not (null y)))
        ((null y) t)
        ((member (car x) y)
         (собственное-подмножество (cdr x)
                                     (remove (car x) y)))
        (t nil)))
```

9. Проверка различия множеств:

```
(defun непересекающиеся (x y)
  (cond ((null x) t)
        ((member (car x) y) nil)
        (t (непересекающиеся (cdr x) y)))))
```

10. Пересечение множеств:

```
(defun пересечение (x y)
  (cond ((null x) nil)
        ((member (car x) y)
         (cons (car x)
               (пересечение (cdr x) y))))
        (t (пересечение (cdr x) y)))))
```

11. Объединение множеств:

```
(defun объединение (x y)
  (cond ((null x) y)
        ((member (car x) y)
         ...)))
```

```
(объединение (cdr x) y))
(t (cons (car x)
          (объединение (cdr x) y)))))
```

12. Симметрическая разность множеств:

```
(defun симметрическая-разность (x y)
  (cond ((null x) y)
        ((member (car x) y)
         (симметрическая-разность (cdr x)
                                     (remove (car x) y)))
        (t (cons (car x)
                  (симметрическая-разность (cdr x)
                                              y))))))
```

13. Разность множеств:

```
(defun разность (x y)
  (cond ((null x) nil)
        ((member (car x) y)
         (разность (cdr x) y)))
        (t (cons (car x)
                  (разность (cdr x) y))))))
```

14. Проверка алфавитного порядка двух слов:

```
(defun алфавит-р (x y)
  (cond ((null x) t)
        ((null y) nil)
        ((eq (car x) (car y))
         (алфавит-р (cdr x) (cdr y)))
        (t (раньше-р (car x) (car y) *алфавит*))))
```

Алфавитный порядок задается здесь с помощью глобальной динамической переменной \*АЛФАВИТ\*:

```
(defvar *алфавит* '(а б в г ...))
```

Упорядочивание слов:

```
(defun словарь (слова &optional результат)
  (cond ((null слова) результат)
        (t (словарь (cdr слова)
                     (вставка (саг слова)
                     результат))))))

(defun вставка (слово слова)
  (cond ((null слова) (list слово))
        ((алфавит-р слово (саг слова))
         (cons слово слова))
        (t (cons (саг слова)
                  (вставка слово
                  (cdr слова)))))))
```

Составление обратного словаря:

```
(defun ословарь (слова)
  (cond ((null слова) nil)
        (t (переверни
                  (словарь (переверни слова))))))

(defun переверни (слова)
  (cond ((null слова) nil)
        (t (cons (reverse (саг слова))
                  (переверни (cdr слова)))))))
```

15. Поиск в упорядоченном бинарном дереве:

```
(defun ищи (а дерево)
  (cond ((null дерево) nil)
        ((equal а (first дерево)) дерево)
        ((раньше-р а (first дерево))
         (ищи а (second дерево)))
        (t (ищи а (third дерево))))))

(defun раньше-р (x y) (< x y))
; здесь данные числовые
```

16. Добавление нового элемента в упорядоченное дерево:

```
(defun добавь (а дерево)
  (cond ((null дерево) (list а nil nil))
        ((equal а (first дерево)) дерево)
        ((раньше-р а (first дерево))
         (list (first дерево)
               (добавь а (second дерево))
               (third дерево)))
        (t (list (first дерево)
                  (second дерево)
                  (добавь а (third дерево)))))))

(defun раньше-р (x y) (< x y))
; здесь данные числовые
```

17. Функции ПРЕДДЕРЕВО, ПОСЛЕДЕРЕВО и ОБЪЕДИНИ:

```
(defun преддерево (а дерево)
  (cond ((null дерево) nil)
        ((раньше-р (first дерево) а)
         (list (first дерево)
               (second дерево)
               (преддерево а (third дерево))))
        (t (преддерево а (second дерево))))))

(defun последерево (а дерево)
  (cond ((null дерево) nil)
        ((раньше-р (first дерево) а)
         (последерево а (third дерево)))
        (t (list (first дерево)
                  (последерево а
                  (second дерево))
                  (third дерево))))))

(defun раньше-р (x y) (< x y))
; здесь данные числовые
```

```
(defun объедини (p q)
  (cond ((null p) q)
        (t (list (first p)
                  (объедини
```

(преддерево (first p) q)  
 (second p))  
 (объедини  
 (последерево (first p) q)  
 (third p)))))

18. Функция БЕСПОЛЕЗНАЯ: Значение функции совпадает со значением ее аргумента.

### 3.5 ПРИМЕНЯЮЩИЕ ФУНКЦИОНАЛЫ

1. Значения вызовов APPLY и FUNCALL:

- a) (A B)
- b) ((A B))
- c) (A B)
- d) (APPLY (A B))
- e) (LIST (A B))

2. FUNCALL через функционал APPLY:

```
(defun funcall1 (fn &rest args)
  (apply fn args))
```

### 3.6 ОТОБРАЖАЮЩИЕ ФУНКЦИОНАЛЫ

1. Определение функционала MAPLIST (приблизительное):

```
(defun maplist (fn l)
  (cond ((null l) nil)
        (t (cons (funcall fn (car l))
                  (maplist fn (cdr l)))))))
```

2. Определение функционала APL-APPLY:

```
(defun apl-apply (f x)
  (cond ((null f) nil)
        (t (cons (funcall (car f)
                           (car x))
                  (apl-apply (cdr f)))))))
```

(cdr x)))))

3. Функциональные предикаты КАЖДЫЙ и НЕКОТОРЫЙ:

```
(defun каждый (p l)
  (cond ((null l) t)
        ((funcall p (car l))
         (каждый p (cdr l)))
        (t nil)))
```

```
(defun некоторый (p l)
  (cond ((null l) nil)
        ((funcall p (car l)) t)
        (t (каждый p (cdr l))))))
```

4. Фильтры УДАЛИТЬ-ЕСЛИ и УДАЛИТЬ-ЕСЛИ-НЕ:

```
(defun удалить-если (pred l)
  (марсан (function (lambda (x)
                        (if (funcall pred x) nil (list x))))
            l))
```

```
(defun удалить-если-не (pred l)
  (марсан (function (lambda (x)
                        (if (funcall pred x) (list x) nil)))
            l))
```

### 3.7 ЗАМЫКАНИЯ

1. а) Генератор чисел Фибоначчи:

```
(defun fibgen (n)
  (function
    (lambda ()
      (setq n (+ n 1))
      (fib n)))))
```

```
(defun fib (n)
  (cond ((= n 0) 1)
```

```
((= n 1) 1)
(t (+ (fib (- n 2)) (fib (- n 1))))))
```

```
(setq fib (fibgen -1))
(LEXICAL-CLOSURE ...)
(funcall fib)
t
```

б) Генератор списков:

```
(defun abgen (значение)
  (function (lambda ()
    (if (eq (car x) 'a)
        (setq значение (cons 'b значение))
        (setq значение
              (cons 'a значение))))))
  (setq ab (abgen nil))
  (LEXICAL-CLOSURE ...)
  (funcall ab)
  (A)
  (funcall ab)
  (B A))
```

### 3.8 АБСТРАКТНЫЙ ПОДХОД

- Функционал МНОГОФУН, применяющий последовательность функций к одному аргументу:

```
(defun многофун (f1 x)
  (mapcar #'(lambda (fn) (funcall fn x)) f1))
```

- Функции, возвращающие в качестве значения свой вызов, свое определение и форму своего определения:

- Функция, возвращающая в качестве значения свой вызов:

```
(defun свой-вызов (&rest args)
  (list 'apply 'свой-вызов args))
```

b) Функция, возвращающая в качестве значения свое определение:

**(defun свое-определение (&rest args)  
(symbol-function 'свое-определение))**

c) Функция, возвращающая в качестве значения форму своего определения:

**(defun сама nil  
((lambda (x)  
 (list 'defun 'сама nil  
 (list x (list 'quote x))))  
 '(lambda (x)  
 (list 'defun 'сама nil  
 (list x (list 'quote x)))))))**

3. Примеры лисповских объектов, для которых (EQUAL x 'x):

- константы (числа, Т, NIL),
- символ, значением которого является он сам,
- текст возвращающего самого себя лямбда-вызыва.

4. Неподвижная точка – 2.

### 3.9 МАКРОСЫ

1. Макрос возвращает значение своего аргумента так же, как, например, функция идентичности:

**(defun аргумент (x) x)**

2. Макрос, возвращающий свой вызов:

**(defmacro свой-вызов (&whole форма &rest args)  
 ('(quote ,форма))**

**(свой-вызов с произвольными аргументами)**

## (СВОЙ-ВЫЗОВ С ПРОИЗВОЛЬНЫМИ АРГУМЕНТАМИ)

3. Макрос ВОТ приводит с произвольными аргументами к бесконечному вычислению. В качестве расширения возникает такая же форма, как и сам вызов. После этого форма вычисляется и т.д.
4. Макрос POP, читающий из стека элемент и меняющий значение переменной стека:

```
(defmacro pop (p)
  '(prog1 (car ,p)
          (setq ,p (cdr ,p))))
```

5. Лисповская форма IF в виде макроса:

```
(defmacro if (условие p q)
  '(cond (,условие ,p)
         (t ,q)))
```

6. Фортрановский оператор IF в виде макроса (если он кому-нибудь нужен):

```
(defmacro fif (тест отр нуль полож)
  '(cond ((< ,тест 0) ,отр)
         ((= ,тест 0) ,нуль)
         (t ,полож)))
```

7. Паскалевское предложение (REPEAT e UNTIL p) в форме макроса. В следующих макросах условие окончания проверяется перед вычислением:

```
(defmacro repeat (e until p)
  '(if ,p nil
      (progn ,e (repeat ,e until ,p))))
```

или

```
(defmacro repeat (e until p)
  '(do ()
```

(,р nil)  
,e))

8. Логическая функция ИЛИ в виде макроса:

```
(defmacro или (&rest args)
  '(cond ((null ',args) nil)
         (,(car args) t)
         (t ,(cons 'или ,(cdr args))))))
```

# ПРИЛОЖЕНИЕ 1

## СВОДКА КОММОН ЛИСПА

Приложение содержит сводку определенных в языке Коммон Лисп (Steele G. Common Lisp – the Language, Digital Press, 1984) типов данных, функций и других форм и структур. Стандарт представлен шире, чем в тексте учебника. Однако содержащиеся в Коммон Лиспе не обязательные с практической точки зрения функции исключены.

Группировка функций и форм, как того требует стандарт, произведена в основном по типам данных. Однако в выделении, порядке следования и заголовках некоторых групп сделана попытка приблизиться к естественным для пользователя порядку и связям, чтобы подходящую функцию можно было найти исходя не только из типов данных, но и в соответствии с выполняемой ею операцией. Указатель символов (Приложение 2) можно использовать для поиска функции в тексте и сводке.

Предлагаемая сводка не предназначена вместо справочного руководства, а лишь дополняет его. Ее можно использовать как записную книжку по синтаксису и параметрам форм. В то же время она является связующим звеном между книгой "Мир Лиспа" – определением Коммон Лиспа – и справочными руководствами конкретных машин.

Сделана попытка описать формы и структуру их параметров в наиболее полном объеме. Описание некоторых более сложных форм ограничено рассмотрением наиболее типичных случаев. При описании дается короткое пояснение назначения и действия формы, при этом указываются номера страниц текста, где она рассматривается. В отношении деталей и незатронутых в книге форм рекомендуем обратиться к какому-либо справочному руководству по Коммон Лиспу.

## Используемые обозначения

В описании синтаксиса форм используются следующие обозначения:

1. Символы Коммон Лиспа (зарезервированные слова), такие, как имена функций и форм, выделяются **полужирным шрифтом**.
2. Параметры выделяются *курсивом* с помощью терминов на естественном языке, описывающих тип или их использование.
3. Подвыражения макросов и специальных форм, если их число переменное, помечаются фигурными скобками. Например: *{форма}*.
4. Необязательные подвыражения, т.е. подвыражения, которые можно опускать, помечаются квадратными скобками. Например: *[необязательный]*.
5. Альтернативные структуры разделяются вертикальной чертой "**!**". Например: *{декларация ! документация}* обозначает последовательность деклараций и документаций.

В описании параметров функций используются ключевые слова лямбда-списка.

Некоторые формы (например: SETQ, DEFUN и другие) не вычисляют свои аргументы или часть их, то есть перед ними в вызове не надо (нельзя) ставить апостроф. Для невычисляемых аргументов используются термины, соответствующие ситуации. Например, для атомарных символов используются термины *имя*, *имя функции* и другие.

**1 ТИПЫ ДАННЫХ**

- Иерархия типов
- Числа и биты
- Символы и списки
- Знаки
- Строки
- Векторы и массивы
- Последовательности
- Структуры
- Функции
- Замыкания и объекты
- Другие типы
- Определение типа
- Проверка типа объектов
- Преобразование типов

**2 СРАВНЕНИЕ ДАННЫХ**

- Сравнения общего типа
- Сравнения в рамках одного типа

**3 СИМВОЛЫ**

- Имя и значение
- Свойства символа
- Работа со списком свойств
- Создание символа
- Глобальные переменные и определение констант
- Обобщенные переменные (ячейки памяти)

**4 ОБРАБОТКА СПИСКОВ**

- Селекторы
- Построение и копирование списка
- Анализ списков
- Преобразование списков
- Списки как множества
- Работа с ассоциативным списком
- Применяющие функционалы
- Псевдофункции, изменяющие структуру

**5 ФУНКЦИИ**

- Описание функции
- Ключевые слова лямбда-списка
- Вызов и применение

**6 МАКРОСЫ**

- Определение макроса
- Ключевые слова определения макроса
- Вызов и расширение
- Тестирование

**7 ОПИСАНИЯ**

- Локальные и глобальные описания
- Область действия и тип переменных
- Тип формы

**8 УПРАВЛЯЮЩИЕ СТРУКТУРЫ**

- Блокировка вычислений
- Создание контекста
- Условные предложения
- Последовательное вычисление
- Циклические предложения
- Локальные и динамические переходы
- Предложение PROG
- Функциональные циклы
- Многозначные значения

**9 ЛОГИЧЕСКИЕ ДЕЙСТВИЯ**

- Логические константы
- Логические действия
- Работа с битами

**10 МАТЕМАТИЧЕСКИЕ ФУНКЦИИ**

- Числовые предикаты
- Сравнение чисел
- Арифметические действия
- Экспоненциальные и логарифмические функции
- Тригонометрические функции
- Преобразование чисел
- Случайные числа

**11 ЗНАКИ**

- Анализ типов
- Сравнение
- Преобразования

**12 СТРОКИ**

- Селекторы
- Сравнение
- Алфавитное сравнение
- Преобразования

**13 ПОСЛЕДОВАТЕЛЬНОСТИ**

- Формирование последовательности
- Предикат
- Чтение и поиск элементов
- Преобразования
- Сортировка

**14 МАССИВЫ**

- Создание массива
- Чтение и присваивание
- Анализ вида массива
- Хэш-массивы

**15 СТРУКТУРЫ**

- Определение структур
- Описание полей
- Функции, генерируемые системой

**16 ВВОД И ВЫВОД**

- Ввод
- Вывод
- Функция FORMAT
- Запрос да/нет

**17 ПОТОКИ**

- Предикат
- Создание и закрытие потока
- Преобразование потоков

**18 РАБОТА С ФАЙЛАМИ**

- Имя пути и имя файла
- Открытие и закрытие файла
- Действия с файлами
- Загрузка файлов и модулей

**19 СРЕДА ПРОГРАММИРОВАНИЯ**

- Интерпретатор
- Компилятор
- Редактор
- Комментарии и пояснения
- Отслеживание вычислений и статистика
- Ход диалога
- Использование времени и памяти
- Прерывания и обработка ошибок

## 1 ТИПЫ ДАННЫХ

В следующей таблице приведены наиболее важные типы данных Коммон Лиспа и их характерные изображения в случае, если они применимы.

ТИП	Внешнее представление	КОММЕНТАРИЙ
<b>• Иерархия типов</b>		
NIL	NIL	; подтип всех типов 317
T	T	; надтип всех типов 317
COMMON		; надтип типов Коммон Лиспа 317
<b>• Числа и биты</b>		
NUMBER		; число 316, 317, 320
RATIONAL		; рациональное число 316, 323
INTEGER	231	; целое число 316, 323
FIXNUM		; целое число ограниченной точности 323
BIGNUM		; целое число неограниченной точности 323
RATIO	2/3	; дробное число 316, 323
FLOAT	2.1E3	; число с плавающей запятой 316, 323
SHORT-FLOAT	2.1S3	; короткое число с плавающей запятой 324
LONG-FLOAT	2.1L3	; длинное число с плавающей запятой 324
SINGLE-FLOAT	2.1F3	; число с плавающей запятой одинарной точности 324
DOUBLE-FLOAT	2.1D3	; число с плавающей запятой двойной точности 324
COMPLEX	C(x y)	; комплексное число 316, 323
BIT	1 или 0	; бит
<b>• Символы и списки</b>		
SYMBOL	xxx	; символ 317, 326
NULL	NIL	; тип данных NIL-а 317, 332
LIST	(...) или NIL	; список 200, 315, 317, 331
CONS	(x . y)	; точечная пара 156, 315, 332

• Знаки

<b>CHARACTER</b>	#\...	; знак 315, 337
<b>STRING-CHAR</b>	#\...	; знак строки
<b>STANDARD-CHAR</b>	#\...	; стандартный знак

• Строки

<b>STRING</b>	"..."	; строка 315, 337
<b>SIMPLE-STRING</b>		; простая строка

• Векторы и массивы

<b>ARRAY</b>	#nA	; массивы, п – размерность 317, 349
<b>SIMPLE-ARRAY</b>		; простой массив
<b>VECTOR</b>	#(...)	; вектор 315, 341
<b>SIMPLE-VECTOR</b>		; простой вектор
<b>BIT-VECTOR</b>	#*...	; битовый вектор 317
<b>SIMPLE-BIT-VECTOR</b>		; простой битовый вектор

• Последовательности

<b>SEQUENCE</b>		; последовательность 317, 341
<b>LIST</b>	(...) или NIL	; список, см. выше
<b>VECTOR</b>	#(...)	; вектор, см. выше
<b>STRING</b>	"..."	; строка, см. выше

• Структуры

<b>STRUCTURE</b>	#S(...)	; структура 315, 353
------------------	---------	----------------------

• Функции

<b>FUNCTION</b>		; функция
<b>COMPILED-FUNCTION</b>		; компилированная функция

• Замыкания и объекты

<b>замыкание (closure)</b>		; не является самостоятель- ным типом Коммон Лиспа
<b>объект (object)</b>		; нет (еще) в Коммон Лиспе

- Другие типы

<b>PACKAGE</b>	; пространство имен 180, 317
<b>HASH-TABLE</b>	; хэш-массив 317, 351
<b>READ-TABLE</b>	; таблица чтения 177, 317
<b>STREAM</b>	; поток 187, 317
<b>PATHNAME</b>	; имя пути

- Определение типа

**(DEFTYPE имя  
      лямбда-список  
      {декларации}  
      {форма})** ; определение нового типа

Прикладные и некоторые системные типы определяются как структуры с помощью макроязыка DEFSTRUCT 306, 316, 354.

- Проверка типа объектов

<b>(TYPEP объект тип)</b>	; проверка типа 314
<b>(ATOM объект)</b>	; атомарный объект 79
<b>(KEYWORDP объект)</b>	; проверка на ключевое слово
<b>(TYPE-OF объект)</b>	; тип объекта 314

Имя предиката, определяющего тип, обычно имеет вид типР.

<b>(NUMBERP объект)</b>	; число
<b>(SYMBOLP объект)</b>	; символ
<b>(CONSP объект)</b>	; списочная ячейка
<b>(LISTP объект)</b>	; список или nil 86
<b>(FUNCTIONP объект)</b>	; функциональный объект
<b>(ARRAYP объект)</b>	; массив 86
<b>(STRINGP объект)</b>	; строка 338

Остальные предикаты для конкретных типов приведены вместе с соответствующим типом.

- Преобразование типов

**(COERCE объект нов-тип)** ; преобразование типа 314, 328, 339

Кроме того, для определенных типов используются преобразования к некоторому типу, например к знаку или строке.

## 2 СРАВНЕНИЕ ДАННЫХ

- Сравнения общего типа

<b>(EQ объект1 объект2)</b>	; физическое равенство (указателей) 79, 86, 87, 154
<b>(EQL объект1 объект2)</b>	; совпадение значения и типа 88
<b>(EQUAL объект1 объект2)</b>	; логическая идентичность объектов 89, 154, 226
<b>(EQUALP объект1 объект2)</b>	; совпадение без учета типов 90

#### • Сравнения в рамках одного типа

Для отдельных типов используются свои предикаты сравнения.  
Предикаты для наиболее важных типов:

Числа	Знаки	Строки	Алфавитный порядок (прописные = строчные)
=	CHAR=	STRING=	STRING-EQUAL
/=	CHAR/=	STRING/=	STRING-NOT-EQUAL
<	CHAR<	STRING<	STRING-LESSP
<=	CHAR<=	STRING<=	STRING-NOT-GREATERP
>	CHAR>	STRING>	STRING-GREATERP
>=	CHAR>=	STRING>=	STRING-NOT-LESSP

## 3 СИМВОЛЫ

#### • Имя и значение

<b>(SET символ значение)</b>	; вычисляющее присваивание 96, 99
<b>(SETQ {имя значение})</b>	; невычисляющее присваивание 97, 99, 128
<b>(PSETQ {имя форма})</b>	; параллельное присваивание

#### • Свойства символа

<b>(SYMBOL-NAME символ)</b>	; имя символа 327
<b>(SYMBOL-VALUE символ)</b>	; значение символа 97, 98, 327
<b>(SYMBOL-FUNCTION символ)</b>	; функциональное значение символа 110, 327
<b>(SYMBOL-PLIST символ)</b>	; список свойств символа 171, 327
<b>(SYMBOL-PACKAGE символ)</b>	; пространство имен символа
<b>(BOUNDP символ)</b>	; проверка, есть ли у символа значение 98, 327
<b>(FBOUNDP символ)</b>	; проверка, является ли символ функцией 110, 327
<b>(MAKUNBOUND символ)</b>	; удалить значение символа
<b>(FMAKUNBOUND символ)</b>	; удалить функцию символа
<b>(SAME-PNAMEP сим1 сим2)</b>	; проверка на совпадение печатного вида

• Работа со списком свойств

<b>(SYMBOL-PLIST символ)</b>	; список свойств
<b>(GET символ свойство   &amp;OPTIONAL умолч.)</b>	; значение свойства символа 169, 170
<b>(GETF память свойство   &amp;OPTIONAL умолч.)</b>	; чтение свойства из списка свойств
<b>(REMPROP символ   свойство)</b>	; удаление свойства символа 171
<b>(REMPROP память свойство)</b>	; удаление свойства из списка свойств

• Создание символа

<b>(MAKE-SYMBOL строка)</b>	; создать имя 329
<b>(GENSYM &amp;OPTIONAL x)</b>	; создать новое имя 330
<b>(GENTEMP &amp;OPTIONAL   тело пр-имен)</b>	; создать новый символ 330
<b>(INTERN строка   &amp;OPTIONAL пр-имен)</b>	; включение символа в пространство имен 329
<b>(UNINTERN символ)</b>	; удаление символа из системы
<b>(COPY-SYMBOL символ   &amp;OPTIONAL свойство)</b>	; копирование символа

• Глобальные переменные и определение констант

<b>(DEFCONSTANT имя нач-   значение [докум])</b>	; определение константы
<b>(DEFVAR имя [нач-значе-   ние [докум]])</b>	; определение динамической переменной 121, 122
<b>(DEFPARAMETER имя нач-   значение [докум])</b>	; глобальная переменная, используемая в качестве параметра

• Обобщенные переменные (ячейки памяти)

<b>(SETF {память значение})</b>	; присваивание значения в ячейку памяти
<b>(PSETF {память значение})</b>	; параллельное присваивание
<b>(SHIFT {память} нов-   значение)</b>	; сдвиг
<b>(ROTAREF {память})</b>	; циклический сдвиг

## 4 ОБРАБОТКА СПИСКОВ

• Селекторы

<b>(CAR список)</b>	; головная часть списка 79, 80, 81, 84, 91, 151
---------------------	---

(CDR список)	; хвостовая часть списка 79, 81, 84, 91, 151
(C..R список)	; композиция CAR и CDR 91
(FIRST список)	; первый элемент списка
(SECOND список)	; второй элемент списка
(THIRD список)	; третий элемент списка
(FOURTH список)	; четвертый элемент списка
(FIFTH список)	; пятый элемент списка
(SIXTH список)	; шестой элемент списка
(SEVENTH список)	; седьмой элемент списка
(EIGHTH список)	; восьмой элемент списка
(NINTH список)	; девятый элемент списка
(TENTH список)	; десятый элемент списка
(NTH n список)	; n-й элемент списка 92
(NTHCDR n список)	; n-й CDR списка
(REST список)	; хвостовая часть списка
(LAST список)	; последняя точечная пара 92
(BUTLAST список &OPTIONAL n)	; без n последних элементов

- Построение и копирование списка

(CONS голова хвост)	; присоединение головы к списку 79, 83, 84, 151, 155
(LIST &REST элементы)	; формирование списка из элементов 93, 105, 114, 296
(LIST* элемент &REST элементы)	; формирование точечного списка
(MAKE-LIST длина &KEY :INITIAL-ELEMENT)	; построение списка из одинаковых элементов
(COPY-LIST список)	; копирование списка на верхнем уровне 208, 225, 271
(COPY-TREE список)	; копирование списка на всех уровнях 225, 226

- Анализ списков

(NULL объект)	; пустой список 91
(ATOM объект)	; атомарный объект 79
(SYMBOLP объект)	; символ
(LISTP объект)	; список или nil 86
(CONSP объект)	; точечная пара
(TAILP подсписок список)	; проверка, является ли список физическим подсписком
(ENDP объект)	; проверка конца списка
(LIST-LENGTH список)	; длина списка

- Преобразование списков

<b>(REVERSE</b> <i>посл</i> )	; перевернутый список 218, 227, 234
<b>(APPEND &amp;REST</b> <i>списки</i> )	; логическое соединение списков 164, 165, 214, 216, 271
<b>(SUBST</b> <i>нов стар дерево</i> <b>&amp;KEY</b> <i>ключи</i> )	; замена вхождений элемента в список
<b>(SUBST-IF</b> <i>нов тест дерево</i> <b>&amp;KEY :KEY</b> )	; замена при условии
<b>(SUBST-IF-NOT</b> <i>нов тест</i> <i>дерево &amp;KEY :KEY</i> )	; замена, если условие не выполнено
<b>(SUBLIS</b> <i>a-список дерево</i> <b>&amp;KEY</b> <i>ключи</i> )	; замена элементов в соответствии с а-списком

#### • Списки как множества

<b>(SUBSETP</b> <i>подмнож множ</i> <b>&amp;KEY</b> <i>ключи</i> )	; проверка на подмножество
<b>(ADJOIN</b> <i>элемент множ</i> <b>&amp;KEY</b> <i>ключи</i> )	; добавить элемент в множество
<b>(UNION</b> <i>множ1 множ2</i> <b>&amp;KEY</b> <i>ключи</i> )	; сформировать объединение множеств
<b>(INTERSECTION</b> <i>множ1</i> <i>множ2 &amp;KEY</i> <i>ключи</i> )	; сформировать пересечение множеств
<b>(SET-DIFFERENCE</b> <i>множ1</i> <i>множ2 &amp;KEY</i> <i>ключи</i> )	; сформировать разность множеств
<b>(MEMBER</b> <i>элемент множ</i> <b>&amp;KEY</b> <i>ключи</i> )	; проверить, принадлежит ли элемент списку
<b>(MEMBER-IF</b> <i>условие</i> <i>множ &amp;KEY</i> <i>ключи</i> )	; проверить принадлежность при выполнении условия
<b>(MEMBER-IF-NOT</b> <i>условие</i> <i>множ &amp;KEY</i> <i>ключи</i> )	; проверить принадлежность, если условие не выполнено

#### • Работа с ассоциативным списком

<b>(PAIRLIS</b> <i>ключи данные</i> <b>&amp;OPTIONAL</b> <i>a-список</i> )	; построение а-списка 332, 333
<b>(ACONS</b> <i>ключ данное</i> <i>a-список</i> )	; добавить пару в а-список (как стек) 334
<b>(ASSOC</b> <i>ключ a-список</i> <b>&amp;KEY</b> <i>ключи</i> )	; найти пару, соответствующую ключу 333, 334
<b>(ASSOC-IF</b> <i>условие</i> <i>a-список</i> )	; найти пару по условию на ключ
<b>(ASSOC-IF-NOT</b> <i>условие</i> <i>a-список</i> )	; найти пару, если условие для ключа не выполнено
<b>(RASSOC</b> <i>ключ a-список</i> <b>&amp;KEY</b> <i>ключи</i> )	; найти пару, соответствующую значению 334
<b>(RASSOC-IF</b> <i>условие</i> <i>a-список</i> )	; найти пару по условию на значение
<b>(RASSOC-IF-NOT</b> <i>тест</i> <i>a-список</i> )	; найти пару, если условие на значение не выполнено

- Применяющие функционалы

<b>(MAPCAR fn список &amp;REST списки)</b>	; повторить для головных частей и собрать 250, 257
<b>(MAPLIST fn список &amp;REST списки)</b>	; повторить для хвостовых частей и собрать 252, 257
<b>(MAPC fn список &amp;REST списки)</b>	; повторить для головных частей 254, 257
<b>(MAPCAN fn список &amp;REST списки)</b>	; повторить для головных частей и соединить 252, 254, 257
<b>(MAPCON fn список &amp;REST списки)</b>	; повторить для хвостовых частей и соединить 252, 254, 257
<b>(MAPL fn список &amp;REST списки)</b>	; повторить для хвостовых частей 254, 255, 257
<b>(MAP тип-результата fn посл &amp;REST посл-ти)</b>	; повторить функцию для элементов последовательности 255, 344

Кроме того, для последовательностей определены обычно более обобщенные функционалы, подобные функциям MAP (см. последовательности).

- Псевдофункции, изменяющие структуру

<b>(RPLACA ячейка нов-car)</b>	; присвоить в поле CAR ячейки 154, 162, 163
<b>(RPLACD ячейка нов-cdr)</b>	; присвоить в поле CDR ячейки 154, 162, 163
<b>(NREVERSE посл)</b>	; перевернуть список физически
<b>(NCONC &amp;REST списки)</b>	; соединить списки физически
<b>(NBUTLAST список &amp;OPTIONAL n)</b>	; удалить из списка n последних элементов
<b>(PUSH элемент память)</b>	; поместить элемент в стек 292
<b>(PUSHNEW элемент память :TEST тест)</b>	; поместить новый элемент в стек
<b>(POP память)</b>	; взять элемент из стека

## 5 ФУНКЦИИ

- Описание функции

<b>(LAMBDA лямбда-список {декл:докум} {форма})</b>	; лямбда-выражение 105 ; необязательные описания и пояснения ; вычисляемые формы
--	--

- Ключевые слова лямбда-списка

<b>&amp;OPTIONAL</b>	; необязательные параметры 112, 296
<b>&amp;REST</b>	; переменное количество параметров 112, 113, 114, 117, 296
<b>&amp;AUX</b>	; вспомогательные переменные 112, 296
<b>&amp;KEY</b>	; ключевые параметры 112, 113, 115, 128, 296
<b>&amp;ALLOW-OTHER-KEYS</b>	; дополнительные аргументы
<b>(DEFUN имя лямбда-список {декл!докум} {форма})</b>	; определение функции
<b>(DEFSETF память-fn хранилище-fn [докум])</b>	; определение функции присваивания

- Вызов и применение

<b>((LAMBDA ...) ...)</b>	; лямбда-вызов 106
<b>(fn ...)</b>	; вызов функции 71, 240
<b>(FUNCTION функция-объект)</b>	; функциональная блокировка, формирование замыкания 259, 260, 261
<b>(APPLY fn arg &amp;REST аргументы)</b>	; применение функционального объекта 245, 246
<b>(FUNCALL fn &amp;REST аргументы)</b>	; вызов функционального объекта 245, 246

Многозначные функции описаны в управляющих структурах.

## 6 МАКРОСЫ

- Определение макроса

<b>(DEFMACRO имя лямбда-список {декл!докум} {форма})</b>	; определение макроса 290
--	---------------------------

- Ключевые слова определения макроса

<b>&amp;OPTIONAL</b>	; необязательные параметры 112, 296
<b>&amp;REST</b>	; переменное количество параметров 112, 113, 114, 117, 296
<b>&amp;AUX</b>	; вспомогательные переменные 112, 296
<b>&amp;KEY</b>	; ключевые параметры
<b>&amp;ALLOW-OTHER-KEYS</b>	; дополнительные аргументы
<b>&amp;BODY</b>	; (почти) то же, что &REST
<b>&amp;WHOLE</b>	; вся форма вызова
<b>&amp;ENVIRONMENT</b>	; контекст
<b>(DEFINE-MODIFY-MACRO</b>	; определение макроса для работы с

**имя лямбда-список fn [докум])** ; ячейкой памяти

- **Вызов и расширение**

**(макро-имя ...)** ; вызов макроса 290  
**(MACROEXPAND макровы- зов &OPTIONAL контекст)** ; расширение макроса 296

- **Тестирование**

**(MACRO-FUNCTION символ)** ; значение определения макроса 326

## 7 ОПИСАНИЯ

- **Локальные и глобальные описания**

**(DECLARE {определение})** ; локальное описание  
**(PROCLAIM определение)** ; глобальное описание

- **Область действия и тип переменных**

**Определения**  
**(SPECIAL {имя})** ; динамическая специальная переменная

Кроме того, используются оптимизирующие компиляцию описания, такие, как:

**(TYPE тип {имя})** ; описание типа символа  
**(FTYPE тип {имя-функции})** ; описание типа функции

- **Тип формы**

**(THE тип-значения форма)** ; описание типа формы

## 8 УПРАВЛЯЮЩИЕ СТРУКТУРЫ

- **Блокировка вычислений**

**(QUOTE форма)** ; блокировка вычислений 76, 128, 259  
**'форма** ; то же, что (QUOTE форма) 76, 260

<b>(FUNCTION fn)</b>	: функциональная блокировка, формирование замыкания 259, 260, 261
<b>#'fn</b>	; то же, что (FUNCTION fn) 260
<b>'основа</b>	; обратная блокировка 298
<b>,форма</b>	; замещающая отмена блокировки 299

**.@форма**

; присоединяющая отмена блокировки 299

#### • Создание контекста

<b>(LET присваивания {декл} {форма})</b>	: контекст, присваивания параллельные 129, 130
<b>(LET* присваивания {декл} {форма})</b>	; как LET, присваивания последовательные 131
<b>(COMPILER-LET присваивания форма)</b>	; присваивания, как в макросе
<b>(FUNCTION fn)</b>	; формирование замыкания, функциональная блокировка 259-261
<b>#'fn</b>	; то же, что (FUNCTION fn) 260

#### • Условные предложения

<b>(IF условие форма1 [форма2])</b>	; условное предложение 136
<b>(WHEN условие {форма})</b>	; вычисление формы, когда условие выполнено 136
<b>(UNLESS условие {форма})</b>	; вычисление формы, когда условие не выполнено 136
<b>(COND {(тест {следствие})})</b>	; многовариантное условное предложение 132, 133
<b>(CASE ключ {(тест {следствие})})</b>	; разветвление по ключу 136
<b>(TYPECASE ключ {(тип {форма})})</b>	; разветвление по типу ключа, см форму обработки ошибки ETYPECASE

#### • Последовательное вычисление

<b>(PROG1 форма1 {форма})</b>	; составная форма, значение – форма1 131
<b>(PROG2 форма1 форма2 {форма})</b>	; составная форма, значение – форма2 131
<b>(PROGN {форма})</b>	; составная форма, значение – формаN 131

#### • Циклические предложения

<b>(DO обновления окончание ;</b>	циклическое предложение с
-----------------------------------	---------------------------

<b>(DEKL {метка:форма})</b>	; переменными 137
<b>(DO* обновления окончание {декл} {метка:форма})</b>	; как DO, но присваивание переменным последовательное
<b>(DOLIST (var список [результат]) {декл} {метка:форма})</b>	; цикл над элементами 142
<b>(DOTIMES (var n [результат]) {декл})</b>	; цикл n раз 142
<b>(LOOP {форма})</b>	; бесконечный цикл 142

- Локальные и динамические переходы

<b>(BLOCK имя {форма})</b>	; именование блока для структурного выхода
<b>(RETURN-FROM имя [значение])</b>	; структурный выход из блока
<b>(RETURN [значение])</b>	; структурный выход со значением 140, 142
<b>(CATCH метка {форма})</b>	; поставить динамическую метку 145, 146
<b>(THROW метка значение)</b>	; вернуться к динамической метке 145, 146
<b>(UNWIND-PROTECT защищенная-форма {альтернативная-форма})</b>	; безусловное вычисление альтернативной формы в случае динамического прерывания

- Предложение PROG

<b>(PROG</b> $\{(\text{var}; (\text{var} [\text{значение}]))\}$ <b>{декл} {форма:метка})</b>	; PROG-механизм с метками перехода 140
<b>(PROG*</b> $\{(\text{var}; (\text{var} [\text{значение}]))\}$ <b>{декл} {форма:метка})</b>	; как PROG, но PROG-переменные вычисляются последовательно
<b>(GO метка)</b>	; переход на метку 140, 142

Возврат из фрагмента, см. выше формы RETURN и RETURN-FROM.

- Функциональные циклы

<b>MAP-функции для списков</b>	; см. отображающие функции для списков
<b>циклы над последовательностями</b>	; см. работу с последовательностями

- Многозначные значения

<b>(VALUES &amp;REST значения)</b>	; вернуть многозначное значение
<b>(VALUES-LIST список)</b>	; многозначное значение в виде списка
<b>(MULTIPLE-VALUE-LIST форма)</b>	; многозначное значение в качестве значения формы
<b>(MULTIPLE-VALUE-SETQ vars форма)</b>	; присваивание многозначного значения

## 9 ЛОГИЧЕСКИЕ ДЕЙСТВИЯ

- Логические константы

<b>T</b>	; логическое значение истина 64, 128
<b>NIL</b>	; логическое значение ложь 64, 128

- Логические действия

<b>(NOT объект)</b>	; логическое отрицание 91
<b>(AND {формы})</b>	; логическое И 135
<b>(OR {формы})</b>	; логическое ИЛИ 135

- Работа с битами

<b>(LOGIOR &amp;REST целые)</b>	; побитовое ИЛИ
<b>(LOGXOR &amp;REST целые)</b>	; побитовое исключающее ИЛИ
<b>(LOGAND &amp;REST целые)</b>	; побитовое И
<b>(LOGEQV &amp;REST целые)</b>	; побитовая эквивалентность
<b>(LOGNOT целое)</b>	; побитовое отрицание
<b>(LOGCOUNT целое)</b>	; количество бит в целом числе
<b>(INTEGER-LENGTH целое)</b>	; двоичный логарифм целого числа

## 10 МАТЕМАТИЧЕСКИЕ ФУНКЦИИ

- Числовые предикаты

<b>(ZEROP число)</b>	; проверка на ноль 320
<b>(PLUSP число)</b>	; проверка на положительность 320
<b>(MINUSP число)</b>	; проверка на отрицательность 320
<b>(ODDP целое)</b>	; проверка на нечетность 323
<b>(EVENP целое)</b>	; проверка на четность 323

- Сравнение чисел

<b>(= число &amp;REST числа)</b>	; равны (все) 88, 320
<b>(&lt; число &amp;REST числа)</b>	; меньше (для всех) 320
<b>(&gt; число &amp;REST числа)</b>	; больше (для всех) 320
<b>(&lt;= число &amp;REST числа)</b>	; меньше или равно (для всех) 320

(>= число &REST числа)	; больше или равно (для всех) 320
(/= число &REST числа)	; не равны (все) 320
(MAX число &REST числа)	; максимальное число 321
(MIN число &REST числа)	; минимальное число 321

#### • Арифметические действия

(+ &REST числа)	; сложение чисел 321
(- число &REST числа)	; вычитание чисел из числа 321
(* &REST числа)	; умножение чисел 321
(/ число &REST числа)	; деление числа на число 321
(ABS число)	; абсолютная величина числа 320
(- число)	; противоположное число 321
(SIGNUM число)	; знак числа -1, 0 или +1
(1+ число)	; добавление единицы к числу 321
(1- число)	; вычитание единицы из числа 321
(INCF память [делета])	; добавление приращения к числу в памяти 321
(DECFL память [делета])	; вычитание приращения из числа в памяти 321
(CONJUGATE число)	; дополнение комплексного числа
(GCD &REST n)	; наибольший общий делитель 323
(LCM n &REST n)	; наименьшее общее кратное 323

#### • Экспоненциальные и логарифмические функции

(EXP число)	; экспонента числа 322
(EXPT x y)	; x в степени y 139, 142, 322
(LOG число &OPTIONAL основание)	; логарифм числа 322
(SQRT число)	; квадратный корень из числа 322
(ISQRT целое)	; целый квадратный корень из целого числа

#### • Тригонометрические функции

PI	; значение числа $\pi$
(PHASE число)	; аргумент (комплексного) числа в радианах
(SIN угол)	; синус угла 322
(COS угол)	; косинус угла 322
(TAN угол)	; тангенс угла 322
(ASIN отношение)	; арксинус 322
(ACOS отношение)	; арккосинус 322
(ATAN отношение)	; арктангенс 322
(SINH угол)	; гиперболический синус 322
(COSH угол)	; гиперболический косинус 322
(TANH угол)	; гиперболический тангенс 322
(ASINH отношение)	; гиперболический арксинус 322

(ACOSH <i>отношение</i> )	; гиперболический арккосинус 322
(ATANH <i>отношение</i> )	; гиперболический арктангенс 322
▪ Преобразование чисел	
(FLOOR <i>число &amp;OPTIONAL делитель</i> )	; наибольшее целое меньшее данного
(CEILING <i>число &amp;OPTIONAL делитель</i> )	; наименьшее целое большее данного
(TRUNCATE <i>число &amp;OPTIONAL делитель</i> )	; обрубание до целого
(ROUND <i>число &amp;OPTIONAL делитель</i> )	; округление числа до целого
(FFLOOR <i>число &amp;OPTIONAL делитель</i> )	; как FLOOR, но число с плавающей запятой
(FCEILING <i>число &amp;OPTIONAL делитель</i> )	; как CEILING, но число с плавающей запятой
(FTRUNCATE <i>число &amp;OPTIONAL делитель</i> )	; как TRUNCATE, но число с плавающей запятой
(FROUND <i>число &amp;OPTIONAL делитель</i> )	; как ROUND, но число с плавающей запятой
(MOD <i>число делитель</i> )	; модуль числа
(REM <i>число делитель</i> )	; остаток от деления
▪ Случайные числа	
(RANDOM <i>число &amp;OPTIONAL состояние</i> )	; случайное число из промежутка 0...число 322
(MAKE-RANDOM-STATE <i>&amp;OPTIONAL состояние</i> )	; чтение состояния генератора случайных чисел
*RANDOM-STATE*	; состояние генератора случайных чисел по умолчанию

## 11 ЗНАКИ

### ▪ Анализ типов

(CHARACTERP <i>объект</i> )	; проверка на тип данных CHARACTER 338
(ALPHA-CHAR-P <i>знак</i> )	; проверка на букву
(DIGIT-CHAR-P <i>знак &amp;OPTIONAL основание</i> )	; проверка на цифру
(ALPHANUMERICP <i>знак</i> )	; проверка на алфавитно-цифровой знак

- Сравнение

- (CHAR= знак1 знак2)** ; сравнение знаков в алфавитном порядке (равенство)  
**(CHAR< знак1 знак2)** ; сравнение знаков в алфавитном порядке (раньше)  
**(CHAR> знак1 знак2)** ; сравнение знаков в алфавитном порядке (после)

- Преобразования

- (CHARACTER объект)** ; преобразование объекта в знак, если возможно  
**(CHAR-UPCASE знак)** ; преобразование строчной буквы в прописную  
**(CHAR-DOWNCASE знак)** ; преобразование прописной буквы в строчную  
**(CHAR-INT знак)** ; числовое значение знака  
**(INT-CHAR числ-значение)** ; знак, соответствующий числовому значению  
**(CHAR-NAME знак)** ; имя знака или nil  
**(NAME-CHAR имя-знака)** ; соответствующий имени знак или nil  
**(CHAR-BIT знак битовое-имя)** ; определение битового имени знака  
**(SET-CHAR-BIT знак имя значение)** ; присваивание битового имени знаку

## 12 СТРОКИ

- (STRINGP объект)** ; является ли объект строкой 338

- Селекторы

- (CHAR строка n)** ; n-й знак строки 339  
**(SCHAR n-строка n)** ; n-й знак простой строки

- Сравнение

- (STRING= строка1 строка2 &KEY :START1 :END1 :START2 :END2)** ; одинаковые (под)строки  
**(STRING< строка1 строка2 &KEY ключи)** ; возрастание в коде ASCII  
**(STRING> строка1 строка2 &KEY ключи)** ; убывание в коде ASCII  
**(STRING/= строка1 строка2 &KEY ключи)** ; неравенство в коде ASCII

- Алфавитное сравнение

<b>(STRING-EQUAL</b> строка1 строка2 &KEY :START1 :END1 :START2 :END2)	; равный в алфавитном порядке
<b>(STRING-LESSP</b> строка1 строка2 &KEY ключи)	; возрастающий в алфавитном порядке
<b>(STRING-GREATERP</b> строка1 строка2 &KEY ключи)	; убывающий в алфавитном порядке
<b>(STRING-NOT-EQUAL</b> строка1 строка2 &KEY ключи)	; неравный в алфавитном порядке
<b>(STRING-NOT-LESSP</b> строка1 строка2 &KEY ключи)	; невозрастающий в алфавитном порядке
<b>(STRING-NOT-GREATERP</b> строка1 строка2 &KEY ключи)	; неубывающий в алфавитном порядке

- Преобразования

<b>(STRING</b> объект)	; преобразовать объект в строку 328, 339
<b>(STRING-UPCASE</b> строка)	; заменить строчные буквы на прописные
<b>(STRING-DOWNCASE</b> строка)	; заменить прописные буквы на строчные
<b>(STRING-CAPITALIZE</b> строка)	; заменить первые буквы на прописные
<b>(NSTRING-UPCASE</b> строка)	; преобразовать строчные буквы в прописные
<b>(NSTRING-DOWNCASE</b> строка)	; преобразовать прописные буквы в строчные
<b>(NSTRING-CAPITALIZE</b> строка)	; преобразовать первые буквы в прописные
<b>(STRING-TRIM</b> знаки строка)	; удалить знаки из начала и конца строки
<b>(STRING-LEFT-TRIM</b> знаки строка)	; удалить знаки из начала строки
<b>(STRING-RIGHT-TRIM</b> зна- ки строка)	; удалить знаки из конца строки

## 13 ПОСЛЕДОВАТЕЛЬНОСТИ

<b>(LENGTH</b> поса)	; длина последовательности 147
----------------------	--------------------------------

- Формирование последовательности

(MAKE-SEQUENCE <i>тип</i> размер &KEY :INITIAL- ELEMENT)	; создать новую последовательность 343
(CONCATENATE <i>тип</i> &REST <i>посл-ти</i> )	; скопировать и объединить последо- вательности
(COPY-SEQ <i>посл</i> )	; скопировать последовательность
(SUBSEQ <i>посл</i> начало &OPTIONAL конец)	; скопировать подпоследовательность последовательности

У подтипов последовательности LIST, VECTOR и STRING существует большое число своих функций.

- Предикат

(EVERY <i>условие посл</i> &REST <i>посл-ти</i> )	; проверка условия для всех элемен- тов 345
(SOME <i>условие посл</i> &REST <i>посл-ти</i> )	; проверка условия для некоторого элемента 345
(NOTEVERY <i>условие посл</i> &REST <i>посл-ти</i> )	; логическое отрицание предиката EVERY
(NOTANY <i>условие посл</i> &REST <i>посл-ти</i> )	; логическое отрицание предиката SOME

- Чтение и поиск элементов

(ELT <i>посл</i> <i>н</i> )	; n-й элемент последовательности 343
(SUBSEQ <i>посл</i> начало &OPTIONAL конец)	; подпоследовательность последова- тельности 344
(FIND <i>начало посл</i> &KEY ключи)	; поиск элемента в последовательности 345
(FIND-IF <i>условие посл</i> &KEY ключи)	; поиск элемента в соответствии с условием
(FIND-IF-NOT <i>условие посл</i> &KEY ключи)	; поиск элемента, для которого усло- вие не выполнено
(POSITION <i>начало посл</i> &KEY ключи)	; позиционирование на первое вхождение элемента
(POSITION-IF <i>условие посл</i> &KEY ключи)	; позиционирование на первое вхождение элемента в соответствии с условием
(POSITION-IF-NOT условие посл &KEY ключи)	; позиционирование на первое вхождение элемента, для которого условие не выполнено
(COUNT <i>начало посл</i> &KEY ключи)	; количество вхождений элемента в последовательность
(COUNT-IF <i>условие посл</i>	; количество элементов, для которых

<b>&amp;KEY</b> ключи)	; выполнено условие
<b>(COUNT-IF-NOT</b> условие посл <b>&amp;KEY</b> ключи)	; количество элементов, для которых не выполнено условие
<b>(MISMATCH</b> посл1 посл2 <b>&amp;KEY</b> ключи)	; позиционировать на первое отличие
<b>(SEARCH</b> подпосл посл <b>&amp;KEY</b> ключи)	; позиционировать на первое вхожде- ние подпоследовательности

▪ Преобразования

<b>(REMOVE</b> элемент посл <b>&amp;KEY</b> ключи)	; удаление вхождения элемента 216, 345, 346
<b>(REMOVE-IF</b> условие посл <b>&amp;KEY</b> ключи)	; удаление на основе условия
<b>(REMOVE-IF-NOT</b> условие посл <b>&amp;KEY</b> ключи)	; удаление элемента, если условие не выполнено
<b>(REMOVE-DUPLICATES</b> посл <b>&amp;KEY</b> ключи)	; удаление повторяющихся элементов
<b>(DELETE</b> элемент посл <b>&amp;KEY</b> ключи)	; удаление элементов, изменяя последовательность
<b>(DELETE-IF</b> условие посл <b>&amp;KEY</b> ключи)	; удаление элементов на основе усло- вия
<b>(DELETE-IF-NOT</b> условие посл <b>&amp;KEY</b> ключи)	; удаление элементов, если условие не выполнено
<b>(DELETE-DUPLICATES</b> посл <b>&amp;KEY</b> ключи)	; удаление повторяющихся элементов
<b>(SUBSTITUTE</b> нов стар посл <b>&amp;KEY</b> ключи)	; замена вхождения элемента на новое
<b>(SUBSTITUTE-IF</b> нов усло- вие посл <b>&amp;KEY</b> ключи)	; замена вхождения элемента на основе условия
<b>(SUBSTITUTE-IF-NOT</b> нов условие посл <b>&amp;KEY</b> ключи)	; замена вхождения элемента, если условие не выполнено
<b>(NSUBSTITUTE</b> нов стар посл <b>&amp;KEY</b> ключи)	; присвоить вхождению элемента
<b>(NSUBSTITUTE-IF</b> нов тест посл <b>&amp;KEY</b> ключи)	; присвоить элементу на основе усло- вия
<b>(NSUBSTITUTE-IF-NOT</b> нов тест посл <b>&amp;KEY</b> ключи)	; присвоить элементу, если условие не выполнено
<b>(FILL</b> посл элемент <b>&amp;KEY</b> ключи)	; заполнить последовательность эле- ментом
<b>(REPLACE</b> посл1 посл2 <b>&amp;KEY</b> ключи)	; заменить часть последовательности на другую
<b>(MAP</b> тип fn посл &REST посл-ти)	; применить функцию к последова- тельности 255, 344
<b>(REDUCE</b> fn посл &KEY ключи))	; применить функцию каскадно

- Сортировка

<b>(SORT</b> посл условие &KEY ключ)	; отсортировать последовательность 348
<b>(STABLE-SORT</b> посл условие &KEY ключ)	; отсортировать последовательность на месте

## 14 МАССИВЫ

- Создание массива

<b>(MAKE-ARRAY</b> размерности &KEY :ELEMENT-TYPE :INITIAL-ELEMENT :INITIAL-CONTENTS :ADJUSTABLE :FILL-POINTER)	, создание массива 349
<b>(VECTOR</b> &REST элементы)	; тип элемента
<b>(ADJUST-ARRAY</b> массив новые-размерности &KEY ключи)	; начальное значение
	; начальная последовательность
	; динамический массив
	; указатель заполненности массива
	; создание одномерного массива
	; изменение вида массива

- Чтение и присваивание

<b>(AREF</b> массив &REST индексы)	; взятие элемента массива 350
<b>(SVREF</b> п-вектор индексы)	; взятие элемента простого вектора
<b>(BIT</b> массив-знаков &REST n)	; элемент битового массива
<b>(SBIT</b> n-массив-знаков &REST n)	; элемент простого битового массива
<b>(VECTOR-PUSH</b> элемент массив)	; занесение в массив следующего элемента
<b>(VECTOR-POP</b> массив)	; взятие текущего элемента из массива

- Анализ вида массива

<b>(ARRAYP</b> объект)	; проверка на массив 86
<b>(VECTORP</b> объект)	; проверка на одномерный массив
<b>(ADJUSTABLE-ARRAY-P</b> массив)	; проверка на динамический массив
<b>(ARRAY-ELEMENT-TYPE</b> массив)	; проверка на тип элемента массива
<b>(ARRAY-DIMENSIONS</b> массив)	; размерность массива
<b>(ARRAY-HAS-FILL-POINT</b>	; проверка на наличие указателя за-

<b>ER-P массив</b> <b>(FILL-POINTER вектор)</b>	; полненности массива ; указатель заполненности массива
<b>• Хэш-массивы</b>	
<b>(MAKE-HASH-TABLE &amp;KEY ключи)</b>	; создание хэш-массива
<b>(HASH-TABLE-P объект)</b>	; проверка на хэш-массив
<b>(GETHASH ключ таблица &amp;OPTIONAL умолчание)</b>	; взятие элемента, связанного с ключом
<b>(REMHASH ключ таблица)</b>	; удаление ключа и значения из массива
<b>(MAPHASH fn таблица)</b>	; применить функционал к содержимому массива
<b>(CLRHASH таблица)</b>	; очистить массив
<b>(HASH-TABLE-COUNT таблица)</b>	; количество элементов массива
<b>(SXHASH объект)</b>	; вычислить хэш-функцию объекта

## 15 СТРУКТУРЫ

### • Определение структур

<b>(DEFSTRUCT имя-типа- структурь [докум] {описание-поля})</b>	; новый тип структуры 306, 316, 354
--	-------------------------------------

### • Описание полей

<b>:TYPE</b>	; тип поля 355
<b>:READ-ONLY</b>	; полю нельзя присваивать 355

### • Функции, генерируемые системой

<b>(MAKE-тип [поле значение])</b>	; создать новую структуру соответствующего типа 355
<b>(COPY-тип структура)</b>	; копирование структуры 355
<b>(тип-Р объект)</b>	; предикат, проверяющий тип структуры 355

**(тип-поле структура)** ; взятие значения поля 355

Полю можно присваивать, и с ним можно работать как с ячейкой памяти.

## 16 ВВОД И ВЫВОД

- Ввод

Описания параметров следующих далее функций чтения несколько сокращены, особенно в части обработки конца файла (eof) (см. справочное руководство по системе).

<b>(READ &amp;OPTIONAL поток)</b>	; чтение выражения из входного потока 175, 176, 189
<b>(READ-LINE &amp;OPTIONAL поток)</b>	; чтение следующей строки
<b>(READ-CHAR &amp;OPTIONAL поток)</b>	; чтение следующего знака
<b>(READ-CHAR-NO-HANG &amp;OPTIONAL поток)</b>	; чтение знака, если он есть
<b>(UNREAD-CHAR знак &amp;OPTIONAL поток)</b>	; возвращение знака во входной поток
<b>(PEEK-CHAR &amp;OPTIONAL способ поток)</b>	; подглядывание следующего знака
<b>(LISTEN &amp;OPTIONAL поток)</b>	; проверка, есть ли во входном потоке знаки
<b>(CLEAR-INPUT &amp;OPTIONAL поток)</b>	; стирание потока
<b>(READ-FROM-STRING строка &amp;OPTIONAL опции &amp;KEY ключи)</b>	; чтение из строки

- Вывод

<b>(WRITE объект &amp;KEY ключи)</b>	; общая функция вывода
<b>(PRIN1 объект &amp;OPTIONAL поток)</b>	; вывести объект на ту же строку 181, 189
<b>(PRINT объект &amp;OPTIONAL поток)</b>	; вывести перевод строки, объект и пробел 180, 184, 189
<b>(PPRINT объект &amp;OPTIONAL поток)</b>	; структурная печать
<b>(PRINC объект &amp;OPTIONAL поток)</b>	; символьная печать 181, 189
<b>(TERPRI &amp;OPTIONAL поток)</b>	; вывести перевод строки 183, 189
<b>(FRESH-LINE &amp;OPTIONAL поток)</b>	; проверка на начало строки

- Функция FORMAT

<b>(FORMAT поток образец &amp;REST аргументы)</b>	; сформировать выводимое значение 184
---	---------------------------------------

Используемые в образце директивы имеют следующий общий вид:  
 $\sim p1, p2, \dots, pN:@x$

Здесь  $p1$  – необязательные параметры,  $x$  – управляющий код и @ – необязательные модификаторы управляющего кода.

Специальными параметрами являются:

Параметры	Интерпретация
V	; очередной аргумент вызова FORMAT
#	; количество оставшихся аргументов

Далее приведен перечень наиболее важных управляющих кодов. Необязательные параметры в некоторых случаях из-за нехватки места обозначены многоточием "...". Некоторые варианты использования модификаторов ":" и "@" для некоторых управляющих кодов не упомянуты. Сошлемся на справочное руководство.

#### Вывод

~...A	; вывод объекта функцией PRINC 185
~...S	; вывод объекта функцией PRIN1 185
~n~	; вывод тильды n раз 185
~...<m1~; m2; ...~>	; вывод и выравнивание сегментов текста
~...:<m1~; m2; ...~>	; – выравнивание только в начале
~...@<m1~; m2; ...~>	; – выравнивание промежутков и конца
~...@:<m1~; m2; ...~>	; – выравнивание начала, промежутков и конца

#### Формат вывода чисел

~...D	; десятичный вывод
~...B	; двоичный вывод
~...O	; восьмеричный вывод
~...X	; шестнадцатеричный вывод
~...G	; наиболее общая форма вывода чисел с плавающей запятой
~...F	; вывод в форме с плавающей запятой
~...E	; вывод с экспонентой
~...\$	; вывод в денежных единицах
~...R	; вывод числа словами
~...:R	; вывод порядкового числа словами (на англ.)
~...@R	; вывод числа в римской системе

## Знаки и строки

<code>~C</code>	; вывод знака в сокращенной форме
<code>~:C</code>	; вывод знака в форме слова
<code>~(строка~)</code>	; заменить прописные буквы на строчные
<code>~:(строка~)</code>	; заменить первые буквы слов на прописные
<code>~@(строка~)</code>	; заменить первую букву на прописную
<code>~@:(строка~)</code>	; заменить все буквы на прописные
<code>~P</code>	; вывести слово во множественном числе (на англ.)

## Управление заполнением

<code>~n:</code>	; вывод n переводов страниц
<code>~n%</code>	; вывод n переводов строк 185
<code>~n&amp;</code>	; вывести при необходимости n переводов строк
<code>~&lt;newline&gt;</code>	; перенос управляющей строки на новую строку
<code>~n,mT</code>	; абсолютная табуляция 185
<code>~n,m@T</code>	; относительная табуляция

## Управление выводом

<code>~n*</code>	; перейти на n аргументов вперед
<code>~n:*</code>	; перейти на n элементов назад
<code>~n@*</code>	; перейти к n-ому элементу
<code>~{строка~}</code>	; повторный вывод
<code>~{строка~}</code>	; повторить вывод для последовательности аргументов
<code>~n{строка~}</code>	; вывести n раз
<code>~:{строка~}</code>	; повторное применение к подспискам
<code>~:\${строка~}</code>	; применить подсписки к последовательности аргументов
<code>~:{ложь~;истина~}</code>	; условный вывод
<code>~{m0~;m1~;..~}</code>	; многовариантный условный вывод
<code>~?</code>	; рекурсивный вывод
<code>~@?</code>	; рекурсивное применение последовательности
<code>~^</code>	; закончить охватывающую управляющую структуру

- Запрос да/нет

**(YES-OR-NO-P  
 &OPTIONAL *тело*  
 &REST *аргументы*)** ; да/нет запрос пользователю

## 17 ПОТОКИ

**\*STANDARD-INPUT\*** ; поток ввода по умолчанию 64, 187  
**\*STANDARD-OUTPUT\*** ; поток вывода по умолчанию 187

- Предикат

<b>(STREAMP <i>объект</i>)</b>	; проверить, является ли объект потоком
<b>(INPUT-STREAM-P <i>поток</i>)</b>	; проверить, открыт ли поток для ввода
<b>(OUTPUT-STREAM-P  <i>поток</i>)</b>	; проверить, открыт ли поток для вывода
<b>(STREAM-ELEMENT-TYPE  <i>поток</i>)</b>	; тип элементов в потоке

- Создание и закрытие потока

<b>(WITH-OPEN-STREAM (<i>не-</i>  <i>ременная поток</i>)  <i>{декл}</i> {<i>форма</i>})</b>	; форма открывающая и закрывающая поток
<b>(MAKE-TWO-WAY-    STREAM <i>in out</i>)</b>	; создание двунаправленного потока
<b>(MAKE-SYNONYM-    STREAM <i>символ</i>)</b>	; создание синонима потока
<b>(MAKE-ECHO-STREAM <i>in    out</i>)</b>	; создание двунаправленного потока с эхом
<b>(MAKE-BROADCAST-    STREAM &amp;REST  <i>потоки</i>)</b>	; создание потока сообщений
<b>(MAKE-CONCATENATED-    STREAM &amp;REST  <i>потоки</i>)</b>	; соединить потоки ввода
<b>(CLOSE <i>поток &amp;KEY    :ABORT</i>)</b>	; закрыть поток 189

- Преобразование потоков

<b>(MAKE-STRING-INPUT-    STREAM <i>строка</i>    &amp;OPTIONAL <i>начало    конец</i>)</b>	; создать поток из (под)строки
<b>(GET-OUTPUT-STREAM-  <i>строка</i>)</b>	; этот вызов выдает строку

**STRING (MAKE-  
STRING-OUTPUT-  
STREAM))**

выводного потока

## 18 РАБОТА С ФАЙЛАМИ

- Имя пути и имя файла

(PATHNAMEР <i>объект</i> )	; является ли объект именем пути
(MAKE-PATHNAME &KEY	; создать абсолютный путь
:HOST	; система файлов
:DEVICE	; устройство
:DIRECTORY	; директорий
:NAME	; файл
:TYPE	; тип файла
:VERSION	; версия
:DEFAULTS)	; имя пути
(PATHNAME <i>изображение- имени-пути</i> )	; имя пути, соответствующее изобра- жению
(TRUENAME <i>имя-пути</i> )	; преобразовать имя пути в имя файла
(NAMESTRING <i>имя-пути</i> )	; преобразовать имя пути в строку

- Открытие и закрытие файла

(WITH-OPEN-FILE ( <i>поток</i> <i>имя {опция} {декл}</i> <i>{форма}</i> )	; создать поток для файла и закрыть его 189
(OPEN <i>файл &amp;KEY</i> :DIRECTION :ELEMENT-TYPE :IF-EXISTS :IF-DOES-NOT-EXIST)	; открыть поток для файла 188 ; направление ; тип элемента потока ; форма ; форма
(CLOSE <i>поток &amp;KEY</i> :ABORT)	; закрыть поток 189

- Действия с файлами

(RENAME-FILE <i>стар нов</i> )	; переименовать файл
(DELETE-FILE <i>имя</i> )	; удалить файл
(PROBE-FILE <i>файл</i> )	; проверить, существует ли файл
(DIRECTORY <i>имя-пути</i> &KEY)	; путь директории
(FILE-LENGTH <i>файл</i> )	; размер файла
(FILE-WRITE-DATE <i>файл</i> )	; время последнего обновления файла
(FILE-AUTHOR <i>файл</i> )	; автор файла

- Загрузка файлов и модулей

<b>(LOAD</b> <i>файл &amp;KEY</i> <i>ключи</i> )	; загрузить программный файл 190
<b>(REQUIRE</b> <i>модуль</i> )	; загрузить модуль при необходимости
<b>(PROVIDE</b> <i>модуль</i> )	; пометить модуль как загруженный

## 19 СРЕДА ПРОГРАММИРОВАНИЯ

- Интерпретатор

<b>(EVAL</b> <i>форма</i> )	; вычисление формы, интерпретатор Лиспа 100, 101, 245
<b>(APPLY</b> <i>fn arg &amp;REST аргументы</i> )	; применение функционального объекта 245, 246
<b>(FUNCALL</b> <i>fn arg &amp;REST аргументы</i> )	; вызов функционального объекта 245, 246
<b>(EVALHOOK</b> <i>форма eval-захват apply-захват &amp;OPTIONAL контекст</i> )	; назначение функций захвата
<b>(IDENTITY</b> <i>объект</i> )	; сам объект
<b>(SLEEP</b> <i>n</i> )	; приостановка вычислений на <i>n</i> секунд

- Компилятор

<b>(COMPILE</b> <i>имя &amp;OPTIONAL определение</i> )	; компиляция функции
<b>(COMPILE-FILE</b> <i>имя-путь-ввода &amp;KEY :OUTPUT-FILE</i> )	; компиляция файла

- Редактор

<b>(ED &amp;OPTIONAL</b> <i>объект</i> )	; вызов редактора
--	-------------------

- Комментарии и пояснения

<b>;</b>	; комментарий в конце строки 178
<b>;;</b>	; вложенный комментарий 178
<b>;;;</b>	; внешний комментарий 178
<b>(DOCUMENTATION</b> <i>символ тип-докум</i> )	; документация символа

Форма определения

Тип документации

**DEFVAR**

**VARIABLE**

<b>DEFPARAMETR</b>	<b>VARIABLE</b>
<b>DEFCONSTANT</b>	<b>VARIABLE</b>
<b>DEFUN</b>	<b>FUNCTION</b>
<b>DEFMACRO</b>	<b>FUNCTION</b>
<b>DEFSTRUCT</b>	<b>STRUCTURE</b>
<b>DEFTYPE</b>	<b>TYPE</b>
<b>DEFSETF</b>	<b>SETF</b>
<b>(DESCRIBE объект)</b>	; получить описание объекта
<b>(INSPECT объект)</b>	; анализ объекта
<b>(APROPOS строка &amp;OPTIONAL пр-имен)</b>	; сведения об объектах по имени (в пространстве имен)
<b>(APROPOS-LIST строка &amp;OPTIONAL пр-имен)</b>	; перечислить имена объектов (в пространстве имен)
<b>• Отслеживание вычислений и статистика</b>	
<b>(TRACE {имя-функции})</b>	; включить трассировку 206
<b>(UNTRACE {имя-функции})</b>	; выключить трассировку 206
<b>(STEP форма)</b>	; вычислить форму в пошаговом режиме
<b>• Ход диалога</b>	
*	; последний результат
**	; предпоследний результат
***	; предпредпоследний результат
+	; последнее введенное выражение
++	; предпоследнее введенное выражение
+++	; предпредпоследнее введенное выражение
/	; последнее выведенное выражение
//	; предпоследнее выведенное выражение
///	; предпредпоследнее выведенное выражение
-	; вычисляемая форма
<b>(DRIBBLE &amp;OPTIONAL имя-пути)</b>	; запись протокола диалога в файл
<b>• Использование времени и памяти</b>	
<b>(TIME форма)</b>	; вычисление формы и вывод затраченного времени
<b>(ROOM &amp;OPTIONAL)</b>	; сведения об использовании памяти
<b>(GET-UNIVERSAL-TIME)</b>	; текущее время
<b>• Прерывания и обработка ошибок</b>	

(BREAK &OPTIONAL фрейм &REST аргументы)	; прерывание
(CHECK-TYPE память отоб- ражение-типа &OPTIONAL строка)	; проверка типа значения в ячейке памяти
(ERROR фрейм &REST аргументы)	; вызов ошибки и выдача сообщения об ошибке в ситуации, не допуска- ющей продолжения вычислений
(CERROR фрейм-продолже- ния фрейм-ошибки &REST аргументы)	; вызов ошибки и выдача сообщения об ошибке в ситуации, допуска- ющей продолжение вычислений
(ETYPECASE форма-ключ ({тип {форма}}))	; проверка типа и выдача сообщения, ср. с условным TYPECASE (управ- ляющие структуры)

## **ПРИЛОЖЕНИЕ 2**

### **УКАЗАТЕЛЬ СИМВОЛОВ КОММОН ЛИСПА**

Указатель содержит разбитые на логические группы, упорядоченные в алфавитном порядке имена функций и форм, ключевые слова, признаки, системные переменные, а также специальные знаки и способы обозначения. С каждым элементом связано описание, отражающее значение и основные характеристики применения символа. Номера страниц относятся к тексту и сводке Коммон Лиспа (приложение 1). Для полноты в указатель также включены слова, не встречающиеся в книге.

#### **Формы и типы**

- ABS**, абсолютная величина 320
- ACONS**, а-список, изменение 334
- ACOS**, аркосинус 322
- ACOSH**, гиперболический аркосинус 322
- ADJOIN**, добавить элемент, множество 394
- ADJUSTABLE-ARRAY-P**, тип массива, предикат 407
- ADJUST-ARRAY**, изменение вида массива 407
- ALPHANUMERICP**, алфавитно-цифровой знак, предикат
- ALPHA-CHAR-P**, буква, предикат
- AND**, логическое И 135
- APPEND**, соединение списков 164, 165, 214, 216
- APPLY**, применение функционального объекта 245, 246
- APROPOS**, сведения о символах
- APROPOS-LIST**, поиск символов
- AREF**, элемент массива 350
- ARRAY**, массив, тип 317, 349
- ARRAYP**, массив, предикат 86
- ARRAY-DIMENSIONS**, размерность массива
- ARRAY-ELEMENT-TYPE**, тип элемента массива
- ARRAY-HAS-FILL-POINTER-P**, указатель заполненности
- ASET**, присваивание элементу массива
- ASIN**, арксинус 322
- ASINH**, гиперболический арксинус 322
- ASSOC**, а-список, поиск 333, 334
- ASSOC-IF**, а-список, поиск
- ASSOC-IF-NOT**, а-список, поиск
- ATAN**, арктангенс 322

**ATANH**, гиперболический арктангенс 322  
**ATOM**, атомарный объект, предикат 79  
**ATOMICP**, атомарный объект, предикат

**BIGNUM**, целое неограниченной точности 323  
**BIT**, тип данных  
**BIT-VECTOR**, битовый вектор 317  
**BLOCK**, именование блока  
**BOUNDP**, наличие связи, предикат 98, 327  
**BREAK**, прерывание  
**BUTLAST**, список без последних элементов

**C . R**, композиция CAR и CDR 91  
**CAR**, головная часть списка, поле 79, 80, 81, 84, 91, 151  
**CASE**, разветвление по ключу 136  
**CATCH**, перехват 145, 146  
**CDR**, хвостовая часть списка, поле 79, 81, 84, 91, 151  
**CEILING**, наименьшее целое большее данного  
**CERROR**, вызов ошибки и выдача сообщения  
**CHAR**, знак строки 339  
**CHAR<**, сравнение знаков в алфавитном порядке (раньше)  
**CHAR=/**, сравнение знаков в алфавитном порядке (неравенство)  
**CHAR=**, сравнение знаков в алфавитном порядке (равенство)  
**CHAR>**, сравнение знаков в алфавитном порядке (после)  
**CHARACTER**, тип строки 315, 337  
**CHARACTERP**, проверка на тип данных CHARACTER 338  
**CHAR-BIT**, битовое имя знака  
**CHAR-DOWNCASE**, прописные буквы в строчные  
**CHAR-INT**, числовое значение знака  
**CHAR-NAME**, имя знака или nil  
**CHAR-UPCASE**, строчные буквы в прописные  
**CHECK-TYPE**, проверка типа значения  
**CLEAR-INPUT**, стирание потока  
**CLOSE**, закрытие потока 189  
**CLRHASH**, очистить массив  
**COERCE**, преобразование типа 314, 328, 339  
**COMMON**, надтип типов Коммон Лиспа 317  
**COMPILE**, компиляция функции  
**COMPILED-FUNCTION**, компилированная функция, тип  
**COMPILER-LET**, присваивание как в макросе  
**COMPILE-FILE**, компиляция файла  
**COMPLEX**, комплексное число, тип 316, 323  
**CONCATENATE**, объединение последовательностей  
**COND**, условное предложение 132, 133  
**CONJUGATE**, дополнение комплексного числа  
**CONS**, создание списочной ячейки 79, 83, 84, 151, 155  
    точечная пара, тип 156, 315, 332  
**CONSP**, списочная ячейка, предикат  
    точечная пара, тип

- COPY-TYPE**, копирование структуры 355  
**COPY-LIST**, копирование списка 208, 225, 271  
**COPY-SEQUENCE**, копирование последовательности  
**COPY-SYMBOL**, копирование символа  
**COPY-TREE**, копирование списочной структуры 225, 226  
**COS**, косинус угла 322  
**COSH**, гиперболический косинус 322  
**COUNT**, количество элементов  
**COUNT-IF**, количество элементов  
**COUNT-IF-NOT**, количество элементов
- DEFCAF**, вычитание приращения из числа в памяти 321  
**DECLARE**, локальное описание  
**DEFCONSTANT**, определение константы  
**DEFINE-MODIFY-MACRO**, определение макроса работы с памятью  
**DEFMACRO**, определение макроса 290  
**DEFPARAMETER**, глобальная переменная-параметр  
**DEFSETF**, определение функции присваивания  
**DEFSTRUCT**, новый тип структуры 306, 316, 354  
**DEFTYPE**, определение нового типа  
**DEFUN**, определение функции  
**DEFVAR**, определение динамической переменной 121, 122  
**DELETE**, удаление элементов  
**DELETE-DUPLICATES**, удаление повторяющихся элементов  
**DELETE-FILE**, удалить файл  
**DELETE-IF**, удаление элементов  
**DELETE-IF-NOT**, удаление элементов  
**DESCRIBE**, получить описание объекта  
**DIGIT-CHAR-P**, проверка на цифру  
**DIRECTORY**, директорий  
**DO**, циклическое предложение 137  
**DO\***, циклическое предложение, присваивание последовательное  
**DOCUMENTATION**, документация символа  
**DOLIST**, цикл по списку 142  
**DOTIMES**, цикл 142  
**DOUBLE-FLOAT**, число с плавающей запятой, двойная точность 324  
**DRIBBLE**, запись протокола диалога в файл
- ED**, вызов редактора  
**EIGHTH**, восьмой элемент списка  
**ELT**, элемент последовательности 343  
**ENDP**, проверка конца списка  
**EQ**, физическое равенство (указателей) 79, 86, 87, 154  
**EQL**, совпадение значения и типа 88  
**EQUAL**, логическая идентичность объектов 89, 154, 226  
**EQUALP**, совпадение без учета типов 90  
**ERROR**, вызов ошибки и выдача сообщения  
**ETYPECASE**, проверка типа и выдача сообщения

**EVAL**, вычисление формы, интерпретатор Лиспа 100, 101, 245  
**EVALHOOK**, назначение функций захвата  
**EVENP**, проверка на четность 323  
**EVERY**, проверка условия для всех элементов 345  
**EXP**, экспонента числа 322  
**EXPT**, степень числа 139, 142, 322

**FBOUNDP**, функциональная связь, предикат 110, 327  
**FCEILING**, CEILING с плавающей запятой  
**FFLOOR**, FLOOR с плавающей запятой  
**FIFTH**, пятый элемент списка  
**FILE-AUTHOR**, автор файла  
**FILE-LENGTH**, размер файла  
**FILE-WRITE-DATE**, время последнего обновления файла  
**FILL**, заполнить последовательность элементом  
**FILL-POINTER**, указатель заполненности массива  
**FIND**, поиск в последовательности 345  
**FIND-IF**, поиск  
**FIND-IF-NOT**, поиск  
**FIRST**, первый элемент списка  
**FIXNUM**, целое ограниченной точности, тип 323  
**FLOAT**, число с плавающей запятой, тип 316, 323  
**FLOOR**, наибольшее целое меньшее данного  
**FMAKUNBOUND**, удаление функциональной связи  
**FORMAT**, сформировать выводимое значение 184  
**FOURTH**, четвертый элемент списка  
**FRESH-LINE**, начало строки, предикат  
**FROUND**, ROUND с плавающей запятой  
**FTRUNCATE**, TRUNCATE с плавающей запятой  
**FTYPE**, описание типа функции  
**FUNCALL**, вызов функционального объекта 245, 246  
**FUNCTION**, функциональная блокировка 259, 260, 261  
**FUNCTIONP**, функциональный объект, тип

**GCD**, наибольший общий делитель 323  
**GENSYM**, создание символа 330  
**GENTEMP**, создание символа 330  
**GET**, значение свойства символа 169, 170  
**GETF**, чтение свойства из списка свойств  
**GETHASH**, взятие элемента, хеш-массив  
**GET-OUTPUT-STREAM-STRING**, строка выводного потока  
**GET-UNIVERSAL-TIME**, текущее время  
**GO**, переход на метку 140, 142

**HASH-TABLE**, хэш-массив, тип 317, 351  
**HASH-TABLE-COUNT**, количество элементов массива  
**HASH-TABLE-P**, проверка на хэш-массив

**IDENTITY**, сам объект

- IF, условное предложение 136  
INCF, добавление приращение к числу в памяти 321  
INPUT-STREAM-P, поток для ввода, предикат  
INSPECT, анализ объекта  
INTEGER, целое число, тип 316, 323  
INTEGER-LENGTH, двоичный логарифм целого числа  
INTERN, включение символа в пространство имен 329  
INTERSECTION, пересечение множеств  
INT-CHAR, знак, соответствующий числовому значению  
ISQRT, целый квадратный корень из целого числа  
  
KEYWORDP, проверка на ключевое слово  
  
LAMBDA, лямбда-выражение 105  
LAST, последняя точечная пара 92  
LCM, наименьшее общее кратное 323  
LENGTH, длина последовательности 147  
LET, локальные присваивания 129, 130  
LET\*, последовательные присваивания 131  
LIST, формирование списка из элементов 93, 105, 114  
    список, тип 200, 315, 317, 331  
LIST\*, формирование точечного списка  
LISTEN, проверка, есть ли во входном потоке знаки  
LISTP, список или nil, предикат 86  
LIST-LENGTH, длина списка  
LOAD, загрузить программный файл 190  
LOG, логарифм числа 322  
LOGAND, побитовое И  
LOGCOUNT, количество битов в целом числе  
LOGEQV, побитовая эквивалентность  
LOGIOR, побитовое ИЛИ  
LOGNOT, побитовое отрицание  
LOGXOR, побитовое исключающее ИЛИ  
LONG-FLOAT, длинное число с плавающей запятой 324  
LOOP, бесконечный цикл 142  
  
MACROEXPAND, расширение макроса 296  
MACRO-FUNCTION, значение определения макроса 326  
MAKE-type, создание структуры 355  
MAKE-ARRAY, создание массива 349  
MAKE-BROADCAST-STREAM, создание потока сообщений  
MAKE-CONCATENATED-STREAM, соединение потоков ввода  
MAKE-ECHO-STREAM, создание двунаправленного потока с эхом  
MAKE-HASH-TABLE, создание хэш-массива  
MAKE-LIST, построение списка из одинаковых элементов  
MAKE-PATHNAME, создание пути  
MAKE-RANDOM-STATE, состояние генератора случайных чисел  
MAKE-SEQUENCE, создание последовательности 343  
MAKE-STRING-INPUT-STREAM, создание потока из (под)строки

**MAKE-SYMBOL**, создать имя 329  
**MAKE-SYNONYM-STREAM**, создание синонима потока  
**MAKE-TWO-WAY-STREAM**, создание двунаправленного потока  
**MAKUNBOUND**, удалить значение символа  
**MAP**, повторить для элементов последовательности 255, 344  
**MAPC**, повторить для головных частей 254, 257  
**MAPCAN**, повторить для головных частей и соединить 252, 254, 257  
**MAPCAR**, повторить для головных частей и собрать 250, 257  
**MAPCON**, повторить для хвостов и соединить 252, 254, 257  
**MAPHASH**, применить функционал к содержимому массива  
**MAPL**, повторить для хвостовых частей 254, 255, 257  
**MAPLIST**, повторить для хвостовых частей и собрать 252, 257  
**MAX**, максимальное число 321  
**MEMBER**, проверить, принадлежит ли элемент списку  
**MEMBER-IF**, проверить принадлежность  
**MEMBER-IF-NOT**, проверить принадлежность  
**MIN**, минимальное число 321  
**MINUSP**, проверка на отрицательность 320  
**MISMATCH**, позиционировать на первое отличие  
**MOD**, модуль числа  
**MULTIPLE-VALUE-LIST**, многозначное значение  
**MULTIPLE-VALUE-SETQ**, многозначное присваивание

**NAMESTRING**, преобразовать имя пути в строку  
**NAME-CHAR**, соответствующий имени знак  
**NBUTLAST**, удаление последних элементов  
**NCONC**, соединить списки физически  
**NIL**, логическое значение, ложь 64, 65, 128, 332  
    подтип всех типов 317  
**NINTH**, девятый элемент списка  
**NOT**, логическое отрицание 91  
**NOTANY**, логическое отрицание предиката **SOME**  
**NOTEVERY**, логическое отрицание предиката **EVERY**  
**NREVERSE**, перевернуть список физически  
**NSTRING-CAPITALIZE**, первые буквы в прописные  
**NSTRING-DOWNCASE**, прописные буквы в строчные  
**NSTRING-UPCASE**, строчные буквы в прописные  
**NSUBSTITUTE**, замена элемента, физическая  
**NSUBSTITUTE-IF**, замена элемента, физическая  
**NSUBSTITUTE-IF-NOT**, замена элемента, физическая  
**NTH**, n-й элемент списка 92  
**NTHCDR**, n-й CDR списка  
**NULL**, пустой список, предикат 91  
    типа данных NIL-а 317, 332  
**NUMBER**, число, тип 316, 317 320  
**NUMBERP**, число, предикат

**ODDP**, проверка на нечетность 323  
**OPEN**, открыть поток для файла 188

**OR**, логическое ИЛИ 135  
**OUTPUT-STREAM-P**, поток для вывода, предикат

**PACKAGE**, пространство имен, тип 180, 317  
**PAIRLIS**, построение а-списка 332, 333  
**PATHNAME**, имя пути, соответствующее изображению  
    имя пути, тип  
**PATHNAMEP**, имя пути, предикат  
**PEEK-CHAR**, подглядывание следующего знака  
**PHASE**, аргумент (комплексного) числа в радианах  
**PI**, значение числа  $\pi$   
**PLUSP**, проверка на положительность 320  
**POP**, взять элемент из стека  
**POSITION**, позиционирование на вхождение элемента  
**POSITION-IF**, позиционирование по условию  
**POSITION-IF-NOT**, позиционирование по условию  
**PPRINT**, структурная печать  
**PRIN1**, вывод объекта на ту же строку 181, 189  
**PRINC**, символьная печать 181, 189  
**PRINT**, вывод на отдельную строку 180, 184, 189  
**PROBE-FILE**, проверить, существует ли файл  
**PROCLAIM**, глобальное описание  
**PROG**, PROG-механизм с метками перехода 140  
**PROG\***, как PROG, но PROG-переменные последовательно  
**PROG1**, составная форма, значение – форма1 131  
**PROG2**, составная форма, значение – форма2 131  
**PROGN**, составная форма, значение – формаN 131  
**PROVIDE**, загрузка модуля  
**PSETF**, параллельное присваивание  
**PSETQ**, параллельное присваивание  
**PUSH**, поместить элемент в стек 292  
**PUSHNEW**, поместить элемент в стек

**QUOTE**, блокировка вычислений 76, 128, 259

**RANDOM**, случайное число 322  
**RASSOC**, поиск в а-списке, 333, 334  
**RASSOC-IF**, поиск в а-списке по условию  
**RASSOC-IF-NOT**, поиск в а-списке по условию  
**RATIO**, дробное число 316, 323  
**RATIONAL**, рациональное число, тип 316, 323  
**READ**, чтение выражения 175, 176, 189  
**READ-CHAR**, чтение знака  
**READ-CHAR-NO-HANG**, условное чтение знака  
**READ-FROM-STRING**, чтение из строки  
**READ-LINE**, чтение строки  
**READ-TABLE**, таблица чтения 117, 317  
**REDUCE**, применить функцию каскадно  
**REM**, остаток от деления

**REM**, удаление свойства из списка свойств  
**REMHASH**, удаление ключа и значения из хэш-массива  
**REMOVE**, удаление элемента 216, 345, 346  
**REMOVE-DUPLICATES**, удаление повторяющихся элементов  
**REMOVE-IF**, удаление на основе условия  
**REMOVE-IF-NOT**, удаление на основе условия  
**REMPROP**, удаление свойства символа 171  
**RENAME-FILE**, переименовать файл  
**REPLACE**, заменить часть последовательности  
**REQUIRE**, загрузка модуля  
**REST**, хвостовая часть списка  
**RETURN**, структурный выход со значением 140, 142  
**RETURN-FROM**, структурный выход из блока  
**REVERSE**, перевернутый список 218, 227, 234  
**ROOM**, сведения об использовании памяти  
**ROTAREF**, циклический сдвиг  
**ROUND**, округление числа  
**RPLACA**, присвоить в поле CAR ячейки 154, 162, 163  
**RPLACD**, присвоить в поле CDR ячейки 154, 162, 163

**SAME-PNAMEP**, совпадение печатного имени, предикат  
**SBIT**, элемент простого битового массива  
**SCHAR**, n-й знак простой строки  
**SEARCH**, первое вхождение подпоследовательности  
**SECOND**, второй элемент списка  
**SEQUENCE**, последовательность, тип 317, 341  
**SET**, присваивание 96, 99  
**SETF**, присваивание значения в ячейку памяти  
**SETQ**, присваивание 97, 99, 128  
**SET-CHAR-BIT**, присваивание битового имени знаку  
**SET-DIFFERENCE**, разность множеств  
**SEVENTH**, седьмой элемент списка  
**SHIFT**, сдвиг  
**SHORT-FLOAT**, короткое число с плавающей запятой 324  
**SIGNUM**, знак числа  
**SIMPLE-ARRAY**, простой массив, тип  
**SIMPLE-BIT-VECTOR**, простой битовый вектор, тип  
**SIMPLE-STRING**, простая строка, тип  
**SIMPLE-VECTOR**, простой вектор, тип  
**SiN**, синус угла 322  
**SINGLE-FLOAT**, число с плавающей запятой одинарной точности 324  
**SINH**, гиперболический синус 322  
**SIXTH**, шестой элемент списка  
**SLEEP**, приостановка вычислений  
**SOME**, проверка условия для некоторого элемента 345  
**SORT**, сортировка последовательности 348  
**SPECIAL**, динамическая специальная переменная  
**SQRT**, квадрантный корень из числа 322

**STABLE-SORT**, сортировка последовательности  
**STANDARD-CHAR**, стандартный знак, тип  
**STEP**, вычисление в пошаговом режиме  
**STREAM**, поток, тип 187, 317  
**STREAMP**, поток, предикат  
**STREAM-ELEMENT-TYPE**, тип элементов потока  
**STRING**, строка, тип 315, 337  
    преобразовать объект в строку 328, 339  
**STRING/=**, сравнение строк  
**STRING<**, сравнение строк  
**STRING<=**, сравнение строк  
**STRING=**, сравнение строк  
**STRING>**, сравнение строк  
**STRING>=**, сравнение строк  
**STRINGP**, сравнение строк  
**STRING-CAPITALIZE**, первые буквы на прописные  
**STRING-CHAR**, знак строки, тип  
**STRING-DOWNCASE**, прописные буквы на строчные  
**STRING-EQUAL**, равный в алфавитном порядке  
**STRING-GREATERP**, убывающий в алфавитном порядке  
**STRING-LEFT-TRIM**, удалить знаки из начала строки  
**STRING-LESSP**, возрастающий в алфавитном порядке  
**STRING-NOT-EQUAL**, неравный в алфавитном порядке  
**STRING-NOT-GREATERP**, неубывающий в алфавитном порядке  
**STRING-NOT-LESSP**, невозрастающий в алфавитном порядке  
**STRING-RIGHT-TRIM**, удалить знаки из конца строки  
**STRING-TRIM**, удалить знаки из начала и конца строки  
**STRING-UPCASE**, заменить строчные буквы на прописные  
**STRUCTURE**, структура, тип 315, 353  
**SUBLIS**, замена элементов по а-списку  
**SUBSEQ**, подпоследовательность 344  
**SUBSETP**, подмножество, предикат  
**SUBST**, замена элемента в-списке  
**SUBSTITUTE**, замена элемента  
**SUBSTITUTE-IF**, замена элемента по условию  
**SUBSTITUTE-IF-NOT**, замена элемента по условию  
**SUBST-IF**, замена по условию  
**SUBST-IF-NOT**, замена по условию  
**SVREF**, элемент простого вектора  
**SXHASH**, хэш-функция объекта  
**SYMBOL**, символ, тип 317, 326  
**SYMBOLP**, символ, предикат  
**SYMBOL-FUNCTION**, функциональное значение символа 110, 327  
**SYMBOL-NAME**, имя символа 327  
**SYMBOL-PACKAGE**, пространство имен символа  
**SYMBOL-PLIST**, список свойств символа 171, 327  
**SYMBOL-VALUE**, значение символа 97, 98, 327

**T**, логическое значение, истина 64, 128

- надтип всех типов 317  
**TAILP**, физический подсписок, предикат  
**TAN**, тангенс угла 322  
**TANH**, гиперболический тангенс 322  
**TENTH**, десятый элемент списка  
**TERPRI**, перевод строки 183, 189  
**THE**, описание типа формы  
**THIRD**, третий элемент списка  
**THROW**, вернуться к динамической метке 145, 146  
**TIME**, время вычисления  
**TRACE**, включить трассировку 206  
**TRUENAME**, преобразовать имя пути в имя файла  
**TRUNCATE**, обрубание до целого  
**TYPE**, описание типа символа  
**TYPECASE**, разветвление по типу ключа  
**TYPEP**, проверка типа, предикат 314  
**TYPE-OF**, тип объекта 314
- UNINTERN**, удаление символа из системы  
**UNION**, объединение множеств  
**UNLESS**, условное вычисление 136  
**UNREAD-CHAR**, возвращение знака во входной поток  
**UNTRACE**, выключить трассировку 206  
**UNWIND-PROTECT**, безусловное вычисление
- VALUES**, возврат многозначного значения  
**VALUES-LIST**, многозначное значение в виде списка  
**VECTOR**, создание одномерного массива  
    вектор, тип 315, 341  
**VECTORP**, одномерный массив, предикат  
**VECTOR-POP**, взятие текущего элемента из массива  
**VECTOR-PUSH**, занесение в массив следующего элемента  
**WHEN**, условное вычисление  
**WITH-OPEN-FILE**, создание и закрытие потока 189  
**WITH-OPEN-STREAM**, открытие и закрытие потока  
**WRITE**, общая функция вывода
- YES-OR-NO-P**, да/нет запрос пользователю
- ZEROP**, проверка на нуль 320
- +**, сложение чисел 321  
    переменная журнала, последнее введенное выражение  
**-**, вычитание чисел из числа 321  
    противоположное число 321  
    переменная журнала, текущая форма
- \***, умножение чисел 321  
    признак системной переменной  
    переменная журнала, последний результат

**/**, деление числа на числа 321

дробная черта

переменная журнала, последнее выведенное выражение

**=**, равны 88, 320

**/=**, не равны 320

**<**, меньше 320

**>**, больше 320

**>=**, больше или равно 320

**<=**, меньше или равно 320

**1+**, добавление единицы к числу 321

**1-**, вычитание единицы из числа 321

## Ключевые слова

**&ALLOW-OTHER-KEYS**, дополнительные аргументы

**&AUX**, вспомогательная переменная 112, 296

**&BODY**, макрос

**&ENVIRONMENT**, контекст, макрос

**&KEY**, ключевые параметры 112, 113, 115, 128, 296

**&OPTIONAL**, необязательные параметры 112, 296

**&REST**, переменное количество параметров 112, 113, 114, 117, 296

**&WHOLE**, вся форма вызова, макрос

## Ключи

**:ABORT**, закрытие потока

**:ADJUSTABLE**, динамический массив

**:APPEND**, OPEN

**:ARRAY**, WRITE

**:BASE**, WRITE

**:CAPITALIZE**, значение, \*PRINTCASE\*

**:CASE**, WRITE

**:CIRCLE**, WRITE

**:CONC-NAME**, DEFSTRUCT

**:CONSTRUCTOR**, DEFSTRUCT

**:COPIER**, DEFSTRUCT

**:COUNT**, количество, последовательность,

**:CREATE**, OPEN

**:DEFAULT**, OPEN

**:DEFAULTS**, MAKE-PATHNAME

**:DEVICE**, устройство

**:DIRECTION**, направление

**:DIRECTORY**, директорий

**:DISPLACED-INDEX-OFFSET**, массив

**:DISPLACED-TO**, массив

**:DOWN-CASE**, значение, \*PRINTCASE\*

**:ELEMENT-TYPE**, тип элемента

**:END**, конец, последовательность

**:ERROR**, OPEN

**:ESCAPE**, WRITE

**:EXTERNAL**, INTERN

**:FILL-POINTER**, указатель заполненности

**:FROM-END**, направление, последовательность

**:GENSYM**, WRITE

**:HOST**, система файлов

**:IF-DOES-NOT-EXIST**, форма

**:IF-EXISTS**, форма

**:INCLUDE**, DEFSTRUCT

**:INDEX**, поток в строке

**:INHERITED**, INTERN

**:INITIAL-CONTENTS**, массив

:INITIAL-ELEMENT, последова-	:PRINT-FUNCTION, DEFSTRUCT
тельность	
:INITIAL-OFFSET, DEFSTRUCT	:PROBE, OPEN
:INITIAL-VALUE, REDUCE	:RADIX, WRITE, PARSE-INT-
:INPUT, поток ввода	TEGER
:INTERNAL, INTERN	:READ-ONLY, полю нельзя при-
:IO, поток	сваивать 355
:JUNK-ALLOWED	:REHASH-SIZE, хэш-массив
:KEY, ограничение на элемент,	:REHASH-THRESHOLD, хэш-мас-
последовательность	сив
:LENGTH, WRITE	:RENAME, OPEN
:LEVEL, WRITE	:RENAME-AND-DELETE, OPEN
:NAME, имя	:SIZE, хэш-массив
:NAMED, DEFSTRUCT	:START, начало, последователь-
:NEW-VERSION, OPEN	ность
:NICKNAMES, пространство	:STREAM, WRITE
имен	:SUPERSEDE, OPEN
:OUTPUT, поток вывода	:TEST, условие выбора
:OUTPUT-FILE	:TEST-NOT, условие выбора
:OVER-WRITE, OPEN	:TYPE, тип поля 355
:PREDICATE, DEFSTRUCT	:UPCASE, значение,
:PRESERVE-WHITE-SPACE,	*PRINTCASE*
поток в строке	:USE, пространство имен
:PRETTY, WRITE	:VERBOSE, LOAD
:PRINT, LOAD	:VERSION, версия файла

### Системные переменные

- \*APPLYHOOK\*, переменная, интерпретатор
- \*BREAK-ON-WARNINGS\*, переменная, прерывания
- \*DEBUG-IO\*, переменная, отладка
- \*DEFAULT-PATHNAME-DEFAULTS\*, переменная, файл
- \*ERROR-OUTPUT\*, переменная, сообщения об ошибках
- \*EVALHOOK\*, переменная, интерпретатор
- \*FEATURES\*, переменная, системные возможности
- \*LOAD-VERBOSE\*, переменная, LOAD
- \*MACROEXPAND-HOOK\*, переменная, макросы
- \*MODULES\*, переменная, загруженные модули
- \*PACKAGE\*, переменная, пространство имен
- \*PRINT-ARRAY\*, признак, вывод
- \*PRINT-BASE\*, переменная, вывод
- \*PRINT-CASE\*, переменная, вывод
- \*PRINT-CIRCLE\*, признак, вывод
- \*PRINT-ESCAPE\*, признак, вывод
- \*PRINT-GENSYM\*, признак, вывод
- \*PRINT-LENGTH\*, переменная, вывод
- \*PRINT-LEVEL\*, переменная, вывод
- \*PRINT-PRETTY\*, признак, вывод
- \*PRINT-RADIX\*, переменная, вывод

- \*QUERY-IO\*, переменная, поток ввода/вывода
- \*RANDOM-STATE\*, переменная, состояние генератора по умолчанию
- \*READ-DEFAULT-FLOAT-FORMAT\*, точность по умолчанию
- \*READ-SUPPRESS\*, признак, ввод
- \*READTABLE\*, переменная, таблица по умолчанию
- \*STANDARD-INPUT\*, переменная, поток ввода по умолчанию 64, 187
- \*STANDARD-OUTPUT\*, переменная, поток вывода по умолчанию 187
- \*TERMINAL-IO\*, переменная, поток для диалога
- \*TRACE-OUTPUT\*, переменная, поток для трассировки

### Переменные журнала и комментарии

+, последнее введенное выражение

++, предпоследнее введенное

+++, предпредпоследнее введенное

\*, последний результат

\*\*, предпоследний

\*\*\*, предпредпоследний

/, последнее выведенное выражение

//, предпоследнее выведенное

///, предпредпоследнее выведенное

;, комментарий в конце строки 178

;;, вложенный комментарий 178

;;;, внешний комментарий 178

### Специальные знаки и способы обозначения

#, количество аргументов

форма представления 315

#', функциональная блокировка, FUNCTION 260

\#, знак 315

#(...), вектор 315

#C(...), комплексное число 315

#S(...), структура 315

#\*(...), битовый вектор 315

", ограничитель строки

', блокировка вычислений, QUOTE 76, 260

(, начало списка, ограничитель 200

изменение вида букв, FORMAT

), конец списка, ограничитель 200

,, отмена блокировки 299

', обратная блокировка 298

., точечная пара, разделитель 84, 152  
 ;, начало комментария 178  
 :, пространство имен 180  
     признак ключа 112  
     модификатор управляющего кода  
 \, выделение знака 62, 179, 328  
 !, выделение строки в имени 62, 179, 327  
 ~, признак управляющего кода, FORMAT  
 -, вычисляемая форма  
     приглашение 74  
 &, признак ключевого слова 111  
 @, присоединяющая отмена 299  
     модификатор управляющего кода

### Коды управления печатью

A, управляющий код, ASCII 185  
 B, управляющий код, двоичный  
 C, управляющий код, символьный  
 D, управляющий код, десятичный  
 E, управляющий код, с экспонентой  
 F, управляющий код, с плавающей запятой  
 \$, управляющий код, в денежных единицах  
 G, управляющий код, число с плавающей запятой  
 O, управляющий код, восьмеричный  
 R, управляющий код, число словами  
 S, управляющий код, s-выражение 185  
 T, управляющий код, табуляция  
 X, управляющий код, шестнадцатеричный

:, управляющий код, разделитель страниц  
 %, управляющий код, новая строка  
 &, перевод строки  
 \*, управляющий код, пропуск аргумента  
 ?, управляющий код, рекурсия  
   , управляющий код, конец цикла

<, управляющий код, выравнивание  
 >, управляющий код, конец выравнивания  
 [, управляющий код, условная форма  
 ], управляющий код, конец условной  
 {, управляющий код, цикл  
 }, управляющий код, конец цикла

# ПРИЛОЖЕНИЕ 3

## УКАЗАТЕЛЬ ИМЕН И СОКРАЩЕНИЙ

В этом приложении собраны встречающиеся в тексте имена и сокращения. Одновременно оно может служить указателем авторов по перечням литературы. С его помощью можно найти литературу, которая вследствие разбиения по темам приведена в разных разделах. Символы и зарезервированные слова Коммон Лиспа образуют отдельный указатель (приложение 2).

- Авсоний 22  
Ада 194  
Аккерман 202, 232  
Алгол-68 310  
**АНАЛИТИК** 35  
Ауэрбах Б. 309
- Байрон 249  
Бейсик 40, 194  
Бирс А. 194  
Блейк У. 319  
Бэкон Р. 69
- Витгенштейн Л. 46, 288  
Вольтер 349
- Гёте И. 337
- Дюамель Ж. 270  
Дюма А. мл. 239
- Зеталисп 16, 40
- Интерлисп 17, 40, 55, 116
- Калевала 19  
Клини 116  
Кобол 40, 194  
Коммон Лисп 16, 54, 55, 319  
Конфуций 245  
Крабб Дж. 174
- Лавров С. 58  
Лец Е. 127, 148, 205
- Лисп 40, 48, 194, 284  
Лого 40
- Маккарти Дж. 82  
Маклисп 55, 116, 183, 214, 319  
Мир 35
- Ницше Ф. 104
- Паронен С. 119  
Паскаль 40, 54, 194, 312, 353  
Паскаль Б. 331  
ПЛ/1 353  
Пролог 40, 55, 194, 284  
пятое поколение 24
- СБИС 40  
Си 40, 194  
Силагадзе Г. 58  
Симула 37  
Смолтолк 40, 284
- Твен М. 326, 353  
Тьюринг 201
- Фибоначчи 202  
Фортран 40, 48, 54, 194, 313, 319, 320  
Франц Лисп 111, 262  
Фуллер Т. 224
- Харрис С. 11
- Черч 104

- Эмерсон Р. 341  
д'Юрфе О. 259  
Abrahams P. 235  
Allen J. 286  
Alpac 23 /  
Alwey 12  
Apl 194  
ART 41  
ATN 41
- Bagley S. 59  
Banerji R. 28  
Barr A. 28  
Berwick R. 42  
Bibel W. 43  
Boden M. 44  
Boyer R. 286  
Brady J. 204, 244, 286  
Brady M. 42, 44  
Bramer D. 43  
Bramer M. 43  
Brown B. 306  
Brown R. 29  
Bundy A. 28  
Burge W. 286
- CAD 35, 40  
CAI 35  
CAM 35  
Campbell J.A. 43  
CAR 82  
CASNET/GLAUCOMA 23  
CAT 35  
Cattel G. 58  
CDR 82  
Chang C. 43  
Charniak E. 28, 42  
Clancey W. 42  
Clark K. 43  
Clocksin W. 43  
Cohen B. 28  
Cohen H. 45  
Cohen P. 45  
Curry H. 269
- Danicic I. 58  
Darlington J. 286
- DARPA 12  
DCG 41  
DENDRAL 23  
Dreyfus H. 44
- Edwards D. 235  
Eisenstadt M. 29  
Emycin 41  
ESPRIT 12  
EUREKA 12  
Expert 41
- Feigenbaum E. 28  
Flavor 16  
Foerster H. 204, 286  
FP 194  
Franz Lisp 111, 262  
Freys R. 269  
Friedman D. 58  
Friedman P. 269  
FRL 41
- Gloess P. 58  
Goerz G. 287, 306  
Gorz G. 59
- Hall E. 44  
Hamann C.-M. 58  
Hanson A. 44  
Hart T. 235  
Hasemer T. 58  
Hayes-Roth F. 42  
Henderson P. 58, 235, 244, 269, 286  
Hofstadter D. 45, 204, 286  
Horn B. 306  
Huomo T. 44  
Hyvonen E. 42, 44
- IBM 605 82  
ICOT 12  
Interlisp 17, 40, 55, 116
- Johnston T. 44
- KEE 41  
KL-TWO 41  
Knowledge Engineering Ky 15  
Kowalski R. 43  
Krc 194

- Krutch J. 29  
 Kulikowski C. 43  
 Landin P. 269  
 Lee R. 43  
 Lenat D. 42  
 Levin M. 235  
 Logo 194  
 Loveland D. 43  
 MacLisp 55, 116, 183, 214, 319  
 MACSYMA 23, 35  
 Makelin M. 44  
 Marr D. 44  
 Maurer W.D. 58  
 MCC 12  
 McCarthy J. 58, 235  
 McCorduck P. 45  
 McDermott D. 28  
 Mellish C. 43  
 Michie D. 42  
 Milner R. 244  
 Minsky M. 204  
 MIT 14  
 MITI 12  
 Moon D. 306  
 Moore J. 286  
 Morris J. 269  
 MYCIN 23  
 Neumann J. 286  
 Nial 194  
 Nii P. 45  
 NIL 285  
 Nilsson N. 29  
 NITEC 15  
 Nokia Inforamatiojarjestelmat 20  
 Norman A. 58  
 O'Shea T. 29  
 OPS5 41  
 Perlis D. 287  
 Prendergast K. 44  
 Queinnec C. 58, 306  
 Rank Xerox 20  
 Raphael B. 29, 45  
 REDUCE 35  
 Reisbeck C. 42  
 Ribbens D. 59  
 Rich E. 29  
 Riivari J. 44  
 Riseman H. 44  
 Rogers H. 204  
 Rozsa P. 204  
 S-1 319  
 SAC 35  
 Schank R. 42  
 Schraeger J. 59  
 Scott D. 287  
 SECD-машина 268  
 Seppänen J. 59, 235  
 Shortliffe E. 42  
 Siklossy L. 59  
 SITRA 13  
 Steele G. 59, 306  
 STeP-84 13  
 Stoyan H. 59, 287, 306  
 Stratchey C. 306  
 Tarnlund S.-U. 43  
 TEKES 13  
 Tennant H. 42  
 Texas Instruments 20  
 Touretzky D. 59  
 Tracton K. 59  
 Turner D. 269  
 VAX-11/780 275  
 Waite M. 306  
 Walker D. 42  
 Waterman D. 42, 43  
 Wegner P. 204  
 Weinreb D. 306  
 Weismann C. 59  
 Weiss S. 43  
 Weizenbaum J. 45  
 Wilensky R. 59, 306  
 Wilks Y. 42  
 Winograd T. 42  
 Winston P. 29, 43, 44, 59, 306  
 Wise D. 269  
 Zetalisp 16, 40

# ПРИЛОЖЕНИЕ 4

## ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

Указатель составлен из встречающихся в тексте понятий и терминов, а также из специальных терминов языка Лисп. Имена и сокращения собраны в свой указатель (приложение 3) так же, как символы Коммон Лиспа (приложение 2).

- Абстракция вычислений** 270
  - данных (data abstraction) 310
  - отображения 270
- Автофункция (auto-function)** 243
- Аксиомы типа** 310
- Анализ дискурса (discourse)** 32
  - синтаксический 72
- Анализатор (parser)** 177
- Аргумент** 69, 106
  - функциональный (functional argument) 239
- Атом (atom)** 64
- Барьер сложности (complexity barrier)** 286
- Блокировка вычислений (quote)** 75
  - обратная (back quote) 298
  - функциональная (function quote) 259
- Вектор (vector)** 350
- Верификация программ (program proving verification)** 37
- Включение или интернирование (intern)** 180, 329
- Выборка (selection)** 312
- Вызов макроса (macro call)** 128, 290
  - функции (call) 71, 240
- Выражение по умолчанию (init-form)** 113
  - символьное (s-expression) 66
- Вычисление (evaluation)** 73
  - отложенное (lazy/suspended evaluation) 244, 265
  - параллельное 265
  - символьное и алгебраическое (symbolic and algebraic computing SAC) 35
  - частичное (partial evaluation) 244, 263
  - численное (numeric computing) 24
- Вычислимость (computability)** 201
  - алгоритмическая (effective computability) 202
- Генератор (generator)** 264
- случайных чисел (random number generator)** 322
- Голова списка (head)** 80
- Гражданин полноправный (first class citizen)** 111
- Графика компьютерная (computer graphics)** 39
- Действия универсальные (generic)** 311
- Дерево бинарное (binary tree)** 225
- Доказательство теорем (theorem proving)** 36
- Замыкание (lexical closure)** 261, 315
- Запись (structure/record)** 311, 353
  - (нотация) префиксная 71

- Знак (character)** 337  
**Знания (knowledge)** 33  
 – активные 28  
**Значение (value)** 69, 312, 326  
 – глобальное (global) 122  
 – логическое (boolean) 309  
**Зрение машинное (machine vision)** 39
- Игра, ведение (game playing)** 37  
 – программирование 37
- Иерархия понятий (conceptual hierarchy)** 316
- Имя печатное (print name рпame)** 150, 326, 327
- Индекс (index)** 350
- Интеллект искусственный (artificial intelligence machine intelligence)** 23, 27
- Интерпретатор** 100, 127, 174, 245, 289
- Интерпретация (interpretation)** 34  
 – изображений (scene analysis) 39  
 – режим (interpretation) 73
- Интерфейс пользователя (human interface)** 32
- Источник (source)** ввода 189
- Код управляющий (directive)** 185
- Компилятор кремниевый (silicon compiler)** 40
- Константа (constant)** 64, 95
- Конструктор (constructor)** 312
- Контекст вычислительный (evaluation environment)** 121, 291
- Лингвистика (linguistics)** 32  
 – математическая (computational linguistics) 32
- Лямбда-вызов** 106
- Лямбда-выражение** 105, 270
- Лямбда-исчисление (lambda calculus)** 104
- Лямбда-преобразование (lambda conversion)** 107
- Лямбда-список (lambda list)** 105
- Макрознак (macro character)** 177  
**Макрос (macro)** 289  
 – вызов 128, 290  
 – расширение (expansion) или раскрытие 290  
 – структуроразрушающий (destructive) 301  
 – трансляция (translation) 290  
 – чтения (read macro) 177
- Массив (array)** 86, 349  
 – объектов (obarray) 179  
 – специализированный (specialized array) 350  
 – универсальный (general arrays) 350
- Машина вывода (inference engine)** 33
- Метка перехода (tag)** 140
- Механизм возвратов (backtracking)** 201
- Множество всех подмножеств множества (power set)** 274
- значений (range codomain) 69  
 – определения (domain) 69
- Моделирование (modelling)** 37  
 – когнитивное (cognitive modelling) 38  
 – работы (simulation) 37
- Морфология (morphology)** 32
- Мусор (garbage)** 159
- Мусорщик (garbage collector)** 51, 160
- Мутатор (mutator)** 312
- Наблюдение (monitoring)** 34
- Наука когнитивная (cognitive science)** 38
- Нотация списочная (list notation)** 157  
 – точечная (dot notation) 156, 157
- Область действия (scope)** 121
- Обновление (update mutation)** 312
- Обработка естественного языка (natural language processing)** 31

- знаний (knowledge processing) 22
- изображений (image processing) 39
- рисунков (picture processing) 39
- сигналов (signal processing) 38
- символьная (symbol manipulation) 24
- символьной информации (symbolic processing/computing) 24
- Образ графический (icon) 187
- Образец (template skeleton) 300
- Обучение (instruction tutoring) 35
  - языку (language learning) 32
- Объект (object) 311
  - данных (data object) 309
  - функциональный (functional object) 241
- Окружение вычислительное (evaluation environment) 121, 291
- Определение неявное (implicit) 309
  - функции (definition) 71
  - явное (explicit) 309
- Отмена блокировки замещающая 299
  - присоединяющая 299
- Отображение (mapping) 69
- Пакет (package) 180
- Пара точечная (dotted pair) 84, 156
- Параметр ключевой (key) 112
  - необязательный (optional) 112
  - передача по значению (call by value) 119
    - - - ссылке (call by reference) 119
  - фактический (actual parameter) 106
  - формальный (formal parameter) 105
- Перевод машинный (machine translation) 32
- Переменная (variable) 64
  - внешняя (external) 180
  - внутренняя (internal) 180
  - вспомогательная (auxiliary) 112
  - глобальная специальная (global special/dynamic variable) 64
  - динамическая или специальная (dynamic/special variable) 121 293
  - лексическая или статическая (lexical/static variable) 120
  - программная (program variable) 141
  - свободная (free variable) 121
- Печать структурная (prettyprinter) 67
- Планирование действий (planning) 34
- По умолчанию (default) 112
- Подсписок 65
- Поиск (search) 201
  - ошибки (diagnosis) 34
- Поле 149, 354
- Получатель (sink) вывода 189
- Последовательность (sequence) 257, 337, 340, 341
- Построение (construction) 312
  - прототипов быстрое (rapid prototyping) 53
- Поток (stream) 176, 187, 265
- Прагматика (pragmatics) 32
- Предикат (predicate) 79, 85
- Предложение (clause) 128
- Приглашение (prompt) 74
- Применение функции (apply) 71
- Присваивание (set) 96
- Прогнозирование (prediction) 34
- Программирование автоматическое (automatic programming program synthesis) 41
  - исследовательское (exploratory programming) 53
  - логическое (logic programming) 36, 52
  - объектно-ориентированное (object oriented programming) 38, 52, 244

- пошаговое (incremental programming) 52
- продукционное (rule-based programming) 52
- процедурное (procedural programming) 195
- ситуационное (event-based programming) 52
- управляемое данными (data driven programming) 50, 244
- функциональное (functional programming) 51, 196
- Проектирование (design)** 35
  - машинное (computer aided design computer aided engineering) 39
- Производство гибкое (flexible manufacturing)** 40
- Пространство имен (name space)** 180
- Процедура доступа (access operator accessor)** 312
  - чтения (lisp reader) 176
- Пустой список 65**
- Размер (size)** 349
- Размерность (dimensionality)** 349
- Распознавание образов (pattern recognition)** 38
  - речи (speech recognition) 32
- Рассуждения на уровне здравого смысла (common sence reasoning)** 37
- Режим EVALQUOTE 102**
- Рекурсия более высокого порядка 225, 232**
  - взаимная (mutual) 224, 228
  - параллельная 224, 225
  - по аргументам 205
    - значению 205
  - простая (simple) 205
- Решение алгоритмическое (algorithmic)** 25
  - эвристическое (heuristic) 25, 37
- Робототехника (robotics)** 39
- Самоизменение (self-modification)** 282
- Самосознание (self-consciousness)** 282
- Свойство (property)** 168
- Связывание (bind)** 96
  - параметров (spreading) 107
  - позднее (late binding) 54
- Связь (binding)** 96
  - время действия (extent) 122
- Селектор (selector)** 312
- Семантика (semantics)** 32, 101
- Символ 61, 326**
- Синтаксис (syntax)** 32
- Синтез программ (program synthesis)** 37
  - речи (speech synthesis) 32
- Система поддержки принятия решений (decision support system)** 33
  - самообучающаяся (learning system) 41
  - экспертная (expert system) 23, 33
- Слова ключевые (lambda-list keyword)** 111
- Сопоставление с образцом (pattern matching)** 39
- Список (list)** 65
  - ассоциативный (association list a-list) 311, 331, 332
  - объектов (object list oblist) 179
    - пар 311, 331, 332
    - свободной памяти (free list) 160
    - свойств (property list p-list) 150, 168, 311, 326
      - свободный (disembodied property list) 172
  - строка (string) 309, 337
    - управляющая (control string) 185
  - структура (structure/record) 311, 353
    - символьная 24, 61
- Таблица чтения (read table)** 177
- Тело функции (body)** 105

- Теория рекурсивных функций  
201
  - функций более высокого порядка 283
- Технология знаний (knowledge engineering) 23
- Тип (type) 70, 312
  - данных (data type) 309
  - абстрактный (abstract data type) 310
  - составной (compound) 309, 312
- Точка неподвижная (fixed point) 282
- Транслирование по частям (incremental compiling) 53
- Трассировка (trace) 206
- Указатель (pointer) 149
- Управление производством (control) 35
- Уровень командный (top level) 102
- Ускоритель вычислений с плавающей точкой (floating point accelerator) 319
- Фонология (phonology) 31
- Форма (form) 127
  - самоопределенная (self-evaluating) 127
  - специальная (special form) 128
- Формулы рекуррентные (recurrence formula) 202
- Фунарг-проблема (funarg problem) 265, 267
- Функционал (functional) 194, 239
  - применяющий или аппликативный (applicative functional) 245
- Функция 69, 240
  - автоаппликативная (self-applicative auto-applicative) 243, 280
  - авторепликативная (self-replicative auto-replicative) 243, 280
  - более высокого порядка (higher order) 243
- вызов (call) 71, 240
- доступа (access function) 305
- значение 69, 240
- многозначная (multiple valued function) 115
- общерекурсивная (general recursive) 203
- определение (definition) 71
- отображающая (mapping function) 249, 344
- порядок (order) 284
- применение (apply) 71
- примитивно рекурсивная (primitive recursive) 202
- рекурсивная (recursive) 142, 196
- с функциональным значением (function valued) 241
- структуроразрушающая (destructive) 161
- тип 283
- универсальная (universal function) 101, 245
- Хвост списка (tail) 80, 82
- Хэш-массив (hash array) 351
- Хэш-функция 352
- Цикл вложенный (nested) 229
- Черта вертикальная (bar) 62, 179, 328
- обратная косая (backslash) 62, 179, 327
- Число (number) 63, 319
  - вещественное (real) 309
  - дробное (ratio) 323
  - комплексное (complex) 315, 323, 325
  - рациональное (rational) 323
  - с плавающей запятой (float) 323
  - целое (integer) 309
- Шрифт (font) 186
- Экземпляр (instance) 264, 309
- Элемент (element) 65

- Эффект побочный (side effect) 99, 301
- Язык applicативный 194  
– бестиповый (typeless) 54, 284, 313  
– встроенный (embedded language) 289  
– декларативный (declarative) 194  
– естественный (natural language) 23  
– императивный (imperative) 194  
– логический (logic programming) 194  
– операторный (imperative) 194  
– процедурный (procedural) 194  
– функциональный (functional) 194
- Ячейка памяти (storage location) 98  
– списочная (memory cell list cell cons cell cons) 149
- MAP-функция 249, 344  
m-нотация (meta-notation) 115
- NLAMBDA-выражение 117
- PROG-механизм (prog feature) 139
- PROGN неявный (implicit progn feature) 132
- s-выражение (s-expression) 66

# ОГЛАВЛЕНИЕ

Предисловие редактора перевода . . . . .	5
<b>ВВЕДЕНИЕ . . . . .</b>	<b>11</b>
Скачок в развитии вычислительной техники . . . . .	11
Исследовательские программы искусственного интеллекта . . . . .	12
Национальные программы по исследованию языков . . . . .	13
Появление Лиспа в Финляндии . . . . .	13
Лисп – основа искусственного интеллекта . . . . .	14
Учебник Лиспа на финском языке . . . . .	14
Язык Лисп и функциональное программирование . . . . .	15
Методы и системы программирования . . . . .	16
На кого рассчитана книга . . . . .	17
Терминология . . . . .	17
Иконология . . . . .	18
От дерева к мысли и от мысли к дереву . . . . .	19
Благодарности . . . . .	20
<b>1 ВВЕДЕНИЕ В МИР ЛИСПА . . . . .</b>	<b>21</b>
<b>1.1 СИМВОЛЬНАЯ ОБРАБОТКА И ИСКУССТВЕННЫЙ ИНТЕЛЛЕКТ . . . . .</b>	<b>22</b>
Искусственный интеллект и технология знаний . . . . .	23
Исторические предубеждения . . . . .	23
Символьное или численное вычисление . . . . .	24
Эвристическое или алгоритмическое решение задачи . . . . .	25
Искусственный интеллект – сфера исследования многих наук . . . . .	27
Знание дела и умение как товар . . . . .	27
Учебники по искусенному интеллекту . . . . .	28
Упражнения . . . . .	29
<b>1.2 ПРИМЕНЕНИЯ ИСКУССТВЕННОГО ИНТЕЛЛЕКТА . . . . .</b>	<b>30</b>
Многообразие форм искусственного интеллекта . . . . .	30
Обработка естественного языка . . . . .	31
Экспертные системы . . . . .	33
Символьные и алгебраические вычисления . . . . .	35
Доказательства и логическое программирование . . . . .	36
Программирование игр . . . . .	37
Моделирование . . . . .	37
Обработка сигналов и распознавание образов . . . . .	38
Машинное зрение и обработка изображений . . . . .	39
Робототехника и автоматизация производства . . . . .	39
Машинное проектирование . . . . .	40
Языки и средства программирования искусственного интеллекта . . . . .	40
Повышение производительности программирования . . . . .	41
Автоматическое программирование и обучение . . . . .	41
Литература . . . . .	42

<b>Упражнения . . . . .</b>	<b>45</b>
<b>1.3 ЛИСП – ЯЗЫК ПРОГРАММИРОВАНИЯ ИСКУССТВЕННОГО ИНТЕЛЛЕКТА . . . . .</b>	<b>46</b>
Символьная обработка . . . . .	47
Лисп опередил свое время . . . . .	48
Одинаковая форма данных и программы . . . . .	49
Хранение данных, не зависящее от места . . . . .	50
Автоматическое и динамическое управление памятью . . . . .	51
Функциональный образ мышления . . . . .	51
Возможность различных методов программирования . . . . .	52
Пошаговое программирование . . . . .	52
Интерпретирующий или компилирующий режимы выполнения . . . . .	53
Лисп – бестиповый язык программирования . . . . .	53
Единый системный и прикладной язык программирования . . . . .	54
Интегрированная среда программирования . . . . .	55
Широко распространенные заблуждения и предрассудки . . . . .	56
Простой и эффективный язык . . . . .	57
Учебная литература по Лиспу . . . . .	58
Упражнения . . . . .	59
<b>2 ОСНОВЫ ЯЗЫКА ЛИСП . . . . .</b>	<b>60</b>
<b>2.1 СИМВОЛЫ И СПИСКИ . . . . .</b>	<b>61</b>
Символы используются для представления других объектов . . . . .	61
Символы в языке Коммон Лисп . . . . .	62
Числа являются константами . . . . .	63
Логические значения Т и NIL . . . . .	64
Константы и переменные . . . . .	64
Атомы = Символы + Числа . . . . .	64
Построение списков из атомов и подсписков . . . . .	65
Пустой список = NIL . . . . .	65
Список как средство представления знаний . . . . .	66
Значение способа записи . . . . .	67
Различная интерпретация списков . . . . .	67
Упражнения . . . . .	68
<b>2.2 ПОНЯТИЕ ФУНКЦИИ . . . . .</b>	<b>69</b>
Функция – отображение между множествами . . . . .	69
Тип аргументов и функций . . . . .	70
Определение и вызов функции . . . . .	71
Единообразная префиксная нотация . . . . .	71
Аналогия между Лиспом и естественным языком . . . . .	73
Диалог с интерпретатором Лиспа . . . . .	73
Иерархия вызовов . . . . .	74

<b>QUOTE блокирует вычисление выражения . . . . .</b>	<b>75</b>
<b>Упражнения . . . . .</b>	<b>77</b>
<b>2.3 БАЗОВЫЕ ФУНКЦИИ . . . . .</b>	<b>78</b>
Основные функции обработки списков . . . . .	79
Функция CAR возвращает в качестве значения головную часть списка . . . . .	80
Функция CDR возвращает в качестве значения хвостовую часть списка . . . . .	81
Функция CONS включает новый элемент в начало списка . . . . .	83
Связь между функциями CAR, CDR и CONS . . . . .	84
Предикат проверяет наличие некоторого свойства . . . . .	85
Предикат ATOM проверяет, является ли аргумент атомом . . . . .	85
EQ проверяет тождественность двух символов . . . . .	86
EQL сравнивает числа одинаковых типов . . . . .	88
Предикат = сравнивает числа различных типов . . . . .	88
EQUAL проверяет идентичность записей . . . . .	89
EQUALP проверяет наиболее общее логическое равенство . . . . .	90
Другие примитивы . . . . .	90
NULL проверяет на пустой список . . . . .	91
Вложенные вызовы CAR и CDR можно записывать в сокращенном виде . . . . .	91
LIST создает список из элементов . . . . .	93
Упражнения . . . . .	94
<b>2.4 ИМЯ И ЗНАЧЕНИЕ СИМВОЛА . . . . .</b>	<b>95</b>
Значением константы является сама константа . . . . .	95
Символ может обозначать произвольное выражение . . . . .	96
SET вычисляет имя и связывает его . . . . .	96
SETQ связывает имя, не вычисляя его . . . . .	97
SETF – обобщенная функция присваивания . . . . .	98
Побочный эффект псевдофункции . . . . .	99
Вызов интерпретатора EVAL вычисляет значение значения . . . . .	100
Основной цикл: READ-EVAL-PRINT . . . . .	102
Упражнения . . . . .	103
<b>2.5 ОПРЕДЕЛЕНИЕ ФУНКЦИЙ . . . . .</b>	<b>104</b>
Лямбда-выражение изображает параметризованные вычисления . . . . .	104
Лямбда-вызов соответствует вызову функции . . . . .	106
Вычисление лямбда-вызыва, или лямбда-преобразование . . . . .	106
Объединение лямбда-вызовов . . . . .	107
Лямбда-выражение – функция без имени . . . . .	108
DEFUN дает имя описанию функции . . . . .	108

<b>SYMBOL-FUNCTION</b> выдает определение функции . . . . .	110
Задание параметров в лямбда-списке . . . . .	111
Изображение функций в справочных руководствах . . . . .	114
Функции вычисляют все аргументы . . . . .	115
Многозначные функции . . . . .	115
Определение функции в различных диалектах Лиспа . . . . .	115
При вычислении <b>NLAMBDA</b> аргументы не вычисляются . . . . .	117
Упражнения . . . . .	117
<b>2.6 ПЕРЕДАЧА ПАРАМЕТРОВ И ОБЛАСТЬ ИХ ДЕЙСТВИЯ</b> . . . . .	119
В Лиспе используется передача параметров по значению . . . . .	119
Статические переменные локальны . . . . .	120
Свободные переменные меняют свое значение . . . . .	121
Динамическая и статическая область действия . . . . .	121
Одно имя может обозначать разные переменные . . . . .	123
Упражнения . . . . .	125
<b>2.7 ВЫЧИСЛЕНИЕ В ЛИСПЕ</b> . . . . .	127
Программа состоит из форм и функций . . . . .	127
Управляющие структуры Лиспа являются формами . . . . .	128
LET создает локальную связь . . . . .	129
Последовательные вычисления: PROG1, PROG2 и PROGN . . . . .	131
Разветвление вычислений: условное предложение COND . . . . .	132
Другие условные предложения: IF, WHEN, UNLESS и CASE . . . . .	136
Циклические вычисления: предложение DO . . . . .	137
Предложения PROG, GO и RETURN . . . . .	139
Другие циклические структуры . . . . .	142
Повторение через итерацию или рекурсию . . . . .	142
Формы динамического прекращения вычислений: CATCH и THROW . . . . .	145
Упражнения . . . . .	146
<b>2.8 ВНУТРЕННЕЕ ПРЕДСТАВЛЕНИЕ СПИСКОВ</b> . . . . .	148
Лисповская память состоит из списочных ячеек . . . . .	149
Значение представляется указателем . . . . .	149
CAR и CDR выбирают поле указателя . . . . .	151
CONS создает ячейку и возвращает на нее указатель . . . . .	151
У списков могут быть общие части . . . . .	152
Логическое и физическое равенство не одно и то же . . . . .	154
Точечная пара соответствует списочной ячейке . . . . .	155
Варианты точечной и списочной записей . . . . .	157
Управление памятью и сборка мусора . . . . .	159
Вычисления, изменяющие и не изменяющие структуру	160

<b>RPLACA и RPLACD изменяют содержимое полей . . . . .</b>	<b>161</b>
<b>Изменение структуры может ускорить вычисления . . . . .</b>	<b>163</b>
<b>Упражнения . . . . .</b>	<b>166</b>
<b>2.9 СВОЙСТВА СИМВОЛА . . . . .</b>	<b>168</b>
<b>У символа могут быть свойства . . . . .</b>	<b>168</b>
<b>У свойств есть имя и значение . . . . .</b>	<b>168</b>
<b>Системные и определяемые свойства . . . . .</b>	<b>169</b>
<b>Чтение свойства . . . . .</b>	<b>169</b>
<b>Присваивание свойства . . . . .</b>	<b>170</b>
<b>Удаление свойства . . . . .</b>	<b>171</b>
<b>Свойства глобальны . . . . .</b>	<b>171</b>
<b>Упражнения . . . . .</b>	<b>172</b>
<b>2.10 ВВОД И ВЫВОД . . . . .</b>	<b>174</b>
<b>Ввод и вывод входят в диалог . . . . .</b>	<b>174</b>
<b>READ читает и возвращает выражение . . . . .</b>	<b>175</b>
<b>Программа ввода выделяет формы . . . . .</b>	<b>176</b>
<b>Макросы чтения изменяют синтаксис Лиспа . . . . .</b>	<b>177</b>
<b>Символы хранятся в списке объектов . . . . .</b>	<b>179</b>
<b>Пакеты или пространства имен . . . . .</b>	<b>180</b>
<b>PRINT переводит строку, выводит значение и пробел . . . . .</b>	<b>180</b>
<b>PRIN1 и PRINC выводят без перевода строки . . . . .</b>	<b>182</b>
<b>TERPRI переводит строку . . . . .</b>	<b>183</b>
<b>FORMAT выводит в соответствии с образцом . . . . .</b>	<b>184</b>
<b>Использование файлов . . . . .</b>	<b>187</b>
<b>LOAD загружает определения . . . . .</b>	<b>190</b>
<b>Упражнения . . . . .</b>	<b>191</b>
<b>3 ФУНКЦИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ . . . . .</b>	<b>193</b>
<b>3.1 ОСНОВЫ РЕКУРСИИ . . . . .</b>	<b>194</b>
<b>Лисп – это язык функционального программирования . . . . .</b>	<b>194</b>
<b>Процедурное и функциональное программирование . . . . .</b>	<b>195</b>
<b>Рекурсивный – значит использующий самого себя . . . . .</b>	<b>196</b>
<b>Рекурсия всегда содержит терминальную ветвь . . . . .</b>	<b>197</b>
<b>Рекурсия может проявляться во многих формах . . . . .</b>	<b>198</b>
<b>Списки строятся рекурсивно . . . . .</b>	<b>200</b>
<b>Лисп основан на рекурсивном подходе . . . . .</b>	<b>201</b>
<b>Теория рекурсивных функций . . . . .</b>	<b>201</b>
<b>Литература . . . . .</b>	<b>204</b>
<b>3.2 ПРОСТАЯ РЕКУРСИЯ . . . . .</b>	<b>205</b>
<b>Простая рекурсия соответствует циклу . . . . .</b>	<b>205</b>
<b>MEMBER проверяет, принадлежит ли элемент списку . . . . .</b>	<b>208</b>
<b>Каждый шаг рекурсии упрощает задачу . . . . .</b>	<b>209</b>
<b>Порядок следования ветвей в условном предложении существенен . . . . .</b>	<b>211</b>

Ошибка в условиях может привести к бесконечным вычислениям . . . . .	213
APPEND объединяет два списка . . . . .	214
REMOVE удаляет элемент из списка . . . . .	216
SUBSTITUTE заменяет все вхождения элемента . . . . .	217
REVERSE обращает список . . . . .	218
Использование вспомогательных параметров . . . . .	220
Упражнения . . . . .	221
<b>3.3 ДРУГИЕ ФОРМЫ РЕКУРСИИ . . . . .</b>	<b>224</b>
Параллельное ветвление рекурсии . . . . .	225
Взаимная рекурсия . . . . .	228
Программирование вложенных циклов . . . . .	229
Рекурсия более высокого порядка . . . . .	232
Литература . . . . .	235
Упражнения . . . . .	235
<b>3.4 ФУНКЦИИ БОЛЕЕ ВЫСОКОГО ПОРЯДКА . . . . .</b>	<b>239</b>
Функционал имеет функциональный аргумент . . . . .	239
Функциональное значение функции . . . . .	241
Способы композиции функций . . . . .	242
Функции более высокого порядка . . . . .	243
Литература . . . . .	244
<b>3.5 ПРИМЕНЯЮЩИЕ ФУНКЦИОНАЛЫ . . . . .</b>	<b>245</b>
APPLY применяет функцию к списку аргументов . . . . .	246
FUNCALL вызывает функцию с аргументами . . . . .	246
Упражнения . . . . .	248
<b>3.6 ОТОБРАЖАЮЩИЕ ФУНКЦИОНАЛЫ . . . . .</b>	<b>249</b>
Отображающие функции повторяют применение функции . . . . .	249
MAPCAR повторяет вычисление функции на элементах списка . . . . .	250
MAPLIST повторяет вычисление на хвостовых частях списка . . . . .	252
MAPCAN и MAPCON объединяют результаты . . . . .	252
MAPC и MAPL теряют результаты . . . . .	254
Композиция функционалов . . . . .	255
Итоговая таблица отображающих функций . . . . .	256
Упражнения . . . . .	257
<b>3.7 ЗАМЫКАНИЯ . . . . .</b>	<b>259</b>
FUNCTION блокирует вычисление функции . . . . .	259
Замыкание – это функция и контекст ее определения . . . . .	260
Связи свободных переменных замыкаются . . . . .	261
Замыкания позволяют осуществлять частичное вычисление . . . . .	263
Генератор порождает последовательные значения . . . . .	264

Контекст вычисления функционального аргумента . . . . .	265
Литература . . . . .	269
Упражнения . . . . .	269
<b>3.8 АБСТРАКТНЫЙ ПОДХОД . . . . .</b>	<b>270</b>
Обобщение функций, имеющих одинаковый вид . . . . .	271
Параметризованное определение функций . . . . .	275
Рекурсивные функции с функциональным значением .	279
Автоаппликация и авторепликация . . . . .	280
Порядок и тип функций . . . . .	283
Проблемы абстрактного подхода . . . . .	285
Литература . . . . .	286
Упражнения . . . . .	287
<b>3.9 МАКРОСЫ . . . . .</b>	<b>288</b>
Макрос строит выражение и вычисляет его значение .	288
Макрос не вычисляет аргументы . . . . .	290
Макрос вычисляется дважды . . . . .	290
Контекст вычисления макроса . . . . .	291
Пример отличия макроса от функции . . . . .	292
Рекурсивные макросы и продолжающиеся вычисления	294
Тестирование макросов . . . . .	295
Лямбда-список и ключевые слова макроса . . . . .	296
Обратная блокировка разрешает промежуточные вычисления . . . . .	298
Образец удобно использовать для определения макросов . . . . .	300
Макросы с побочным эффектом . . . . .	301
Определение новых синтаксических форм . . . . .	304
Определение типов данных с помощью макросов .	305
Литература . . . . .	306
Упражнения . . . . .	307
<b>4 ТИПЫ ДАННЫХ . . . . .</b>	<b>308</b>
<b>4.1 ПОНЯТИЯ . . . . .</b>	<b>309</b>
Явное и неявное определение . . . . .	309
Абстракция данных . . . . .	310
Составные типы и процедуры доступа . . . . .	312
В Лиспе тип связан со значением, а не с именем .	312
Проверка и преобразование типов . . . . .	314
Иерархия типов . . . . .	316
Определение новых типов . . . . .	316
<b>4.2 ЧИСЛА . . . . .</b>	<b>319</b>
Лисп умеет работать с числами . . . . .	319
Целые числа . . . . .	323
Дробные числа . . . . .	323
Числа с плавающей запятой . . . . .	324

Комплексные числа . . . . .	325
<b>4.3 СИМВОЛЫ . . . . .</b>	<b>326</b>
Системные свойства символа . . . . .	326
Специальные знаки в символах . . . . .	327
Обращение с внешним видом символа . . . . .	328
GENTEMP создает новый символ . . . . .	330
<b>4.4 СПИСКИ . . . . .</b>	<b>331</b>
Ассоциативный список связывает данные с ключами . . . . .	332
PAIRLIS строит список пар . . . . .	332
ASSOC ищет пару, соответствующую ключу . . . . .	333
ACONS добавляет новую пару в начало списка . . . . .	334
PUTASSOC изменяет а-список . . . . .	335
<b>4.5 СТРОКИ . . . . .</b>	<b>337</b>
Знаки и строки . . . . .	337
Преобразования строк . . . . .	338
Работа со строками . . . . .	339
Наследуемые функции . . . . .	340
<b>4.6 ПОСЛЕДОВАТЕЛЬНОСТИ . . . . .</b>	<b>341</b>
Последовательности являются списками или векторами . . . . .	341
Основные действия с последовательностями . . . . .	342
Мощные функционалы . . . . .	344
Упорядочивание последовательности . . . . .	347
<b>4.7 МАССИВЫ . . . . .</b>	<b>349</b>
Типы массивов . . . . .	350
Работа с массивами . . . . .	350
Хэш-массив ассоциирует данные . . . . .	351
Хэш-массивы обеспечивают быстродействие . . . . .	351
<b>4.8 СТРУКТУРЫ . . . . .</b>	<b>353</b>
Структуры представляют собой логическое целое . . . . .	353
Определение структурного типа и создание структуры . . . . .	354
Функции создания, доступа и присваивания . . . . .	354
Действия не зависят от способа реализации . . . . .	356
<b>5 РЕШЕНИЯ . . . . .</b>	<b>358</b>
<b>ПРИЛОЖЕНИЕ 1. СВОДКА КОММОН ЛИСПА . . . . .</b>	<b>384</b>
<b>ПРИЛОЖЕНИЕ 2. УКАЗАТЕЛЬ СИМВОЛОВ КОММОН ЛИСПА . . . . .</b>	<b>417</b>
<b>ПРИЛОЖЕНИЕ 3. УКАЗАТЕЛЬ ИМЕН И СОКРАЩЕНИЙ . . . . .</b>	<b>431</b>
<b>ПРИЛОЖЕНИЕ 4. ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ . . . . .</b>	<b>434</b>