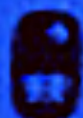


**БИБЛИОТЕЧКА
ПРОГРАММИСТА**

С. С. ЛАВРОВ, Г. С. СИПАГАДЗЕ

**Автоматическая
обработка данных
Язык лисп
и его реализация**



БИБЛИОТЕЧКА
ПРОГРАММИСТА

С. С. ЛАВРОВ, Г. С. СИЛАГАДЗЕ

АВТОМАТИЧЕСКАЯ
ОБРАБОТКА ДАННЫХ
ЯЗЫК ЛИСП
И ЕГО РЕАЛИЗАЦИЯ



МОСКВА «НАУКА»

ГЛАВНАЯ РЕДАКЦИЯ

ФИЗИКО-МАТЕМАТИЧЕСКОЙ ЛИТЕРАТУРЫ

1978

22.18

Л 13

УДК 519.6

Автоматическая обработка данных. Язык лисп и его реализация. Серия «Библиотечка программиста». Лавров С. С., Силагадзе Г. С. Главная редакция физико-математической литературы издательства «Наука», М., 1978.

Язык, одна из версий которого описана в книге, пользуется широкой известностью как язык для описания задач обработки символьной информации и искусственного интеллекта.

В книге довольно подробно изложены методы интерпретации и компиляции программ на этом языке. Некоторые из этих методов могут с успехом быть использованы при реализации других языков, в частности при составлении программ обработки символьной информации на языках низкого уровня. Приводится много примеров описаний функций.

Книга рассчитана на широкий круг программистов, сталкивающихся с задачами указанных классов, в частности, — на студентов старших курсов, специализирующихся в области математического обеспечения ЭВМ.

Святослав Сергеевич Лавров, Гиви Сергеевич Силагадзе

**АВТОМАТИЧЕСКАЯ ОБРАБОТКА ДАННЫХ
ЯЗЫК ЛИСП И ЕГО РЕАЛИЗАЦИЯ**

(серия: «Библиотечка программиста»)

М., 1978 г., 176 стр. с илл.

Редактор Г. Я. Пирогова

Техн. редактор Л. В. Лихачева

Корректор Н. Б. Румянцева

ИБ № 11301

Сдано в набор 01.06.78. Подписано к печати 14.11.78. Т-20143. Бумага 84×108¹/₃₂, тип. № 1. Литературная гарнитура. Высокая печать. Условн. печ. л. 9,24. Уч.-изд. л. 9,67. Тираж 25 000 экз. Заказ № 1988. Цена 60 коп.

Издательство «Наука»

Главная редакция физико-математической литературы
117071, Москва, В-71, Ленинский проспект, 15

Ордена Октябрьской Революции, ордена Трудового Красного Знамени Ленинградское производственно-техническое объединение «Печатный Двор» имени А. М. Горького Союзполиграфпрома при Государственном комитете Совета Министров СССР по делам издательств, полиграфии и книжной торговли. 197136, Ленинград, П-136, Гатчинская ул., 26

Л $\frac{20204-178}{053(02)-78}$ 64-79

© Главная редакция
физико-математической литературы
издательства «Наука», 1978

ОГЛАВЛЕНИЕ

Предисловие	5
Введение	7
ГЛАВА 1. Описание языка	8
1.1. Литеры и алфавит	8
1.2. Атомы	8
1.3. Списки	10
1.4. Выражения	11
1.5. Функции	12
1.6. Функция <i>QUOTE</i>	13
1.7. Функция <i>CAR</i>	13
1.8. Функция <i>CDR</i>	14
1.9. Композиции <i>CAR</i> и <i>CDR</i>	15
1.10. Пустой список	16
1.11. Функция <i>CONS</i>	16
1.12. Логические значения и предикаты	18
1.13. Функция <i>ATOM</i>	18
1.14. Функция <i>EQ</i>	19
1.15. Условные выражения	20
1.16. Определяющие выражения функций	21
1.17. Функция <i>NULL</i>	23
1.18. Встроенные и определяемые функции	23
1.19. Обычные и специальные функции	24
1.20. Функции <i>SEXPR</i> и <i>SFEXPR</i>	25
1.21. Рекурсивные функции	27
1.22. Функция <i>CSETQ</i> , константы	29
1.23. Программа	30
1.24. Аппарат <i>PROG</i>	31
1.25. Переменные	34
1.26. Приемы программирования	35
1.27. Функции <i>READ</i> , <i>PRINT</i> и <i>GENSYM</i>	37
1.28. Функция <i>EVAL</i>	39
1.29. Функция <i>LIST</i>	40
1.30. Предикаты <i>AND</i> и <i>OR</i>	40

1.31. Обобщение понятия выражения	41
1.32. Числа	44
1.33. Предикаты, классифицирующие атомы	45
1.34. Арифметические функции и предикаты	46
1.35. Операции над строками битов	50
1.36. Функционалы	51
1.37. Функция <i>SELECTQ</i>	54
1.38. Пример программы	55
ГЛАВА 2. Реализация языка лисп	62
2.1. Внутреннее представление выражений	62
2.2. Списки свойств атомов	65
2.3. Язык для описания реализации	69
2.4. Распределение памяти	74
2.5. Список объектов	75
2.6. Подпрограммы функций <i>PRINT</i> и <i>READ</i>	78
2.7. Ассоциативный список	83
2.8. Интерпретатор	85
2.9. Интерпретация <i>PROG</i>	90
2.10. Функциональные аргументы	93
2.11. Арифметические функции	97
2.12. Организация рекурсивных подпрограмм	98
2.13. Уборка мусора	102
2.14. Компилятор	110
2.15. Ограничения	121
2.16. Переходы	124
ГЛАВА 3. Библиотека вспомогательных функций	128
3.1. Операции над списками	128
3.2. Функции с побочным эффектом	132
3.3. Предикаты	138
3.4. Порядок и упорядочивание	139
3.5. Поиск	143
3.6. Операции над множествами	144
3.7. Ассоциативные списки	148
3.8. Функционалы	153
3.9. Операторы	154
3.10. Списки свойств	156
Приложение	162
Литература	166
Предметный указатель	168
Указатель лисповских функций и констант	171
Указатель обозначений, использованных в описании реализации	174

ПРЕДИСЛОВИЕ

В этой книге отражен опыт авторов по работе с языком лисп и его реализации на вычислительных машинах БЭСМ-6, ОДРА-1204 и др. Этот язык входит в десятку наиболее распространенных языков, хотя и не занимает «призовых» мест. В его основу положены оригинальные идеи, знакомство с которыми обогащает программистскую квалификацию и культуру и приносит поэтому пользу, даже если программист в дальнейшем и не работает с этим языком. Уже после лиспа появились другие языки, предназначенные для тех же целей и в чем-то его превосходящие: рефал, PLANNER, CONNIVER и др. Современные универсальные языки также обладают большими возможностями в части обработки символьной информации. Но все это не умаляет значения лиспа, который содержит в наиболее чистом виде понятия, теряющиеся среди многих других понятий в более современных и универсальных языках.

Большое внимание уделено в этой книге реализации языка. Авторы не предполагают, что создание интерпретаторов или компиляторов для этого языка станет массовым занятием программистов. Однако методы интерпретации и компиляции, описанные в книге, могут быть применены во многих других случаях, особенно при реализации языков, имеющих дело с символьной информацией. Кроме того, описание интерпретатора может, по нашему мнению, служить примером подхода к составлению любых больших программ, т. е. программ, значительно превосходящих по размеру и сложности те программы, которые встречаются в учебных пособиях или излагаются на занятиях по программированию. Отсутствие подобных примеров создает довольно большие трудности при изучении программирования для ЭВМ. Глава 2 книги в какой-то мере заполняет этот пробел.

В соответствии с традицией, насчитывающей более 20 лет, многие авторы (в том числе и авторы этой книги) пишут названия языков программирования прописным шрифтом. Однако Главная редакция

физико-математической литературы издательства «Наука» решила перейти на написание названий языков программирования строчным шрифтом *).

В начале работы над реализацией языка авторам принесли большую пользу материалы, полученные от Дж. Мак-Карти и Э. Беркли, а также сведения, содержащиеся в книгах [12, 15].

В главе 3 использованы материалы библиотеки подпрограмм на лиспе, собранной Э. Беркли. Рукопись книги прочитали и сделали ряд замечаний, способствовавших ее улучшению, С. А. Абрамов, В. Н. Пильщиков, В. М. Юфа.

Всем названным лицам авторы глубоко признательны.

С. Лавров

Г. Силагадзе

*) **П р и м е ч а н и е р е д а к ц и и.** В связи с неоднозначностью шрифтовых начертаний названий языков программирования в разных изданиях редакция приняла решение унифицировать в своих изданиях шрифт для названий языков программирования, руководствуясь энциклопедическими изданиями.

ВВЕДЕНИЕ

Язык *LISP* (от английского *LISt Processing* — обработка списков) был предложен Дж. Мак-Карти в статье [16]. Основное назначение языка — описывать рекурсивные функции символьных выражений. Как мы увидим, понятия символьного выражения и рекурсивной функции, имеющей такие выражения своими аргументами, определяются очень просто — проще, чем основные понятия большинства других алгоритмических языков. При этом язык обладает важными качествами: универсальностью, удобством в пользовании и простотой реализации. Универсальность — это теоретически важное свойство языка. Оно означает, что на данном языке могут быть описаны любые функции, реализуемые другими известными средствами, например, с помощью так называемых машин Тьюринга или нормальных алгоритмов Маркова. Удобство в пользовании и простота реализации — важные практические достоинства языка. Под удобством в пользовании мы понимаем, что, когда метод решения некоторой задачи ясен, он легко описывается средствами языка. Описание при этом оказывается компактным и легко обозримым. Более того, язык лисп устроен так, что он во многих случаях позволяет находить путь к решению задачи постепенно, последовательно сводя ее к более простым задачам.

Разумеется, как всякий рабочий инструмент, этот язык требует приобретения навыков и сноровки в пользовании им. Предварительное знакомство с другими языками, средствами или понятиями программирования не обязательно, хотя и может способствовать более быстрому овладению языком. Лучший способ изучить язык — это самостоятельное составление программ и сравнение их с программами, составленными более опытными программистами. Для этого можно использовать многочисленные примеры описаний функций, приведенные в этой книге, особенно в главе 3.

ГЛАВА 1

ОПИСАНИЕ ЯЗЫКА

1.1. Литеры и алфавит

Под *литерой* будем понимать любой печатный знак из некоторого конечного набора, называемого *алфавитом*. Состав алфавита более или менее произволен. Обычно он совпадает с набором знаков, которые могут быть отперфорированы на перфокартах или на перфоленте для последующего ввода в машину и отпечатаны печатающим устройством этой машины.

Пробел между печатными знаками также считается литерой. В рукописном тексте для большей четкости пробел изображают знаком — . В печатном тексте эта литера обнаруживается совершенно явно, так как каждый печатный знак имеет определенное положение в строке. Переход с одной строки текста на другую (признак *nl* новой строки при вводе в машину) рассматривается как пробел.

В этой книге будет использоваться следующий алфавит.

Буквы: A B C D E F G H I J K L M N O P Q R S T U V
W X Y Z Б Г Д Ж З И Й Л П Ф Ц Ч Ш Щ Ы Ь Э Ю Я

Цифры: 0 1 2 3 4 5 6 7 8 9

Ограничители лиспа: () . — *nl*

Прочие литеры: + — × / , : ; =

1.2. Атомы

Число литер в алфавите невелико. В задачах же, решаемых с помощью лиспа, часто приходится иметь дело с большим количеством различных объектов. Для их обозначения применяются *атомы*. Атомом называется произвольная последовательность букв, цифр и (или) прочих литер, заключенная между двумя ограничителями языка лисп (не включая самих ограничителей).

В различных реализациях языка лисп на запись атомов накладываются те или иные ограничения. В частности, атом, начинающийся с цифры или со знака $+$ или $-$, обычно изображает число и должен быть записан в соответствии с особыми правилами (см. разд. 1.32).

В последовательности литер

$(A (ATOM X24) - ABCDEFGHIJKLMNOPQRSTUVWXYZ$
 $012 \times (-100\ 2.5) (A + B (X) := (GO\ TO))) +)$

содержатся следующие атомы:

A
 $ATOM$
 $X24$
 $-$
 $ABCDEFGHIJKLMNOPQRSTUVWXYZ$
 012
 \times
 -100
 2
 5
 $A + B$
 X
 $:=$
 GO
 TO
 $+$

В программе на языке лисп атомы могут обозначать функции и константы, имеющие в лиспе фиксированный смысл, функции, переменные и константы, определенные в этой программе, и, наконец, самих себя, т. е. те последовательности литер или отдельные литеры, которыми они изображаются. В последнем случае атому может быть приписан некоторый смысл, остающийся за пределами системы понятий языка лисп. Этот смысл обычно излагается в виде словесных пояснений к программе. Но это так же не обязательно, как не обязательно вкладывать какой-либо смысл в обозначения a , b , c , x , когда речь идет о правилах решения квадратного уравнения $ax^2 + bx + c = 0$.

Атомы, имеющие фиксированный смысл в языке лисп, т. е. наименования стандартных или, как их еще называют, встроенных констант и функций, при некоторых условиях могут быть использованы программистом для обозначения других объектов, вводимых им самим. Во избежание путаницы лучше этого не делать. Однако,

если программист забыл о существовании какой-либо встроенной функции или константы (и, следовательно, не использует ее в своей программе) и применяет ее наименование в своих целях, то ничего страшного не случится. Однако потом, при расширении или модификации программы, воспользоваться этой функцией или константой уже не удастся.

1.3. Списки

Списком называется конечная последовательность элементов списка, заключенная в круглые скобки. Элементом списка может быть либо атом, либо список.

Примеры:

```
(A B C D E)
(U1 U2 U3)
(CAR (QUOTE (P Q R)))
(1 (2 3) (4 (5 6)))
(X)
(((1000)))
( )
```

Последний пример — *пустой* список, не содержащий ни одного элемента.

Элементы списка разделяются пробелами, однако пробел обязателен только в том случае, когда оба соседних элемента — атомы. Во всех других случаях пробелы не существенны. Несколько следующих подряд пробелов эквивалентны одному пробелу. Таким образом, записи

```
(1 (2 3) (4 (5 6)))
(1 (2 3) (4 (5 6)))
( 1 ( 2 3) ( 4 ( 5 6 ) ) )
```

изображают один и тот же список, но первый вариант записи достаточно нагляден и компактен и рекомендуется его придерживаться. Второй вариант — самый компактный.

Скобки в лиспе — важнейшие синтаксические знаки. Необходимо тщательнейшим образом следить за правильностью их расстановки. Отсутствие одной из парных скобок совершенно искажает программу и обычно делает ее бессмысленной. К таким же последствиям приводит отсутствие пары скобок в нужных позициях или наличие лишней пары. Поэтому необходимо скрупулезно соблюдать

правила записи программы и не жалеть труда на контрольный подсчет скобок. Для этого полезна нумерация скобок по следующему образцу:

(1 (2 3) (4 (5 6)))
 1 2 2 2 3 321

1.4. Выражения

Под *выражением* мы пока будем понимать либо атом, либо список. Впоследствии мы познакомимся еще с одним, более общим видом выражения (разд. 1.31). Список — наиболее распространенный вид выражений лиспа, от него произошло и название языка (LISt Processing — обработка списков).

Некоторые выражения могут иметь *значения*. Значениями выражений являются также выражения. Выражения, которые могут иметь значения, — это константы, переменные, наименования функций и обращения к функциям. С каждой функцией связывается набор ее переменных (в другой терминологии — параметров). Значения этих переменных определяются заново при каждом обращении к данной функции. Значения констант остаются неизменными на протяжении ряда обращений к одной и той же или к нескольким функциям. Если константа — не встроенная, то ее значение должно быть определено в программе (см. разд. 1.22) и может быть переопределено несколько раз.

Наименование функции — это по сути дела разновидность константы, отличающаяся тем, что ее значением является так называемое определяющее выражение некоторой функции (см. разд. 1.16), а также способом употребления этого значения. Значение функции вычисляется заново при каждом обращении к функции. Оно зависит в общем случае от аргументов, указанных в обращении, и от состояния программы. Это значение вычисляется по определенным (для данной функции) правилам, которые либо предписаны языком, либо задаются определяющим выражением этой функции, написанным программистом. Более подробно содержание всех этих понятий раскрывается ниже.

Если требуется вычислить значение выражения, имеющего вид списка, то первым элементом этого выражения должно быть наименование функции или определяющее выражение функции. Во всех остальных случаях список заведомо не имеет значения. Так, например, выражение

(P Q R)

может иметь значение лишь в том случае, когда P — наименование

функции двух переменных. Выражение

((1 2) 3)

не имеет значения, так как первый элемент этого списка — список (1 2) не является ни наименованием, ни определяющим выражением какой бы то ни было функции. Впрочем, мы забежали вперед, так как еще не знаем, что такое определяющее выражение функции.

1.5. Функции

Обращение к функции, которое в обычной символике принято писать в виде $F(X, Y, Z)$, записывается так:

(F X Y Z)

Оно имеет вид списка, первым элементом которого является атом — *наименование функции*, а следующими — аргументы данного обращения. Таким образом, перед наименованием функции обязательно стоит открывающая скобка, а между этим наименованием и записью первого аргумента никакой разделитель, кроме, может быть, пробела, не ставится. Точно так же аргументы могут быть разделены лишь пробелом. За последним аргументом следует закрывающая скобка, парная к скобке, стоящей перед наименованием функции.

Аргументы функции — это выражения. Каждая функция может накладывать ограничения на число и вид своих аргументов. Если функция с наименованием G не требует задания аргументов, то обращение к ней имеет вид

(G)

Запись

(CAR (QUOTE (P Q R)))

если мы знаем, что CAR и $QUOTE$ — наименования функций, означает обращение к функции CAR с одним аргументом ($QUOTE (P Q R)$), который изображает обращение к функции $QUOTE$ также с одним аргументом — списком ($P Q R$). Если известно, что при обращении к $CONS$ необходимы два аргумента, а к функции $READ$ можно обращаться без аргументов, то допустима запись

(CONS (READ) (READ))

означающая обращение к функции $CONS$ с двумя аргументами, значения которых определяются в результате двух последовательных обращений к функции $READ$.

Спрашивается — существуют ли списки, не являющиеся обращениями к функциям, и как их распознавать?

1.6. Функция *QUOTE*

Функция *QUOTE* как раз и указывает, что ее аргумент изображает сам себя, что его не следует трактовать как обращение к функции или еще каким-либо способом пытаться определить его значение. Сам этот аргумент в том виде, в каком он написан, служит значением функции *QUOTE* при данном обращении к ней. Так, значением выражения

(QUOTE A)

является атом *A*, а значением выражения

(QUOTE (P Q R))

служит список *(P Q R)*. Значение выражения

(QUOTE P Q R)

не определено, ибо в этой записи для функции *QUOTE* заданы три аргумента вместо допустимого и требующегося одного.

Было бы неверно утверждать, что значение функции *QUOTE* совпадает со значением ее аргумента. Аргумент может не иметь никакого значения, а если и имеет, то не оно является значением функции *QUOTE*, а сама внешняя форма аргумента. Так, значением

(QUOTE (CAR X))

будет список *(CAR X)*, а не значение той функции, к которой можно было бы обратиться в таком виде.

Во многих случаях при выполнении лисповской программы требуется вычислить и подвергнуть дальнейшей обработке значение выражения, стоящего в той или иной позиции программы (например, в позиции аргумента при обращении к функции). Если мы хотим задать это значение явно, то следует прибегнуть к функции *QUOTE*.

1.7. Функция *CAR*

Что можно делать со списками? Одна из первых потребностей — посмотреть, из каких элементов список составлен. Удовлетворению этой потребности частично содействует функция *CAR*. Эта функция выделяет первый элемент из значения своего единственного аргумента, если это значение — непустой список. Если же значение аргумента — атом, а не список, то значение функции *CAR* не определено.

Значение выражения

$(CAR (QUOTE (P Q R)))$

равно P , потому что значением аргумента является список $(P Q R)$ и первый элемент этого списка есть P . Тот факт, что значение выражения e^* равно v , мы будем записывать в виде $e \rightarrow v$, так что

$(CAR (QUOTE (P Q R))) \rightarrow P$.

Аналогично

$(CAR (QUOTE ((1 2) 3))) \rightarrow (1 2)$

потому что

$(QUOTE ((1 2) 3)) \rightarrow ((1 2) 3)$

а первым элементом списка $((1 2) 3)$ является список $(1 2)$.

А чему равно значение выражения $(CAR ((1 2) 3))$? Это значение не определено, так как предварительно требуется вычислить значение аргумента — выражения $((1 2) 3)$, а это выражение не имеет значения.

Не определено и значение выражения

$(CAR (QUOTE ALPHA))$

потому что значение аргумента — атом, а не список.

1.8. Функция *CDR*

Подобно *CAR* функция *CDR* — это функция с одним аргументом, значение которого должно быть непустым списком. Исключая из этого списка первый элемент, получаем список, который и дает значение функции *CDR*.

П р и м е р ы:

$(CDR (QUOTE (P Q R))) \rightarrow (Q R)$

$(CDR (QUOTE ((1 2) 3))) \rightarrow (3)$

(выбрасывая из списка $((1 2) 3)$ его первый элемент — список $(1 2)$, получаем список (3) , но не атом 3).

$(CDR (QUOTE (A))) \rightarrow ()$

— пустой список, получающийся после отбрасывания единственного элемента A списка (A) .

*) Здесь и в дальнейшем строчные буквы обозначают (если нет никаких оговорок) произвольные выражения языка.

$(CDR (QUOTE PARIS))$

не определено, потому что значение аргумента есть атом *PARIS*, а не список.

1.9. Композиции *CAR* и *CDR*

Функции *CAR* и *CDR* позволяют добраться до любого элемента любого списка. Пусть, например, значением переменной *X* является список

$$\begin{array}{ccccccc} ((LAMBDA & (U & V) & (CONS & U & (CONS & V & NIL))) \\ 12 & & 3 & 3 & 3 & & 4 & 432 \\ (ALPHA & A) & (BETA & B)) \\ 2 & & 2 & 2 & & & 21 \end{array}$$

Тогда

$$\begin{aligned} (CAR X) &\rightarrow (LAMBDA (U V) (CONS U (CONS V NIL))) \\ (CDR X) &\rightarrow ((ALPHA A) (BETA B)) \\ (CAR (CAR X)) &\rightarrow LAMBDA \\ (CDR (CAR X)) &\rightarrow ((U V) (CONS U (CONS V NIL))) \\ (CAR (CDR X)) &\rightarrow (ALPHA A) \\ (CDR (CDR X)) &\rightarrow ((BETA B)) \\ (CAR (CDR (CDR (CAR X)))) &\rightarrow (CONS U (CONS V NIL)) \\ (CDR (CDR (CDR X))) &\rightarrow () \\ (CDR (CAR (CDR (CDR X)))) &\rightarrow (B) \end{aligned}$$

и т. д.

Для упрощения записи таких многократных обращений к функциям *CAR* и *CDR* применяются наименования функций вида *CAAR* (двукратное обращение к *CAR*), *CDAR* (обращение сначала к *CAR* — внутреннее обращение, а затем к *CDR*), *CADDAR* и др. Таким образом, некоторые из приведенных выше примеров можно записать так:

$$\begin{aligned} (CAAR X) &\rightarrow LAMBDA \\ (CADR X) &\rightarrow (ALPHA A) \\ (CADDAR X) &\rightarrow (CONS U (CONS V NIL)) \\ (CDADDR X) &\rightarrow (B) \end{aligned}$$

и т. д. В реализациях языка сложность таких обозначений обычно бывает ограничена.

Из комбинированных обозначений в свою очередь можно строить композиции, например,

$$(CADDR (CADDR (CADDR X))) \rightarrow NIL$$

Заметим, что функции *CAR*, *CADR*, *CADDR*, *CADDDR* выделяют соответственно 1-й, 2-й, 3-й и 4-й элементы списка, являющегося значением аргумента функции:

$$(CAR (QUOTE (1\ 2\ 3\ 4\ 5))) \rightarrow 1$$
$$(CADR (QUOTE (1\ 2\ 3\ 4\ 5))) \rightarrow 2$$
$$(CADDR (QUOTE (1\ 2\ 3\ 4\ 5))) \rightarrow 3$$
$$(CADDDR (QUOTE (1\ 2\ 3\ 4\ 5))) \rightarrow 4$$

1.10. Пустой список

Пустой список, который мы обозначали до сих пор так:

()

имеет второе обозначение, а именно

NIL

Оба обозначения совершенно эквивалентны. Атом *NIL* — это пример встроенной константы. Ее значением является пустой список, т. е. сам *NIL* (так принято для этой константы). Это свойство атома *NIL* позволяет обходиться без функции *QUOTE*, когда нам нужно использовать в качестве значения сам этот атом. Другими словами, выражения

(*QUOTE NIL*)

и

NIL

эквивалентны и следует пользоваться вторым из них как более простым. Напротив, выражения *NIL* и (*NIL*) не эквивалентны. Второе выражение — это уже не пустой список, а список, состоящий из одного элемента — атома *NIL*.

Из пустого списка нельзя выделить ни первый элемент, ни хвост списка, т. е. значения выражений (*CAR NIL*) и (*CDR NIL*) не определены.

1.11. Функция *CONS*

Разбирать списки на элементы мы умеем. А как можно построить список из составных частей (эта задача, как известно, всегда сложнее)? Воссоединить первый элемент с остатком списка можно

с помощью функции *CONS*, имеющей два аргумента. Например,

$$\begin{aligned}(CONS (QUOTE A) (QUOTE (B C))) &\rightarrow (A B C) \\(CONS (QUOTE A) (QUOTE (B))) &\rightarrow (A B) \\(CONS (QUOTE A) NIL) &\rightarrow (A)\end{aligned}$$

(в последнем примере при задании второго аргумента не было необходимости прибегнуть к помощи *QUOTE*),

$$\begin{aligned}(CONS (QUOTE (1 2)) (QUOTE (3 4))) \\ \rightarrow ((1 2) 3 4)\end{aligned}$$

Вообще, если значение второго аргумента функции *CONS* — список (а других возможностей мы пока не допускаем), то функция вставляет в этот список перед его первым элементом значение своего первого аргумента в качестве нового первого элемента.

Во всех случаях, когда к выражению *x* применимы функции *CAR* и *CDR*, выражение

$$(CONS (CAR x) (CDR x))$$

эквивалентно выражению *x*. И наоборот, если функция *CONS* применима к выражениям *u* и *v*, то выражение

$$(CAR (CONS u v))$$

эквивалентно *u*, а выражение

$$(CDR (CONS u v))$$

эквивалентно *v*.

Дальнейшие п р и м е р ы:

$$\begin{aligned}(CONS (CAR (QUOTE (A B C))) (CDR \\ (QUOTE (1 2 3)))) &\rightarrow (A 2 3) \\(CONS (CAR (QUOTE (A B C))) \\ (CONS (CADR (QUOTE (P Q R))) \\ (CDDR (QUOTE (1 2 3))))) &\rightarrow (A Q 3) \\(CONS (QUOTE P) (CONS (QUOTE Q) (CONS \\ (QUOTE R) (CONS (QUOTE S) \\ NIL)))) &\rightarrow (P Q R S) \\(CONS (CONS (QUOTE P) (CONS (QUOTE Q) NIL)) \\ (CONS (CONS (QUOTE R) (CONS \\ (QUOTE S) NIL)) NIL)) &\rightarrow ((P Q) (R S))\end{aligned}$$

Обилие обращений к функции *QUOTE* в этих примерах неизбежно, но вообще для языка лисп не типично. В реальных программах списки, заданные явно, встречаются гораздо реже, поэтому к функции *QUOTE* часто прибегать не приходится.

1.12. Логические значения и предикаты

Принимая какое-то решение, мы обычно исходим из суждения типа: «дискриминант уравнения неотрицателен», «элемент *A* содержится в данном списке», «язык хинди интересен» и т. п. Содержащиеся в них утверждения могут быть верными или ошибочными. Про верные утверждения говорят, что они имеют значение «истина», про ошибочные, что их значение — «ложь». «Истина» и «ложь» называются *логическими значениями*.

Для изображения логических значений введены специальные атомы — константы. «Истина» обычно изображается атомом *T* (первая буква английского слова true — истина), а «ложь» принято изображать атомом *NIL*. Как же узнать, что изображает атом *NIL*, если он встретился в качестве значения какого-либо выражения, — пустой список или логическое значение «ложь»? Дело в том, что такой вопрос обычно не возникает, потому что мы заранее знаем, какой класс объектов должен быть представлен значением данного выражения. Если значение должно быть списком, то *NIL* обозначает пустой список; если значение должно быть логическим, то *NIL* обозначает «ложь».

В языке принято, что значение *T* равно *T*. Такое соглашение избавляет нас от необходимости писать *(QUOTE T)*, когда мы хотим явно изобразить значение «истина». Можно писать просто *T*.

Функции, вырабатывающие логические значения, называются *предикатами*, выражения, принимающие только эти значения, — *логическими выражениями*. Атом *T* — это стандартное обозначение логического значения «истина». Однако во всех случаях использования логических выражений за «истину» принимается любое значение, отличное от *NIL*, а не только значение *T*. Благодаря этому соглашению фактически любая функция может служить предикатом, а любое выражение можно считать логическим.

1.13. Функция АТОМ

Очень часто применяемая функция *АТОМ* может служить примером предиката. Ее аргументом (единственным) может быть любое выражение. Значение функции равно *T*, если значение аргумента — атом, и *NIL*, когда значение аргумента — не атом.

Примеры:

```
(ATOM (QUOTE  
      ABCDEFGHIJKLMNOPQRSTUVWXYZ)) → T  
(ATOM (CAR (QUOTE (NIL T)))) → T  
(ATOM (CDR (QUOTE (NIL T)))) → NIL  
(ATOM (CONS x y)) → NIL
```

(в последнем примере x и y — произвольные выражения, имеющие значения).

Предикат *АТОМ* применяется, как правило, в следующей ситуации. Анализируя структуру выражения, расчлняя его на элементы, мы должны знать, когда дальнейшее членение уже невозможно. Ответ на этот вопрос и дает функция *АТОМ*. Правда, пока еще не известно, как воспользоваться полученным ответом, но скоро мы это узнаем.

Каково значение выражения (*АТОМ NIL*) или, что то же самое, (*АТОМ ()*)? Ведь $()$ — это список, хотя и пустой. Условились считать, что это значение равно T независимо от формы записи, т. е. пустой список считается атомом.

1.14. Функция *EQ*

Другим фундаментальным предикатом является функция *EQ*. Она дает ответ на вопрос, нашли ли мы то, что искали. У функции *EQ* два аргумента, значение по крайней мере одного из них должно быть атомом. Функция принимает значение T , если значение другого аргумента тоже атом и оба значения совпадают, и значение NIL во всех остальных случаях, когда значения аргументов определены.

Примеры:

$(EQ\ T\ NIL) \rightarrow NIL$

$(EQ\ T\ (QUOTE\ T)) \rightarrow T$

$(EQ\ (CAR\ (QUOTE\ (A\ A)))\ (CDR\ (QUOTE\ (A\ A)))) \rightarrow NIL$

$(EQ\ (CAR\ (QUOTE\ (A\ A)))\ (CAR\ (QUOTE\ (A\ B\ C\ D)))) \rightarrow T$

Однако значение функции

$(EQ\ (CDR\ (QUOTE\ (A\ C)))\ (CDR\ (QUOTE\ (B\ C))))$

не определено, потому что значения аргументов — не атомы. Чему равно значение выражения

$(EQ\ x\ x)?$

Оно равно T , если x — константа или переменная, значение которой — атом. Оно не определено, если значение x — не атом или если x не имеет значения.

1.15. Условные выражения

Сейчас мы определим функцию *COND*, в которой главным образом и используются предикаты. Правда, эта функция особого рода, непохожая на те функции, которые были описаны до сих пор. Во-первых, число ее аргументов может быть произвольным. Во-вторых, значения этих аргументов не вычисляются перед началом вычисления функции, а берутся в том виде, в каком они написаны (это свойство роднит функцию *COND* с функцией *QUOTE*). Более того, значения аргументов функции *COND* и не могут быть вычислены. Каждый аргумент — это список из двух элементов. Первый элемент используется как логическое выражение, второй — как выражение, значение которого может стать значением функции *COND*. Таким образом, общий вид обращения к функции *COND* таков:

$$(COND (p_1 e_1) (p_2 e_2) \dots (p_n e_n)) \quad (1)$$

где p_1, p_2, \dots, p_n — логические, а e_1, e_2, \dots, e_n — произвольные выражения.

Функция *COND* служит для того, чтобы из выражений e_1, e_2, \dots, e_n выбрать одно и вычислить его значение в качестве значения функции. Выбор происходит следующим образом. Вычисляем значение выражения p_1 . Если получаем любое значение, отличное от *NIL*, то выбираем выражение e_1 . Если же p_1 — *NIL*, то переходим к следующему аргументу — списку $(p_2 e_2)$. Если значение p_2 — не *NIL*, то выбираем выражение e_2 ; если — *NIL*, то продолжаем перебирать аргументы, пока не встретится аргумент $(p_i e_i)$ со значением p_i , отличным от *NIL*; тогда выбор останавливаем на выражении e_i . Процесс завершается вычислением значения выбранного выражения e_i . Если значения всех логических выражений p_1, \dots, p_n равны *NIL*, то значение функции *COND* также равно *NIL*. Однако это соглашение принято не во всех версиях языка, поэтому в качестве выражения p_n обычно ставят константу *T*.

Напишем выражение, принимающее значение *T*, если значение переменной *Y* — не атом, и значение *NIL*, если значение *Y* — атом (т. е. выражение со значением, противоположным значению (*ATOM Y*)):

$$(COND ((ATOM Y) NIL) (T T))$$

Используя полученный опыт, составим выражение, которое исследует значение переменной *X* по следующим правилам. Если это значение — атом или список из одного элемента, в качестве результата должен быть выработан *NIL*; если один из двух начальных эле-

ментов значения X — список, то результатом должен служить атом S ; если эти два элемента — один и тот же атом, то результатом должен быть этот атом; в остальных случаях в качестве результата следует принять остаток значения X после отбрасывания первых двух элементов. Выражение может быть таким:

```

(COND ((ATOM X) NIL)
1      23      3      2
      ((EQ (CDR X) NIL) NIL)
      23      4      4      3      2
      ((COND ((ATOM (CAR X)) NIL)
      23      45      6      65      4
              (T T)) (QUOTE S))
              4      43      3      32
      ((COND ((ATOM (CADR X)) NIL)
      23      45      6      65      4
              (T T)) (QUOTE S))
              4      43      3      32
      ((EQ (CAR X) (CADR X))
      23      4      4      4      43
              (CAR X))
              3      32
      (T (CDDR X)))
      2      3      321

```

Другой вариант:

```

(COND ((ATOM X) NIL)
      ((EQ (CDR X) NIL) NIL)
      ((ATOM (CAR X)) (COND
        ((EQ CAR X) (CADR X)) (CAR X))
        ((ATOM (CADR X)) (CDDR X))
        (T (QUOTE S)) ))
      (T (QUOTE S)))

```

Обращения к функции *COND* называются *условными выражениями*.

1.16. Определяющие выражения функций

Можно ли сказать, что выражение

(COND ((ATOM Y) NIL) (T T)) (1)

определяет функцию? На первый взгляд — да, потому что оно позволяет вычислять требуемое значение (*NIL* или *T*), если Y обладает некоторым значением. Но, с другой стороны, оно ничего не говорит о том, сколько аргументов у этой функции, как они обозначены и как можно задать значения этим аргументам. Все эти

сведения содержатся в так называемом *определяющем выражении* функции, которое имеет следующий общий вид:

$$(LAMBDA (x_1 x_2 \dots x_m) e) \quad (2)$$

где x_1, \dots, x_m — атомы, изображающие переменные, от которых зависит определяемая функция (*связанные* переменные этой функции), e — выражение, в которое входят эти переменные и которое служит для вычисления значения функции, после того как с этими переменными будут связаны подходящие значения. Выражение e иногда называют *телом* определяющего выражения (2).

Пусть мы хотим определить функцию, вычисляемую с помощью выражения (1) и зависящую от одной переменной Y . Для этого мы должны написать такое определяющее выражение:

$$(LAMBDA (Y) (COND ((ATOM Y) NIL) (T T))) \quad (3)$$

Чтобы определить функцию двух переменных U и V , строящую список из первого элемента значения U и хвоста (остатка после отбрасывания первого элемента) значения V , надо воспользоваться определяющим выражением

$$(LAMBDA (U V) (CONS (CAR U) (CDR V)))$$

Определяющее выражение нельзя вычислить, оно не имеет значения. В этом смысле *LAMBDA* не следовало бы считать наименованием функции. Однако само определяющее выражение может быть использовано вместо наименования функции. Обращение к функции имеет при этом вид

$$((LAMBDA (x_1 x_2 \dots x_m) e) a_1 a_2 \dots a_m) \quad (4)$$

где a_1, \dots, a_m — выражения, значения которых следует связать с переменными x_1, \dots, x_m . Подобное обращение к функции выполняется следующим образом. Переменные x_1, \dots, x_m попарно связываются со значениями v_1, \dots, v_m выражений a_1, \dots, a_m . Вычисляется значение выражения e . Когда при этом вычислении встречается переменная x_i , в качестве ее значения берется связанное с ней значение v_i . По окончании вычисления связи между переменными x_i и их временными значениями v_i разрушаются, и если эти переменные были ранее связаны с какими-либо другими значениями, то эти связи восстанавливаются. В качестве значения выражения (4) принимается вычисленное значение выражения e .

Другими словами, механизм связи можно описать, сказав, что за время вычисления выражения e в него вместо каждой из переменных x_i подставляется выражение (*QUOTE* v_i) всюду, где эта переменная встречается. Мы написали (*QUOTE* v_i), а не a_i , потому что

значение выражения a_i вычисляется и связывается с переменной x_i только один раз. Существуют выражения, которые при повторных вычислениях поставляют разные значения. Таким образом, $(QUOTE v_i)$ и a_i — это не одно и то же.

Итак, выражение

$$((LAMBDA (U V) (CONS (CAR U) (CDR V))) \\ (QUOTE (P Q R)) (QUOTE (PP QQ RR)))$$

эквивалентно выражению

$$(CONS (CAR (QUOTE (P Q R))) \\ (CDR (QUOTE (PP QQ RR))))$$

и, следовательно, имеет значение $(P QQ RR)$.

Сказанное здесь будет уточнено в разд. 1.25.

1.17. Функция *NULL*

Чтобы выяснить, не является ли значение выражения X пустым списком, в языке существует предикат *NULL*. Он может быть задан следующим определяющим выражением:

$$(LAMBDA (X) (EQ X NIL))$$

Заметим, что если выражение X — логическое, то выражение $(NULL X)$ принимает логическое значение, противоположное значению X . Например, выражение $(NULL (ATOM X))$ принимает значение *T*, если значение X — не атом, и *NIL* во всех прочих случаях. Поэтому для функции *NULL* часто используют второе обозначение — *NOT*.

1.18. Встроенные и определяемые функции

Все функции, которые были рассмотрены до сих пор, — это *встроенные* функции. Программист не должен давать им определения, чтобы воспользоваться ими в программе, а может писать прямо обращения к ним. Эти функции вычисляются по подпрограммам, написанным на машинном языке, происходит это сравнительно быстро.

Говорят, что эти функции обладают *свойством* (снабжены *признаком*) *SUBR* или *FSUBR* (от английского subroutine — подпрограмма). Мы опишем еще несколько встроенных функций, но, как бы ни был велик их набор, интересные программы должны использовать функции, определяемые программистом специально для решения стоящей перед ним задачи. Такие функции вводятся в про-

грамму с помощью определяющих выражений и снабжаются одним из признаков *EXPR* или *FEXPR* (от expression — выражение). Встроенные функции, которые будут описаны в ближайших разделах, служат для того, чтобы сделать аппарат определяющих выражений более гибким и эффективным. Но предварительно разъясним, что значит буква *F* в признаках *FSUBR* и *FEXPR*.

1.19. Обычные и специальные функции

Функция считается *обычной*, если она имеет фиксированное число аргументов и если действия, указанные в ее определении, выполняются над значениями аргументов. Таковы функции *CAR*, *CDR*, *CONS*, *ATOM*, *EQ*.

Если же хотя бы одно из этих двух условий нарушается, то функция называется *специальной*. Такова, например, функция *QUOTE*. Она имеет определенное число аргументов — один аргумент, но применяется она не к значению аргумента, а к аргументу в том виде, в каком он написан. Слово «применяется» в данном случае не вполне уместно, потому что больше ничего эта функция с аргументом, как мы знаем, не делает. Но предыдущая фраза написана не ради того, чтобы еще раз напомнить о смысле функции *QUOTE*, а в качестве примера аналогичной ситуации в обычном языке. Глагол «применяется», поставленный в кавычки, перестает играть в этой фразе роль глагола, обозначающего некоторое действие (сравните с вариантом: «Слово применяется в данном случае не вполне уместно»), — он обозначает сам себя, становится частью подлежащего. Это редкий случай обращения со словом, но иногда необходимый в естественных языках. Заметим, что глагол «to quote» означает «брать в кавычки», откуда и происходит наименование функции *QUOTE*. Ситуации, когда аргументы лисповских функций не должны вычисляться, а должна использоваться лишь их внешняя форма, сравнительно редки. Их можно было бы совсем исключить, потребовав, чтобы к аргументам в случае необходимости применялась функция *QUOTE*. Именно так мы и поступали во многих примерах. Но есть функции, для которых такое использование аргументов типично и даже обязательно. Применять каждый раз *QUOTE* к каждому аргументу было бы обременительно. Некоторые из них, в частности все определяемые функции класса *FEXPR*, только тем и отличаются от обычных функций, что ко всем их аргументам неявно применяется функция *QUOTE*.

В общем случае функции класса *FSUBR* могут отличаться от обычных встроенных функций еще и тем, что число аргументов для них может быть произвольным, как, например, у *COND*. Неко-

торые же из них (*LAMBDA* и др.) вообще называются функциями лишь потому, что их наименования могут появляться в позиции наименования функции. Обращения к таким функциям никогда не вычисляются и не могут быть вычислены, так как они не имеют значений.

1.20. Функции *SEXPR* и *SFEXPR*

Эти функции связывают наименования функций с определяющими выражениями. Они имеют по два аргумента, принадлежат к классу *FSUBR*, а обращения к ним имеют вид

$$\begin{aligned} &(\text{SEXPR } fn \ de) \\ &(\text{SFEXPR } fn \ de) \end{aligned}$$

где *fn* — определяемое наименование функции (атом), а *de* — определяющее выражение для этой функции. Функция *SEXPR* снабжает наименование *fn* признаком *EXPR*, а функция *SFEXPR* — признаком *FEXPR*. Значением обеих функций является наименование *fn*. Но смысл их выполнения — не в выработке этого значения, а в том побочном эффекте, который они производят, в их влиянии на последующее выполнение программы.

Пусть одна из этих функций связала наименование *fn* с определяющим выражением *de*, которое имеет вид

$$(\text{LAMBDA } (x_1 \ x_2 \ \dots \ x_m) \ e)$$

Тогда становится возможным обращение к функции *fn* с помощью выражения вида

$$(fn \ a_1 \ a_2 \ \dots \ a_m) \tag{1}$$

(число *m* аргументов *a_i* должно быть равно числу переменных *x_i* в определяющем выражении *de*). Если *fn* обладает свойством *EXPR* (т. е. введена с помощью функции *SEXPR*), то обращение (1) как бы заменяется обращением

$$(de \ a_1 \ a_2 \ \dots \ a_m)$$

Если же *fn* было снабжено признаком *FEXPR*, то вместо обращения (1) как бы выполняется обращение

$$(de \ (\text{QUOTE } a_1) \ (\text{QUOTE } a_2) \ \dots \ (\text{QUOTE } a_m))$$

Пусть, для примера, требуется определить функцию *NATOM* переменной *Y*, вычисляемую с помощью выражения (1.16.3). Это

можно сделать, обратившись к функции *SEXPR*:

$$(SEXPR\ NATOM\ (LAMBDA\ (Y)\ (COND\ ((ATOM\ Y)\ NIL)\ (T\ T)\)))$$

Если теперь обратиться к функции *NATOM*, например, с помощью выражения

$$(NATOM\ (QUOTE\ A)) \tag{2}$$

то оно выполнится как выражение

$$((LAMBDA\ (Y)\ (COND\ ((ATOM\ Y)\ NIL)\ (T\ T)))\ (QUOTE\ A)\)$$

Это обращение свяжет переменную *Y* с атомом *A* — значением выражения *(QUOTE A)*, после чего начнет вычисляться выражение

$$(COND\ ((ATOM\ Y)\ NIL)\ (T\ T)) \tag{3}$$

В процессе вычисления встретится выражение

$$(ATOM\ Y), \tag{4}$$

для вычисления которого потребуется значение *Y*. Этим значением будет атом *A*, следовательно, значение выражения (4) будет равно *T*, а значение условного выражения (3) — *NIL*. После этого связь между *Y* и *A* будет разорвана (а если ранее *Y* было связано с каким-то другим значением, то эта прежняя связь восстановится) и *NIL* выдастся в качестве значения выражения (2).

Другой п р и м е р:

$$(SEXPR\ PAIR\ (LAMBDA\ (U\ V)\ (CONS\ U\ (CONS\ V\ NIL))))$$

Функция *PAIR*, вводимая этим выражением, строит список из значений двух своих аргументов, например,

$$(PAIR\ (QUOTE\ (ALPHA\ BETA))\ (QUOTE\ (GAMMA\ DELTA))) \rightarrow \\ \rightarrow ((ALPHA\ BETA)\ (GAMMA\ DELTA))$$

Если функцию с тем же самым определяющим выражением ввести посредством *SFEXPR*:

$$(SFEXPR\ PAIRQ\ (LAMBDA\ (U\ V)\ (CONS\ U\ (CONS\ V\ NIL))))$$

то вычисление значений аргументов будет заблокировано. Обращение с явно заданными аргументами упростится:

$$(PAIRQ\ (ALPHA\ BETA)\ (GAMMA\ DELTA)) \rightarrow \\ \rightarrow ((ALPHA\ BETA)\ (GAMMA\ DELTA)).$$

Но при аргументах, заданных выражениями, получаем

$$\begin{aligned} & (PAIRQ (CAR (QUOTE (A B))) \\ & \quad (CDR (QUOTE (C D)))) \\ & \rightarrow ((CAR (QUOTE (A B))) (CDR (QUOTE (C D))))), \end{aligned}$$

тогда как функция *PAIR* дает

$$\begin{aligned} & (PAIR (CAR (QUOTE (A B))) \\ & \quad (CDR (QUOTE (C D)))) \rightarrow (A (D)) \end{aligned}$$

Вводить функции с помощью *SFEXPR* следует по возможности реже, так как их применение наталкивается на ряд затруднений, которые не всегда легко предвидеть и преодолеть (см. пример в разд. 1.28).

1.21. Рекурсивные функции

Функция называется *рекурсивной*, если в ее определяющем выражении содержится хотя бы одно обращение к ней самой. Такое обращение может быть неявным. Это значит, что обращение происходит к другой функции, а та прямо или также косвенно обращается к определяемой функции.

Лисп допускает использование рекурсивных функций безо всяких ограничений. Более того, вся сила языка заключена в возможности свободно пользоваться рекурсивными функциями. Продемонстрируем это на очень простом примере. Предположим, мы хотим проверить, является ли атом *X* элементом списка *Y*. Дадим функции-предикату, осуществляющей эту проверку, наименование *MEMB*. Ясно, что функция *MEMB* должна поочередно сравнивать *X* со всеми элементами списка *Y*, пока не найдет элемент, совпадающий с *X*, или не исчерпает весь список. В первом случае ее значением должно быть *T*, во втором — *NIL*. Но у нас нет средств для выборки из списка *Y* элемента с произвольным номером. Мы можем выбрать первый элемент (с помощью *CAR*), второй (с помощью *CADR*) и вообще любой элемент с любым известным заранее номером, используя для этого соответствующую комбинацию *CAR* и *CDR*. Но номер элемента должен быть переменным. Как быть? Решение очень просто. Если список *Y* пуст, то результатом должен быть *NIL* (элемента *X* список не содержит). В противном случае проверяем, не совпадает ли *X* с первым элементом *Y*. Если да, то результат — *T*. Если нет, то надо проверить, не содержится ли *X* в остатке списка *Y*. Для этого обращаемся к функции *MEMB* с аргументами *X* и *(CDR Y)*.

Определение функции *MEMB* выглядит так:

```
(SEXPR MEMB (LAMBDA (X Y) (COND
  ((EQ Y NIL) NIL)
  ((EQ X (CAR Y)) T)
  (T (MEMB X (CDR Y))))))
```

Это определение в точности следует данному перед этим словесному описанию.

Заслуживает внимания первая строчка этого определения. Она типична для многих рекурсивных функций:

```
(SEXPR fn (LAMBDA (x1 x2 ... xm) (COND
```

Здесь *fn* — наименование определяемой функции, x_1, \dots, x_m — наименования ее связанных переменных. Далее начинается условное выражение. В нем анализируются простейшие частные случаи, и если ни один из них не дает ответа, то следует прямое или косвенное обращение (или обращения) к той же функции для дальнейшего разбирательства. Но структура аргументов при этих обращениях становится чуточку проще. Так мы добираемся, в конце концов, до простой ситуации, предусмотренной начальными проверками.

У функции *MEMB* есть один существенный изъян — она неприменима, если *X* — не атом, а произвольное выражение. Поэтому определим функцию *EQUAL*, которая действует аналогично *EQ*, но применима к произвольным выражениям:

```
(SEXPR EQUAL (LAMBDA (X Y) (COND
  ((ATOM X) (EQ X Y)) ((ATOM Y) NIL)
  ((EQUAL (CAR X) (CAR Y))
   (EQUAL (CDR X) (CDR Y)))
  (T NIL) )))
```

Словесный эквивалент этого описания таков. Если значение *X* — атом, то его можно сравнить со значением *Y*, используя функцию *EQ*. Если только *Y* имеет своим значением атом, то совпадение значений *X* и *Y* невозможно. Если и *X* и *Y* — списки, то сравниваем их первые элементы. В случае их совпадения остается сравнить хвосты списков. Если же уже первые элементы у *X* и *Y* различны, то и сами *X* и *Y* не совпадают. Как первые элементы, так и хвосты списков *X* и *Y* могут быть сколь угодно сложными выражениями, поэтому их следует сравнивать, обращаясь к той же функции *EQUAL*.

Теперь можно определить функцию *MEMBER*, работающую аналогично *MEMB*, но позволяющую обнаруживать наличие в списке элементов любого вида. Такая функция может быть введена

в программу выражением

```
(SEXPR MEMBER (LAMBDA (X Y) (COND
  ((NULL Y) NIL)
  ((EQUAL X (CAR Y)) T)
  (T (MEMBER X (CDR Y))))))
```

Функции *EQUAL* и *MEMBER* весьма употребительны и поэтому существуют как встроенные во многих реализациях.

1.22. Функция *CSETQ*, константы

В языке *константы* лишь условно оправдывают свое наименование. Программист имеет возможность вводить новые константы и менять значения ранее введенных и даже определенных в самом языке констант (хотя и не следует уподобляться любознательному ребенку и пытаться изменить, скажем, значение константы *NIL* — ничего хорошего из этого не получится).

Объявить атом *c* константой, присвоив ей значение выражения *e*, можно, обратившись к функции *CSETQ*:

```
(CSETQ c e)
```

Добавленная впереди к наименованию этой функции буква *C* (от *constant* — константа) напоминает об отличии этой функции от функции *SETQ*, присваивающей новые значения переменным (см. разд. 1.24). Старое значение *c*, если оно имелось, пропадает. Новое значение сохраняется до тех пор, пока наименованию *c* не будет присвоено новое значение. Заметим, что не только значение, но и свойство наименования быть константой, не вечно — при очередном присваивании *c* может стать наименованием функции, если его употребить в выражении

```
(SEXPR c de)
```

или

```
(SFEXPR c de)
```

Связь, налагаемая на наименования констант функцией *CSETQ*, сильнее, чем связь, накладываемая на наименования связанных переменных функцией *LAMBDA*. Функция *LAMBDA* не может ни изменить значение константы, ни превратить ее, хотя бы временно, в переменную. Поэтому бесполезно пытаться использовать наименования констант (в том числе встроенных) в качестве связанных переменных. Пример на применение этого правила дан в следующем разделе.

1.23. Программа

Лисповская программа состоит из произвольной последовательности обращений к функциям. Эти обращения выполняются одно за другим в том порядке, в каком они входят в программу. Ни одно обращение к функции f класса $EXPR$ и $FEXPR$ не может быть выполнено раньше, чем обращение к функции $SEXPR$ или $SFEXPR$, которое определит функцию f и сделает ее доступной программе.

Более точно, программа выполняется следующим образом. Читается и сразу же печатается очередное выражение программы. Затем вычисляется значение этого выражения. По окончании вычисления выражения печатается его значение. Затем все повторяется с очередным еще не прочитанным выражением.

Пр и м е р программы:

$(CSETQ\ N\ (QUOTE\ A))$ (1)

$(SEXPR\ F\ (LAMBDA\ (T\ N)$ (2)
 $(CONS\ T\ (CONS\ N\ NIL))\))$

$(F\ (QUOTE\ B)\ (QUOTE\ C))$ (3)

В результате выполнения этой программы будут отпечатаны:

выражение (1)

N (значение выражения (1))

выражение (2)

F (значение выражения (2))

выражение (3)

$(T\ A)$ (значение выражения (3))

В соответствии со сказанным в предыдущем разделе значением выражения (3) является именно список $(T\ A)$, а не список $(B\ C)$, как это было бы, если бы для связанных переменных функции F были выбраны другие обозначения, не являющиеся наименованиями констант.

Этим разделом мы заканчиваем описание так называемого *элементарного* или *базового* лиспа. Все понятия и средства программирования, которые будут введены в последующих разделах, могут быть так или иначе заменены средствами элементарной части языка. Однако расширение, описанное ниже, значительно облегчает пользование языком, делает его более эффективным и гибким. Следует отметить, что в разных реализациях языка такое расширение осуществляется по-разному, с добавлением большего или меньшего количества новых средств программирования. В нашем описании мы ориентируемся на те средства, которые включены в действующую реализацию лиспа на машине БЭСМ-6 [6].

Строго говоря, уже при описании элементарной части языка мы допустили отступления от системы понятий, введенных автором [15, 16]. В этой системе вместо функции *SEXPR* есть иной аппарат (функция *LABEL*), позволяющий давать определения рекурсивным функциям, но только в том месте программы, где требуется к ним обратиться, и только на период обращения. Хотя теоретически этого достаточно, но на практике совершенно необходимо запастись определениями функций впрок, составляя программу решения задачи по частям. В авторском варианте базового языка программист лишен также возможности определять специальные функции и заводить константы. Но даже в нашем варианте базовый язык не содержит многого, без чего он не может считаться практическим языком программирования.

1.24. Аппарат *PROG*

Тот, кто знаком с алголом, ощущает некоторое неудобство, вызванное отсутствием столь привычных средств записи программ, как оператор присваивания, оператор перехода, метка, описание типа (переменных). При отсутствии опыта может показаться, что писать программы, состоящие исключительно из одних описаний процедур и указателей функций, вообще невозможно. Конечно, это не так, но подобный стиль программирования вызывает некоторые трудности как практического, так и психологического характера. Поэтому в язык включен аппарат программирования, предоставляющий все упомянутые возможности. Общий вид выражения, приводящего этот аппарат в действие, таков:

$$(PROG (u_1 \dots u_k) s_1 \dots s_r) \quad (1)$$

Здесь *PROG* — наименование специальной функции, u_1, \dots, u_k — атомы, называемые *программными* переменными, s_1, \dots, s_r — *операторы*. Каждый оператор может быть либо обращением к функции, либо атомом. Обращение к функции, если оно занимает позицию оператора, выполняется только ради его побочного эффекта, а не ради значения, которое никак не используется. Оператор-атом изображает *метку*, его выполнение не влечет никаких действий.

Выполнения обращения (1) к функции *PROG* состоит в том, что всем программным переменным u_1, \dots, u_k присваивается начальное значение *NIL* и затем начинают выполняться операторы s_1, s_2, \dots, s_r в том порядке, в каком они написаны. Однако этот порядок может быть изменен *операторами перехода*, имеющими вид обращения к специальной функции *GO*:

$$(GO \ l)$$

где l — атом, встречающийся среди операторов s_1, \dots, s_r в роли метки. После такого обращения выполнение последовательности операторов продолжается с оператора, следующего за меткой l .

Среди операторов может быть помещен оператор выхода из *PROG* вида (*RETURN* x), где x — выражение. Выполнить этот оператор означает — закончить выполнение выражения (1), приняв в качестве результата значение выражения x . После выполнения последнего в выражении (1) оператора s_r , если это не оператор перехода или выхода, выполнение выражения (1) также заканчивается, но его значением считается *NIL*.

Среди операторов может встретиться условный оператор вида (*COND* ($p_1 t_1$) ($p_2 t_2$) ... ($p_n t_n$)), отличающийся от условного выражения (1.15.1) тем, что в его состав вместо выражений e_1, \dots, e_n входят операторы t_1, \dots, t_n . Выполняется условный оператор так же, как условное выражение, т. е. вычисляются логические выражения p_1, p_2, \dots, p_n , пока среди них не встретится выражение p_i , имеющее значение, не равное *NIL*. Тогда выполняется соответствующий оператор t_i и на этом выполнение условного оператора заканчивается. Не требуется, чтобы хоть одно из выражений p_i имело истинное значение. Выполнение условного оператора может исчерпаться тем, что будут вычислены значения всех выражений p_i и все они окажутся равными *NIL*. В позициях операторов t_i разрешается помещать операторы *GO*, *RETURN*, *SETQ* и даже снова *COND*, а также обращения к любым другим функциям, если такие обращения имеют смысл (см. ниже). Оператор *COND* внутри условного оператора записывается и выполняется по тем же правилам, что и на внешнем уровне.

В некоторых версиях языка лисп допускаются более общие конструкции условных операторов, а именно:

$$(\text{COND } (p_1 t_{11} \dots t_{1m_1}) \dots (p_n t_{n1} \dots t_{nm_n}))$$

После того как найдено значение p_i , не равное *NIL*, последовательность операторов $t_{i1} \dots t_{im_i}$ выполняется вместо данного условного оператора. Подробнее это описано в разд. 2.9.

Про операторы *GO* и *RETURN* (перехода и выхода) говорят, что они сами определяют своего преемника. Условный оператор также может определить своего преемника, если при его выполнении был выбран и выполнен оператор, определивший этого преемника. Если же выполненный оператор s_i в последовательности

$$s_1 s_2 \dots s_r$$

не определил своего преемника и не был последним в этой последо-

вательности, то вслед за ним начинается выполнение следующего по порядку оператор s_{i+1} .

Роль оператора присваивания играет оператор вида

$(SETQ\ u\ e)$,

где u — переменная, e — выражение, значение которого присваивается (связывается с) переменной u . Эта связь сохраняется до следующего присваивания этой переменной или до завершения вычисления выражения (1). Значение выражения $(SETQ\ u\ e)$ совпадает со значением e . Поэтому, если требуется присвоить значение выражения e переменной u и одновременно использовать это выражение в других целях, например, в качестве аргумента в обращении к функции $(F\ e)$, то это можно сделать, написав выражение

$(F\ (SETQ\ u\ e))$

Список программных переменных в выражении (1) обязателен, хотя бы пустой. Он играет роль описания переменных u_1, \dots, u_k , т. е. указывает, какими новыми переменными мы хотим (и получаем право) пользоваться в данном обращении к *PROG*.

Если среди операторов встретится обращение к функции с наименованием, отличным от *SETQ*, *GO*, *RETURN* и *COND*, то оно выполняется, как обычно, но вычисленное значение функции нигде не используется. Таким образом, в роли операторов имеют смысл обращения лишь к таким функциям, выполнение которых сопровождается каким-либо побочным эффектом (например, *SEXPR* или *PRINT* — см. разд. 1.27).

Обращение к функции *PROG* применяется чаще всего в составе определяющего выражения (1.16.2) в качестве его тела e . В качестве примера опишем заново функцию *MEMB*:

```
(SEXPR MEMB (LAMBDA (X Y) (PROG ()  
  L (COND ((NULL Y) (RETURN NIL))  
          ((EQ X (CAR Y)) (RETURN T)) )  
  (SETQ Y (CDR Y)) (GO L) )))
```

Два последних оператора — *SETQ* и *GO* — обеспечивают продолжение поиска атома X среди оставшихся элементов списка Y .

Два описания функции *MEMB* не сильно отличаются друг от друга. Описание, использующее механизм *PROG*, несколько длиннее. Однако функции, описанные с помощью этого механизма, иногда вычисляются быстрее, так как позволяют уменьшить число выполняемых обращений к функциям. Это особенно проявляется, если определения функций компилируются — переводятся на машинный

язык (см. разд. 2.14). Значительно экономнее используется память машины.

Не всегда описания функций с использованием *PROG*, но без рекурсий оказываются столь близкими по структуре к рекурсивным описаниям, как в случае функции *MEMB*. В разд. 1.26 мы убедимся в этом на примерах.

1.25. Переменные

Два класса переменных: связанные переменные в определяющих выражениях и программные переменные в выражениях *PROG*, в общем, совершенно равноправны. Каждая переменная может быть связана со значением и даже со многими значениями одновременно. Однако в каждый момент выполнения программы лишь одна из этих связей — та, которая была установлена последней, — находится в *активном* состоянии. Это значит, что в качестве значения переменной, когда оно потребуется, берется значение, соединенное с ней активной связью. Это значит также, что при присваивании переменной нового значения посредством оператора *SETQ* активная связь начинает связывать переменную с этим новым значением, а старое теряется безвозвратно. Значения же, связанные с переменной *пассивными* связями, не могут быть ни использованы в вычислениях, ни заменены другими значениями, пока соответствующая связь не перейдет в активное состояние.

В момент, когда начинается выполнение обращения к функции вида

$$((\text{LAMBDA } (x_1 \ x_2 \ \dots \ x_m) \ e) \ a_1 \ a_2 \ \dots \ a_m) \quad (1)$$

устанавливаются новые активные связи переменных x_1, \dots, x_m со значениями выражений a_1, \dots, a_m . Если некоторые (или все) из этих переменных были до этого момента активно связаны с другими значениями, то эти связи из активного состояния переходят в пассивное. Вновь установленные связи сохраняются до окончания выполнения выражения (1). За это время они могут любое число раз перейти из активного состояния в пассивное и обратно. К концу вычисления выражения (1) эти связи обязательно оказываются в активном состоянии. В период активного состояния значения, связанные с переменными x_1, \dots, x_m , могут неоднократно изменяться в результате выполнения операторов вида *(SETQ x_i e)*. В момент, когда выполнение выражения (1) завершается, все активные связи переменных x_1, \dots, x_m с их значениями разрушаются, перестают существовать. Если у этих переменных имеются пассивные связи,

то те из них, которые были активными к началу выполнения выражения (1), вновь становятся активными.

То же самое происходит с программными переменными u_1, \dots, u_k , когда начинается и когда заканчивается выполнение выражения

$$(PROG (u_1 u_2 \dots u_k) s_1 s_2 \dots s_r) \quad (2)$$

Единственное отличие заключается в том, что вновь устанавливаемые связи начинают связывать переменные u_1, \dots, u_k со значением NIL . Заметим, что одно и то же наименование (атом) может одновременно фигурировать в нескольких выражениях вида (1) и (2). Тогда во время выполнения программы оно поочередно, в произвольной последовательности может выступать то в роли связанной переменной, то в роли программной переменной. Текущая история этих событий сохраняется в виде пассивных связей данного наименования.

Из сказанного ясно, что в операторе $(SETQ u e)$ в качестве переменной u может стоять любая переменная, для которой к моменту выполнения этого оператора существует хотя одна связь (и, значит, ровно одна активная связь). Этот оператор заменяет значение, связанное с переменной активной связью, значением выражения e .

1.26. Приемы программирования

Чтобы лучше познакомиться с основными приемами программирования на лиспе, составим несколько вариантов программы для решения следующей задачи. Требуется переставить элементы списка в обратном порядке, причем, если элемент сам является списком, то с ним продельвается та же операция, и так на всех уровнях. Например, список

$$(A B (C (D E)) F) \quad (1)$$

должен перейти в $(F ((E D) C) B A)$.

Для решения задачи естественно воспользоваться следующим алгоритмом. Пусть X обозначает еще не перевернутую часть (хвост) данного списка, а Y — уже перевернутую часть. Так, в один из моментов работы этого алгоритма над списком (1) значением X может быть список $((C (D E)) F)$, а значением Y — список $(B A)$. В начале работы алгоритма значение X совпадает с данным списком, а список Y пуст. Когда X окажется пустым списком, в Y накопится требуемый результат. Если же X непуст, то следует выделить (и вычеркнуть) из X первый элемент, перевернуть его, если он — не атом, и вставить в начало списка Y .

Такую функцию описать нетрудно, неясно только, откуда взять переменную *Y* — ведь по смыслу задачи функция, ее решающая, должна быть функцией одной переменной. В таких ситуациях, кроме основной функции, к которой происходит первоначальное обращение, описывают вспомогательную функцию с нужным количеством переменных. Для нашей задачи вспомогательная функция — обозначим ее *REV1* — может быть описана так:

```
(SEXPR REV1 (LAMBDA (X Y) (COND
  ((NULL X) Y)
  ((ATOM (CAR X)) (REV1 (CDR X)
    (CONS (CAR X) Y)))
  (T (REV1 (CDR X)
    (CONS (REV (CAR X)) Y) )) )))
```

Здесь *REV* — наша основная функция, которая теперь описывается просто:

```
(SEXPR REV (LAMBDA (X) (REV1 X NIL)))
```

Другая возможность — воспользоваться аппаратом *PROG*:

```
(SEXPR REV (LAMBDA (X) (PROG (Y U)
  A (COND ((NULL X) (RETURN Y)))
  (SETQ U (CAR X)) (SETQ X (CDR X))
  (COND ((NULL (ATOM U))
    (SETQ U (REV U)) ))
  (SETQ Y (CONS U Y)) (GO A) )))
```

Напомним, что начальное значение *NIL* переменная *Y* получает автоматически.

В этом описании переворачивание списка на внешнем уровне происходит без рекурсии, посредством циклического повторения последовательности операторов в *PROG*е. Но для переворачивания на внутренних уровнях используется рекурсия — обращение (*REV U*). Чтобы избавиться и от нее, следует, когда обнаруживается, что значение *U* — не атом, приостановить переворачивание списка на текущем уровне и начать переворачивать список *U*, используя те же операторы. Но надо сохранить текущие значения списков *X* и *Y*. Для этого присоединим их один за другим к еще одному вспомогательному списку *V* (к его началу). Когда очередной список перевернут, посмотрим, не пуст ли список *V*. Если нет, то восстановим с его помощью списки *X* и *Y* и продолжим переворачивание на предыдущем уровне. Если же *V* пуст, то это значит, что завершено переворачивание исходного списка.

В виде лисповской программы все сказанное выглядит так:

```
(SEXPR REV (LAMBDA (X) (PROG (Y U V)
  A (COND ((NULL X) (GO B)))
    (SETQ U (CAR X)) (SETQ X (CDR X))
    (COND ((ATOM U) (GO C)))
    (SETQ V (CONS Y (CONS X V)))
    (SETQ X U) (SETQ Y NIL) (GO A)
  B (COND ((NULL V) (RETURN Y)))
    (SETQ U Y) (SETQ Y (CAR V))
    (SETQ X (CADR V)) (SETQ V (CDDR V))
  C (SETQ Y (CONS U Y)) (GO A) )))
```

Это описание *REV* существенно длиннее и сложнее предыдущего. И использованный здесь прием избавления от рекурсии не оправдывает себя и потому, что в программу приходится явно включать по существу те же действия, которые при интерпретации рекурсивных функций выполняются автоматически (см. разд. 2.7 и 2.8).

Дадим еще один вариант описания функции *REV*, в котором функция *PROG* используется внутри условного выражения

```
(SEXPR REV (LAMBDA (X) (COND
  ((ATOM X) X)
  (T (PROG (U)
    A (COND ((NULL X) (RETURN U)))
      (SETQ U (CONS (REV (CAR X)) U))
      (SETQ X (CDR X)) (GO A) )) )))
```

Хотя этот вариант описания — самый короткий, в нем слишком много времени тратится на проверку того, что очередной элемент списка — атом, и его не нужно переворачивать.

1.27. Функции *READ*, *PRINT* и *GENSYM*

Функция *READ* без аргументов осуществляет перевод выражений из той формы, в какой их записывает программист, к виду, в котором они хранятся в памяти машины (см. разд. 2.1). Ее значением служит переведенное выражение (оно само, а не его значение). В качестве переводимого выражения берется очередное выражение программы. Все эти выражения входят, таким образом, в состав программы и вводятся в машину вместе с программой.

Функция *PRINT* с одним аргументом вычисляет значение своего аргумента, переводит его во внешнее представление и печатает этот перевод. Значение функции совпадает со значением аргумента. Оно может быть использовано в дальнейших вычислениях.

Обе эти функции работают в описанном в разд. 1.23 цикле выполнения программы. В принципе программист может обойтись без обращения к этим функциям. Но иногда они полезны для упрощения программы, для печати промежуточных результатов или отладочной информации и т. п.

Рассмотрим следующую программу:

```
(SEXPR REV (LAMBDA (X) (COND
  ((ATOM X) X) (T
    (REV1 (PRINT X) NIL))) ))) (1)
```

```
(SEXPR REV1 (LAMBDA (U V) (COND ((NULL U) V)
  (T (REV1 (CDR U) (CONS (REV
    (CAR U)) V))) ))) (2)
```

```
(REV (READ)) (3)
```

```
(A B (C (D E)) F) (4)
```

Здесь функция *PRINT* используется для печати аргументов всех обращений к функции *REV*, если эти аргументы — не атомы. Обращение (*READ*) в качестве аргумента функции в выражении (3) позволяет задать любой желаемый аргумент в виде отдельного выражения (не прибегая к *QUOTE*). Если вместо выражений (3) и (4) написать

```
(REV (QUOTE (A B (C (D E)) F))) (5)
```

то для повторения программы с другим значением аргумента надо переписать все выражение (5), тогда как при обращении к *REV* с помощью выражений (3) и (4) для смены аргумента обращения достаточно изменить лишь выражение (4).

Программа в ходе ее выполнения отпечатает:

выражение (1)

REV

выражение (2)

REV1

выражение (3)

<pre>(A B (C (D E)) F) (C (D E)) (D E) (F ((E D) C) B A)</pre>	}	(аргументы-списки последовательных обращений к функции <i>REV</i>) (значение выражения (3))
--	---	--

Функция *GENSYM* без аргументов создает атом, отличающийся от всех атомов, уже используемых в программе. Чтобы такой атом при выводе его на печать был легко заметен, обычно функция *GENSYM* вырабатывает атомы некоторого специального вида, например $Gd_1d_2d_3d_4d_5$, где d_1, \dots, d_5 — десятичные цифры.

1.28. Функция *EVAL*

Встроенная функция *EVAL* — это сердце системы, реализующей язык. Именно эта функция вычисляет значение любого выражения. В подавляющем большинстве случаев она применяется неявно. Например, при выполнении программы

```
(CSETQ A (QUOTE (P Q)))  
(CONS (CAR A) (CDR A))
```

происходит 7 неявных обращений к функции *EVAL* при вычислении значений:

функции *QUOTE*

функции *CSETQ*

константы *A* (в качестве аргумента *CAR*)

функции *CAR*

константы *A* (в качестве аргумента *CDR*)

функции *CDR*

функции *CONS*

Функция *EVAL* может быть применена для аннулирования действия функции *QUOTE*.

Если выражение *x* имеет значение, то оно совпадает со значением выражения (*EVAL (QUOTE x)*). Это можно использовать, например, следующим образом.

Пусть требуется определить функцию *F*, сравнивающую явно заданное выражение *x* со значением выражения *y*. Можно, конечно, воспользоваться выражением

```
(EQUAL (QUOTE x) y)
```

Но мы хотим, чтобы обращение к функции *F* имело вид

$(F\ x\ y).$ (1)

Попытка ввести функцию *F* выражением

```
(SEXPR F (LAMBDA (X Y) (EQUAL (QUOTE X) Y)))
```

конечно, не приводит к цели, так как при обращении (1) значение выражения *y* будет сравниваться не с выражением *x*, а с атомом *X*. Введем поэтому функцию *F* с помощью *SFEXPR* следующим образом:

```
(SFEXPR F (LAMBDA (X Y) (EQUAL X (EVAL Y))))
```

При этом аргументы *x* и *y* в обращении (1) неявно заключаются в кавычки, а обращение (*EVAL Y*) снимает эти кавычки, когда потребуется значение выражения *y*.

Еще один пример применения функции *EVAL*. Печать всех выражений (точнее, обращений к функциям), из которых состоит программа, полезна при отладке, но, когда программа отлажена, может возникнуть желание исключить эту печать. Для этого достаточно предпослать программе выражение

(PROG NIL A (PRINT (EVAL (READ)))) (GO A))

Само оно, конечно, отпечатается, а значения последующих выражений будут вычисляться и печататься без печати самих выражений. Вернуться к нормальному циклу можно, поместив среди этих выражений выражение

(RETURN NIL)

или заменив выражение *x* — последнее, печать которого мы хотим блокировать, выражением

(RETURN x)

1.29. Функция *LIST*

Эта встроенная функция имеет неопределенное число аргументов и по этой причине отнесена к классу *FSUBR*. Ее значением является список, составленный из значений аргументов, например,

(LIST (QUOTE A) (QUOTE B) (QUOTE C)) → (A B C)
(LIST (LIST (QUOTE PP) (QUOTE PQ))
(LIST (QUOTE QP) (QUOTE QQ))) →
→ ((PP PQ) (QP QQ))

То, что функция *LIST* предусматривает вычисление значений аргументов, роднит ее с обычными функциями класса *SUBR*. Чистоту принципа классификации функций можно спасти, приняв (и так оно есть на самом деле — см. разд. 2.8), что вычисление значений аргументов происходит не перед началом вычисления значения функции, ибо это недопустимо для функций класса *FSUBR*, а в процессе этого вычисления.

1.30. Предикаты *AND* и *OR*

Часто приходится проверять либо одновременное соблюдение нескольких условий, либо удовлетворение хотя бы одного из нескольких условий. Для этого можно несколько раз обратиться к функции *COND*, но лучше прибегнуть к одной из встроенных функций *AND* или *OR* класса *FSUBR*. Обращения к ним имеют вид

(*AND* $p_1 p_2 \dots p_n$) и (*OR* $p_1 p_2 \dots p_n$), где $p_1, p_2 \dots, p_n$ — логические выражения, число которых в обоих случаях произвольно.

Мы опишем действие этих функций, определив две другие функции *AND1* и *OR1*, дающие тот же эффект, но отличающиеся тем, что в обращении к ним задается лишь один аргумент — список логических выражений p_1, \dots, p_n .

```
(SFEXPR AND1 (LAMBDA (P) (COND ((NULL P) T)
  ((EVAL (CAR P)) (EVAL (LIST
    (QUOTE AND1) (CDR P) ))) (T NIL) )))
(SFEXPR OR1 (LAMBDA (P) (COND ((NULL P) NIL)
  ((EVAL (CAR P)) T)
  (T (EVAL (LIST (QUOTE OR1)
    (CDR P) ))) )))
```

Заметим, что функции *AND* и *OR* вычисляют в общем случае значения лишь части своих аргументов. Как только встретится аргумент со значением *NIL* (для функции *AND*) или не *NIL* (для *OR*), значение функции определяется и значения последующих аргументов не вычисляются. Это позволяет писать, например, такие выражения:

```
(OR (ATOM E) (NULL (CDR E)))
```

Второй аргумент не имеет смысла, если значение *E* — атом, но в этом случае он и не вычисляется.

Функции *AND* и *OR* часто позволяют упростить описание предикатов, особенно если учесть, что выражение (*NOT* (*NULL* *x*)), когда оно употребляется как логическое, можно заменить на *x*. Например, предикат *MEMBER* (см. разд. 1.21) допускает такое определение:

```
(SEXPR MEMBER (LAMBDA (X Y)
  (AND Y (OR (EQUAL X (CAR Y))
    (MEMBER X (CDR Y)) )) ))
```

1.31. Обобщение понятия выражения

До сих пор вторым аргументом функции *CONS* всегда был список. При этом значение этой функции также оказывается списком. Общности ради (а в некоторых случаях ради простоты) функцию *CONS* можно доопределить так, чтобы вторым аргументом мог быть атом. При этом результат уже не может быть списком, если мы хотим, чтобы применение функции *CDR* к этому результату восстанавливало значение второго аргумента. Для записи такого результата исполь-

зается специальное обозначение, называемое *точечным*. Пусть значения выражений x и y равны соответственно u и v :

$$x \rightarrow u, \quad y \rightarrow v$$

Тогда, каковы бы ни были u и v ,

$$(CONS\ x\ y) \rightarrow (u\ .\ v) \quad (1)$$

В правой части появился новый тип выражения, в котором как раз и применено точечное обозначение. Такое выражение называется *парой*. Формула (1) одновременно вводит новое обозначение и расширяет определение функции *CONS*. Аналогичная вещь происходит, когда мы, например, вводим обозначение i формулой $i = \sqrt{-1}$, расширяя в то же время область применимости операции извлечения квадратного корня. Как и в случае комплексных чисел, мы должны добиваться, чтобы новые объекты — выражения в точечной записи — подчинялись тем же основным правилам, что и ранее существовавшие. Для этого надо доопределить также функции *CAR* и *CDR* следующим образом:

$$\begin{aligned} (CAR\ (QUOTE\ (u\ .\ v))) &\rightarrow u \\ (CDR\ (QUOTE\ (u\ .\ v))) &\rightarrow v \end{aligned}$$

или, в более общей формулировке, если $x \rightarrow (u\ .\ v)$, то

$$(CAR\ x) \rightarrow u, \quad (CDR\ x) \rightarrow v$$

Кроме того, если формула (1) справедлива, когда v — атом, она должна быть справедлива и когда v — список. В частности, должно быть

$$(CONS\ (QUOTE\ Z)\ NIL) \rightarrow (Z\ .\ NIL)$$

С другой стороны, *NIL* — это обозначение для пустого списка, и по старым правилам

$$(CONS\ (QUOTE\ Z)\ NIL) \rightarrow (Z)$$

Поэтому записи $(Z\ .\ NIL)$ и (Z) следует считать эквивалентными. Далее, вычислим значение выражения

$$(CONS\ (QUOTE\ Y)\ (QUOTE\ (Z\ .\ NIL)))$$

В точечной записи получаем результат

$$(Y\ .\ (Z\ .\ NIL))$$

Если же заменить $(Z\ .\ NIL)$ на эквивалентное выражение (Z) и применить первоначальные правила, то получим в качестве результата

(Y Z)

Итак, записи $(Y . (Z . NIL))$ и $(Y Z)$ также надо признать эквивалентными. Аналогичными рассуждениями можно получить эквивалентную запись в точечных обозначениях для любого списка. Например, $(A B C)$ эквивалентно $(A . (B . (C . NIL)))$, выражение $(1 (2 3) (4 (5 6)))$ эквивалентно $(1 . ((2 . (3 . NIL)) . ((4 . ((5 . (6 . NIL)) . NIL)) . NIL)))$ и $((1000))$ эквивалентно $((1000 . NIL) . NIL) . NIL$.

Точечная запись обладает некоторыми преимуществами. Она симметрична по отношению к паре функций $CAR - CDR$, поэтому легче проследить, какой комбинацией этих функций следует воспользоваться для выделения того или иного элемента выражения. Она является более общей, так как любой список можно переписать в точечных обозначениях, но уже простейшее точечное выражение — пара $(A . B)$ — не может быть изображено в виде списка. Когда выражение строится из небольшого и, главное, заранее известного количества элементов, точечные конструкции предпочтительнее. Если же число элементов сравнительно велико и может быть переменным, то целесообразнее пользоваться списочными конструкциями. Хотя они и требуют для своего построения большего количества операций $CONS$, но зато допускают значительно более простую запись *).

Существует соглашение, позволяющее упростить запись и точечных конструкций — таких, при построении которых хотя бы один раз вторым аргументом операции $CONS$ оказывается не список. Оно основывается на следующих соображениях. Для возврата от точечной записи к списочной, когда такой возврат возможен, действует следующее правило. Если точка стоит перед открывающей скобкой, то ее можно заменить пробелом, выбросив одновременно эту открывающую и парную ей закрывающую скобку. Это же правило позволяет избавиться и от лишних NIL ов, если вспомнить, что NIL эквивалентен паре рядом стоящих скобок $()$. Упомянутое соглашение состоит в том, что это же правило разрешается применять к любому правильно построенному выражению в точечной записи, если даже оно не сводится в конечном счете к списку. Так, запись $(A . (B . C))$ эквивалентна записи $(A B . C)$, запись $((A . B) . (C . D))$ эквивалентна $((A . B) C . D)$. Легко убедиться, что сокращенная запись правильного выражения позволяет восстановить полную точечную

*) Авторы языка включают понятие пары в число базовых понятий.

запись. Для этого каждый пробел заменяется точкой, за которой ставится открывающая скобка. Соответствующая закрывающая скобка вставляется непосредственно перед ближайшей справа от этого пробела закрывающей скобкой, не имеющей парной открывающей скобки также справа от пробела.

В самом общем случае правильное выражение, если оно — не атом, представляет собой заключенную в скобки последовательность выражений, разделенных пробелами, причем, если последовательность содержит не менее двух выражений, то перед последним из них вместо пробела может стоять точка. Так, запись $(A . B C)$ не является правильным выражением, она не может быть получена по установленным правилам ни из списка, ни из правильного точечного выражения.

Пр и м е р ы:

$((11\ 12\ .\ 13)\ 2\ .\ (31\ (321\ 322\ 323\ .\ 324)\ 33\ 34))$

$(CADR\ (QUOTE\ (A\ B\ .\ C))) \rightarrow B$

$(CDDR\ (QUOTE\ (A\ B\ .\ C))) \rightarrow C$

$(CDDDR\ (QUOTE\ (A\ B\ C))) \rightarrow NIL$

$(CDDDR\ (QUOTE\ (A\ B\ .\ C)))$ не определено

$(CONS\ (QUOTE\ P)\ (CONS\ (QUOTE\ Q)$

$(QUOTE\ R))) \rightarrow (P\ Q\ .\ R)$

1.32. Числа

В языке лисп употребляются числа двух типов: *целые (десятичные)* и *восьмеричные (строки битов)* *). Атом, изображающий целое число, — это последовательность цифр, которой может предшествовать знак числа (+ или —).

Пр и м е р ы:

2	—100	0
194	+01234	—0

Каждый такой атом — это константа, обладающая соответствующим числовым значением. Знак + и нули в начале записи числа не влияют на его значение. Поэтому оба следующие выражения: $(EQ\ 1234\ +\ 01234)$ и $(EQ\ —\ 0\ 0)$ имеют значение *T*. Выражение

$(EQ\ (QUOTE\ 1234)\ (QUOTE\ 01234))$

*) Существуют варианты языка, в которых допускаются также приближенные десятичные числа, запись которых содержит десятичную точку и (или) десятичный порядок, и варианты, в которых между целыми и восьмеричными числами различия не проводится (строка битов читается как двоичная запись целого числа).

также даст значение T , но обращения к $QUOTE$ здесь излишни, поскольку запись числа — это такая константа, с которой требуемое значение связывается автоматически.

Восьмеричные числа изображаются атомом, составленным из одной или нескольких восьмеричных цифр, за которыми следует буква Q , например,

$$\begin{array}{cc} 777Q & 4000000000000000Q \\ 00247000Q & 0Q \end{array}$$

Нули в начале записи восьмеричного числа могут быть опущены (за исключением случая, представленного в последнем примере). Значением восьмеричного числа является последовательность двоичных цифр (битов) — представление этого числа в двоичной системе, дополненное спереди нулями до получения полного машинного слова. Нули в конце записи восьмеричного числа могут быть заменены *масштабным множителем* — десятичным числом, показывающим, сколько восьмеричных нулей следует приписать к этому числу справа перед его переводом в двоичное представление. Масштабный множитель помещается в записи восьмеричного числа вслед за буквой Q . Таким образом,

$$\begin{aligned} (EQ\ 00247000Q\ 247Q3) &\rightarrow T \\ (EQ\ 0004Q15\ 4000000000000000Q) &\rightarrow T \end{aligned}$$

Если после перевода восьмеричного числа в двоичное представление в нем окажется больше двоичных разрядов, чем вмещает машинное слово, то лишние разряды в начале этого двоичного представления отбрасываются. Так, например, если машинное слово содержит 48 двоичных разрядов, то при вводе чисел $0004Q15$ и $1234Q15$ будет получено одно и то же значение.

1.33. Предикаты, классифицирующие атомы

В связи с наличием в языке нескольких различных классов атомов необходимы предикаты, различающие принадлежность атома к тому или иному классу. Перечислим эти предикаты.

Предикат $NUMBERP$. Проверить, является ли значение выражения x числом (десятичным или восьмеричным), можно, обратившись к предикату $NUMBERP$ с помощью выражения

$$(NUMBERP\ x)$$

Примеры:

$(NUMBERP\ 5) \rightarrow T$
 $(NUMBERP\ NIL) \rightarrow NIL$
 $(NUMBERP\ 1Q) \rightarrow T$

Предикат *FIXP*. Для обращения к нему следует написать выражение

$(FIXP\ x)$

Значение предиката равно T , если значение x — целое десятичное число, и NIL в противном случае.

$(FIXP\ 5) \rightarrow T$
 $(FIXP\ T) \rightarrow NIL$
 $(FIXP\ 1Q) \rightarrow NIL$

Предикат *BITSP*. К этому предикату можно обратиться следующим образом:

$(BITSP\ x)$

Значение предиката равно T , если значение x — восьмеричное число, и NIL в противном случае.

$(BITSP\ 5) \rightarrow NIL$
 $(BITSP\ T) \rightarrow NIL$
 $(BITSP\ 1Q) \rightarrow T$

1.34. Арифметические функции и предикаты

В языке существует ряд функций, называемых *арифметическими*. Аргументы этих функций должны обладать целыми десятичными (не восьмеричными) числовыми значениями. Эти функции вырабатывают значения такого же типа. Перечислим эти функции.

Функция *PLUS*. Обращение к ней имеет вид

$(PLUS\ x_1\ x_2\ \dots\ x_k)$

Число аргументов произвольно. Значение функции равно сумме значений аргументов $(x_1 + x_2 + \dots + x_k)$, например,

$(PLUS\ 1\ 5\ 9) \rightarrow 15$
 $(PLUS\ -1\ (PLUS\ 5\ 9)) \rightarrow 13$

Функция *TIMES*. Функция допускает обращение вида

$(TIMES\ x_1\ x_2\ \dots\ x_k)$

Число аргументов произвольно. Значение функции равно произведению значений аргументов $(x_1 \times x_2 \times \dots \times x_k)$.

$$(TIMES\ 2\ 3\ 1) \rightarrow 6$$

$$(TIMES\ -2\ (TIMES\ 3\ 1)) \rightarrow -6$$

Функция DIFFERENCE. Обращение к ней:

$$(DIFFERENCE\ x\ y)$$

Значение функции равно разности значений аргументов, т. е. $x - y$.

$$(DIFFERENCE\ 7\ 5) \rightarrow 2$$

$$(DIFFERENCE\ 7\ -5) \rightarrow 12$$

Функция MINUS (одноместная). К ней следует обращаться так:

$$(MINUS\ x)$$

Значение функции равно значению аргумента с противоположным знаком $(-x)$.

$$(MINUS\ 7) \rightarrow -7$$

Функция QUOTIENT. Она требует обращения вида

$$(QUOTIENT\ x\ y)$$

Значение функции равно целой части от деления x на y .

$$(QUOTIENT\ 15\ 5) \rightarrow 3$$

$$(QUOTIENT\ 15\ 6) \rightarrow 2$$

Функция REMAINDER. Обращение к ней таково:

$$(REMAINDER\ x\ y)$$

Значение функции равно остатку от деления x на y .

$$(REMAINDER\ 15\ 5) \rightarrow 0$$

$$(REMAINDER\ 15\ 6) \rightarrow 3$$

Функция EXPT. Она допускает такое обращение:

$$(EXPT\ x\ y)$$

Функция определена лишь при неотрицательном значении y . Значение функции равно x в степени y (x^y).

$$(EXPT\ 4\ 3) \rightarrow 64$$

$$(EXPT\ -1\ 3) \rightarrow -1$$

Функция ADD1. Обращение к этой функции:

$(ADD1\ x)$

Значение функции на 1 больше значения x , т. е. равно $x + 1$.

$(ADD1\ 0) \rightarrow 1$

Функция SUB1. К ней можно обратиться так:

$(SUB1\ x)$

Значение функции на 1 меньше значения x , т. е. равно $x - 1$.

$(SUB1\ 0) \rightarrow -1$

Функция MAX. К этой функции следует обращаться с помощью выражения

$(MAX\ x_1\ x_2\ \dots\ x_k)$

Число аргументов произвольно. Значение функции равно наибольшему из значений аргументов.

$(MAX\ 2\ 3\ 5) \rightarrow 5$

Функция MIN. Функция требует обращения вида

$(MIN\ x_1\ x_2\ \dots\ x_k)$

Число аргументов произвольно. Значение функции равно наименьшему из значений аргументов.

$(MIN\ 2\ -3\ 5) \rightarrow -3$

$(MAX\ (MIN\ 25\ 2)\ 3) \rightarrow 3$

$(MAX\ (MIN\ 2\ 5)\ 1) \rightarrow 2$

Арифметическим предикатом называется предикат, аргументы которого должны принимать десятичные числовые значения. В языке имеются следующие арифметические предикаты.

Предикат LESSP. Он требует обращения вида

$(LESSP\ x\ y)$

Значение предиката равно T , если значение x меньше значения y , и NIL в противном случае.

$(LESSP\ 4\ 5) \rightarrow T$

$(LESSP\ 5\ 4) \rightarrow NIL$

$(LESSP\ 5\ 5) \rightarrow NIL$

Предикат GREATERP. Обращение к нему осуществляется так:

$(GREATERP\ x\ y)$

Значение предиката равно T , если значение x больше значения y , и NIL в противном случае.

$(GREATERP\ 4\ 5) \rightarrow NIL$

$(GREATERP\ 5\ 4) \rightarrow T$

Предикат GREQP. Вид обращения к нему:

$(GREQP\ x\ y)$

Значение предиката равно T , если значение x больше значения y или равно ему, и NIL в противном случае.

$(GREQP\ 4\ 5) \rightarrow NIL$

$(GREQP\ 5\ 5) \rightarrow T$

Предикат MINUSP. Обращение к этому предикату имеет вид

$(MINUSP\ x)$

Значение предиката равно T , если значение x отрицательно.

$(MINUSP\ -2) \rightarrow T$

$(MINUSP\ -0) \rightarrow NIL$

Предикат ZEROP. Для обращения к нему служит выражение

$(ZEROP\ x)$

Значение предиката равно T , если значение x равно нулю.

$(ZEROP\ 1) \rightarrow NIL$

$(ZEROP\ 0) \rightarrow T$

Предикат ONEP. Этот предикат требует обращения вида

$(ONEP\ x)$

Значение предиката равно T , если значение x равно 1.

$(COND ((LESSP (ADD 1 -1) (SUB 1 -1)) NIL)$
 $((GREATERP (MIN 1 -1) (MAX 1 -1)) NIL)$
 $((MINUSP (DIFFERENCE (TIMES 1 2 3)$
 $(PLUS 1 2 3))) NIL)$
 $((ZEROP (MINUS (QUOTIENT 1 1))) NIL)$
 $(T (ONEP (REMAINDER (EXPT 5 4)$
 $(COND ((NUMBERP 0Q)$
 $4) (T 5))))) \rightarrow T$

Предикат EQ, хотя и не является арифметическим, позволяет сравнивать между собой числовые значения. Примеры были

приведены в разд. 1.32. Каждое числовое значение считается атомом, так что

$$(ATOM (PLUS 1 1 1 1 1)) \rightarrow T$$

1.35. Операции над строками битов

Хотя восьмеричные числа и относятся к категории чисел, они не могут участвовать ни в каких арифметических операциях *). Их значения рассматриваются как последовательности двоичных разрядов (битов). Такие последовательности могут быть значениями аргументов, а также являются результатами следующих функций.

Функция LOGOR. Обращение к ней имеет вид

$$(LOGOR x_1 x_2 \dots x_k)$$

Число аргументов произвольно. Значение i -го разряда результата равно 1, если i -й разряд значения хотя бы одного из аргументов равен 1, и 0 в противном случае.

$$(LOGOR 54321Q 70707Q) \rightarrow 74727Q$$

$$(LOGOR 123Q 567Q 77Q1) \rightarrow 777Q$$

Функция LOGAND. Функция требует обращения следующего вида:

$$(LOGAND x_1 x_2 \dots x_k)$$

Число аргументов произвольно. Значение i -го разряда результата равно 0, если i -й разряд значения хотя бы одного из аргументов равен 0, и 1 в противном случае.

$$(LOGAND 12Q2 34Q) \rightarrow 0Q$$

$$(LOGAND 4Q 5Q 6Q) \rightarrow 4Q$$

Функция LOGXOR. Обращение к этой функции таково:

$$(LOGXOR x_1 x_2 \dots x_k)$$

Число аргументов произвольно. Значение i -го разряда результата равно сумме по модулю 2 i -х разрядов значений аргументов, т. е. 0, если общее число единиц в i -м разряде значений аргументов четно, и 1, если это число нечетно.

$$(LOGXOR 3Q 6Q) \rightarrow 5Q$$

$$(LOGXOR 1Q 2Q 3Q 4Q 5Q 6Q 7Q) \rightarrow 0Q$$

*) Это соглашение действует не во всех реализациях языка (см. сноску на стр. 44).

Функция LEFTSHIFT. Для обращения к функции служит выражение

$(LEFTSHIFT\ x\ y)$

Первый аргумент должен быть восьмеричным, второй — десятичным числом. Значение функции равно значению x , сдвинутому влево на n двоичных разрядов, где n — значение y , если это значение положительно. Если же n отрицательно, то сдвиг происходит на $|n|$ разрядов вправо. Разряды значения аргумента x , вышедшие при сдвиге за пределы возможного значения результата, теряются. Примеры:

$(LEFTSHIFT\ 2525Q\ 1) \rightarrow 5252Q$

$(LEFTSHIFT\ 2525Q\ -4) \rightarrow 125Q$

1.36. Функционалы

Во многих случаях полезны функции, содержащие параметр (связанную переменную), которому при обращении ставится в соответствие некоторое определяющее выражение и который при вычислении этой функции используется как наименование функции. Такие функции называются *функционалами*, а параметр с указанным свойством — *функциональной переменной*. Разница между обычной и функциональной связанной переменной аналогична разнице между константой и наименованием функции. Среди связанных переменных функционала может быть несколько функциональных переменных. Аргументы обращения, соответствующие функциональным переменным, называются *функциональными аргументами*. Функциональными аргументами могут быть не только определяющие выражения, но и наименования функций, а если функциональный аргумент задается внутри (в теле) функционала, — то и функциональные переменные этого внешнего функционала.

Записывается функциональный аргумент в одном из следующих видов:

$(FUNCTION\ de)$

$(FUNCTION\ fn)$

fv

где de — определяющее выражение функции, fn — наименование функции класса *EXPR*, *SUBR* или *FSUBR*, fv — функциональная переменная внешнего функционала. Соответствующая функциональная переменная fv_1 того функционала, к которому мы обращаемся, связывается со значением de или fn (в этом отношении функция *FUNCTION* идентична функции *QUOTE*) или со значением, которым обладает переменная fv . При вычислении тела этого функционала

вместо fv_1 используется связанное с ней значение. Но этого мало. Внутри определяющего выражения de или определяющего выражения de_1 функции fn , если это — функция класса $EXPR$, могут встречаться не только связанные переменные этого выражения, но и другие переменные, которые называются *свободными*. Значения свободных переменных в момент задания функционального аргумента и в момент его использования могут отличаться друг от друга. Считается, что при вычислении тела определяющего выражения de или de_1 должны использоваться те значения свободных переменных, которые эти переменные имели во время задания функционального аргумента, а не те, которые они могли получить позже. Это — существенное исключение из общего правила (см. разд. 1.25), по которому для переменных всегда берется их активное значение. Задавая функциональный аргумент в виде $(FUNCTION\ de)$ или $(FUNCTION\ fn)$, мы и обеспечиваем соблюдение этого особого правила нахождения значений свободных переменных в телах функциональных аргументов. Если же в теле определяющего выражения ни прямо ни косвенно (через другие функции класса $EXPR$) не используются свободные переменные, то вместо $FUNCTION$ можно прибегнуть к $QUOTE$. Иначе говоря, допустимы еще две формы задания функциональных аргументов: $(QUOTE\ de)$ и $(QUOTE\ fn)$. Последнюю форму заведомо можно применять, если fn — наименование встроенной функции, так как такие функции не связаны ни с какими определяющими выражениями, и проблемы свободных переменных для них нет.

П р и м е р. Если функциональный аргумент задан выражением

$$(FUNCTION\ (LAMBDA\ (X)\ (COND\ ((NULL\ X)\ Y)\ (T\ X))))$$

то переменная X связана в теле этого аргумента, а переменная Y , которая не входит в список (X) — свободна. Пусть в функционале, к которому мы обращаемся, этому аргументу соответствует функциональная переменная F , а кроме того у него есть своя переменная Y . Пусть в теле функционала содержится обращение к функции

$$(F\ Y)$$

Вычисляя это выражение, сначала следует заменить его на

$$((LAMBDA\ (X)\ (COND\ ((NULL\ X)\ Y)\ (T\ X)))\ Y)$$

а затем перейти к вычислению условного выражения

$$(COND\ ((NULL\ X)\ Y)\ (T\ X))$$

предварительно связав переменную X с текущим (активным) значением переменной Y . Но если потребуется найти значение Y , вычисляя

это условное выражение, то следует взять более раннее значение этой переменной — то, которым она обладала во время задания функционального аргумента.

В качестве следующего примера рассмотрим довольно употребительную функцию *MAPLIST*.

Функция *MAPLIST*. У этой функции два аргумента. Первый должен иметь своим значением список *x*, второй — определяющее выражение *f* некоторой функции с одним аргументом-списком. Значением функции *MAPLIST* является список, состоящий из результатов применения функции *f* ко всему списку *x*, к остатку после отбрасывания первого элемента из этого списка, к остатку после отбрасывания двух первых элементов и т. д.

```
(SEXPR MAPLIST (LAMBDA (X F) (COND
  ((NULL X) NIL)
  (T (CONS (F X) (MAPLIST (CDR X) F))) )))
```

Здесь во внутреннем (рекурсивном) обращении к *MAPLIST* функциональный элемент задан просто в виде атома *F* — функциональной переменной, уже получившей свое значение при предыдущем обращении. Это же значение используется на следующем уровне рекурсии. Исходное обращение могло быть, например, таким:

```
(MAPLIST (QUOTE (A B (C D)))
  (QUOTE (LAMBDA (Y) (CONS
    (CAR Y) (QUOTE W) ))) )
```

Его значением является выражение $((A.W) (B.W) ((C D).W))$. Здесь функциональный аргумент задан с помощью *QUOTE*, так как он не содержит свободных переменных.

Но вот пример, где при обращении к *MAPLIST* задание функционального аргумента с помощью *FUNCTION* обязательно. Пусть с помощью *MAPLIST* мы хотим определить аналогичную функцию *MAPCAR*, которая поочередно применяет свой функциональный аргумент ко всем элементам (а не к остаткам, как *MAPLIST*) заданного списка и строит список полученных значений. Такое определение функции *MAPCAR* должно иметь вид:

```
(SEXPR MAPCAR (LAMBDA (X F)
  (MAPLIST X (FUNCTION (LAMBDA (X)
    (F (CAR X)) ))) ))
```

(1)

Здесь определяющее выражение

```
(LAMBDA (X) (F (CAR X)))
```

(2)

$$\begin{aligned} & (MAPCAR (QUOTE (A B (C D))) \\ & \quad (QUOTE (LAMBDA (X) \\ & \quad \quad (CONS X (QUOTE W)))))) \end{aligned} \quad (3)$$
$$(LAMBDA (X) (CONS X (QUOTE W))) \quad (4)$$

Функционалом не может быть функция класса *FEXPR*.

1.37. Функция *SELECTQ*

$$(SELECTQ\ e\ (a_1\ e_1 \dots a_n\ e_n)\ e_0)$$
$$\begin{array}{c} (COND ((EQ e (QUOTE a_1)) e_1) \\ \quad \\ \quad ((EQ e (QUOTE a_n)) e_n) \\ (T e_0)) \end{array}$$

Функция *SELECTQ* очень удобна при обработке формул, записанных в виде списка, первый элемент которого—главный знак операции в формуле. Пример такого применения функции *SELECTQ* можно найти в Приложении.

Нетрудно дать описание этой функции

```
(SFEXPR SELECTQ (LAMBDA (E L E0) (COND
  ((NULL L) (EVAL E0))
  ((EQ (EVAL E) (CAR L)) (EVAL (CADR L)))
  (T (SELECT E (CDDR L) E0)) )))
```

Существует более общий вариант функции *SELECTQ*, в котором каждое a_i может быть либо атомом, либо списком атомов. В первом случае проверяется, что значение выражения e совпадает с a_i , во втором—что оно содержится в списке a_i . Чтобы получить описание этого варианта, надо в предыдущем описании заменить выражение $(EQ (EVAL E) (CAR L))$ на

```
(COND ((ATOM (CAR L)) (EQ (EVAL E) (CAR L)))
  (T (MEMB (EVAL E) (CAR L))))
```

1.38. Пример программы

Составим программу, отыскивающую в лабиринте путь между двумя точками: входом и центром. Нужно выбрать способ изображения лабиринта лисповским выражением. Выделим в лабиринте характерные точки: разветвления и тупики. Присоединим к ним вход и центр и будем их коротко называть точками лабиринта. Лабиринт состоит из коридоров, ведущих от точки к точке. Длина и извилистость коридоров не интересуют нас в этой задаче. Коридор, не содержащий других точек лабиринта, кроме своего начала и конца, назовем ходом, соединяющим эти две точки. Лабиринт полностью определяется для нас перечислением всех его ходов. Ходы будем изображать парами вида $(x \cdot y)$, где x и y — конечные точки хода. Весь лабиринт изобразим в виде списка всех его ходов

$$((x_1 \cdot y_1) (x_2 \cdot y_2) \dots (x_n \cdot y_n))$$

Существует множество других способов задания лабиринта, например, в виде списка, состоящего из списков $(x_i y_{i1} \dots y_{im_i})$, где x_i — точка лабиринта, а y_{i1}, \dots, y_{im_i} — точки, с которыми точка x_i соединена ходами. Но мы остановимся на первоначально выбранном способе. Вообще же в практических задачах надо

стремиться так представить исходную и промежуточную информацию, чтобы алгоритм решения был возможно проще.

Путь в лабиринте между точками a и b — это последовательность ходов

$$(a \cdot u_1), (u_1 \cdot u_2), \dots, (u_k \cdot b) \quad (1)$$

Естественно считать, что все точки a, u_1, \dots, u_k, b различны (иначе путь можно сократить). Коротко путь можно изобразить списком

$$(a \ u_1 \ \dots \ u_k \ b) \quad (2)$$

или

$$(b \ u_k \ \dots \ u_1 \ a) \quad (3)$$

Оба варианта будем считать равноправными.

Один из возможных методов решения задачи основывается на следующих определениях. Назовем нулевым ярусом лабиринта его центр (или, при желании, вход). К i -му ярусу для $i \geq 1$ отнесем все точки, не принадлежащие ни одному из ярусов с номерами $0, 1, \dots, i-1$, которые соединены ходами хотя бы с одной из точек $(i-1)$ -го яруса.

Будем последовательно строить ярусы лабиринта, запоминая ходы, соединяющие точки соседних ярусов. Если в очередном ярусе встретится вход (центр) лабиринта, то накопленная информация позволит легко восстановить путь от входа к центру. Если же очередной ярус пуст, то пути от входа к центру не существует.

Функцию *PATH*, разыскивающую путь в лабиринте, опишем с помощью аппарата *PROG*. Аргумент L этой функции — это список ходов лабиринта, аргумент A — вход, B — центр лабиринта. Программные переменные имеют следующий смысл. Очередной ярус накапливается в списке J . В это время I содержит список точек предыдущего яруса. В момент, когда обнаруживается новая точка U яруса J , соединенная ходом с точкой V яруса I , пара $(U \cdot V)$ присоединяется к списку S . Все пары, у которых оба элемента принадлежат уже построенным ярусам, не представляют дальнейшего интереса и исключаются из списка L . Остальные пары переносятся из списка L в другой список M . Когда список L опустеет, ярус J , если он не пуст, становится ярусом I , а список M — списком L , и начинает формироваться следующий ярус. Переменные X, Y и Z — вспомогательные.

Если искомый путь существует, то рано или поздно очередная точка яруса J совпадет с A . С этого момента начинается розыск в списке S цепочки пар вида (1) и с ее помощью строится список (3), дающий решение задачи. Если же полностью сформированный ярус J пуст, то в качестве ответа выдается *NIL*. Первый оператор функции

PATH проверяет вырожденный случай, когда вход в лабиринт совпадает с центром.

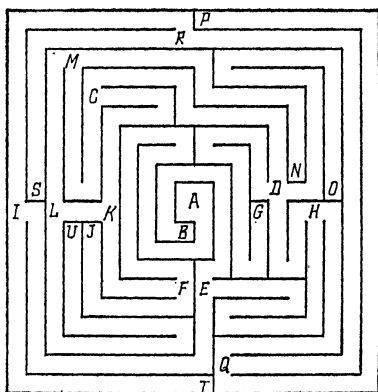


Рис. 1.37.1.

Функция *PATH* использует функцию *MEMB*, описание которой взято из разд. 1.24. Вся программа вместе с обращением к функции *PATH* для лабиринта, изображенного на рис. 1.37.1, такова:

```
(SEXPR MEMB (LAMBDA (X Y) (PROG NIL
  A (COND ((NULL Y) (RETURN NIL))
           ((EQ X (CAR Y)) (RETURN T)))
  (SETQ Y (CDR Y)) (GO A) )))
(SEXPR PATH (LAMBDA (L A B)
  (PROG (I J S M X Y Z)
    (SETQ I (CONS B NIL))
    (COND ((EQ A B) (RETURN I)))
    A (COND ((NULL L) (GO D)))
      (SETQ Z (CAR L)) (SETQ L (CDR L))
      (SETQ X (CAR Z)) (SETQ Y (CDR Z))
      (COND ((MEMB Y I) (GO C))
             ((MEMB X I) (GO B)))
      (SETQ M (CONS Z M)) (GO A)
    B (SETQ Z (CONS Y X))
      (SETQ Y X) (SETQ X (CAR Z))
    C (COND ((MEMB X J) (GO A)))
      (SETQ S (CONS Z S))
      (COND ((EQ X A) (GO E)))
      (SETQ J (CONS X J)) (GO A)
```

```

D (COND ((NULL J) (RETUR N NIL)))
  (SETQ L M) (SETQ I J)
  (SETQ M NIL) (SETQ J NIL) (GO A)
E (SETQ Z NIL)
F (COND ((NULL S) (RETURN (CONS X Z))))
  (SETQ Y (CAR S)) (SETQ S (CDR S))
  (COND ((EQ X (CAR Y)) (GO G))) (GO F)
G (SETQ Z (CONS X Z))
  (SETQ X (CDR Y)) (GO F) )))
(PATH (QUOTE ((O . D) (P . Q) (N . M)
  (G . D) (H . D) (H . E) (H . Q)
  (D . C) (C . M) (C . K) (A . F)
  (B . E) (Q . R) (M . L) (E . I) (I . R)
  (I . T) (R . S) (F . K) (F . J) (L . K) (L . U)))
  (QUOTE T) (QUOTE A) )

```

В результате выполнения этой программы отпечатается она сама и за каждым из трех входящих в нее выражений их значения, а именно:

```

MEMB
PATH
(A F K C D H E I T)

```

Другой вариант программы не использует механизма *PROG*. Его основу составляет функция *STEP*, осуществляющая анализ одной пары списка *L* при построении очередного яруса *J*. Она обращается к функциям *MEMB* и *P*, которые завершают построение пути, пользуясь информацией, накопленной в списке *S*. Переменные *L*, *A*, *I*, *S*, *M*, *J* имеют тот же смысл, что и в *PATH*. Подготовительные операции для начала поиска пути выполняет функция *PATH1*, имеющая те же аргументы, что и *PATH*.

```

(SEXPR MEMB (LAMBDA (X Y) (COND
  ((NULL Y) NIL)
  ((EQ X (CAR Y)) T)
  (T (MEMB X (CDR Y))) )))
(SEXPR P (LAMBDA (X S) (COND
  ((NULL S) (CONS X NIL))
  ((EQ X (CAAR S)) (CONS X
  (P (CDAR S) (CDR S)) ))
  (T (PX (CDR S))) )))
(SEXPR STEP (LAMBDA) (L A I S M J) (COND
  ((MEMB A J) (P A S))
  ((NULL L) (COND ((NULL J) NIL)

```

```

      (T (STEP M A J S NIL NIL)) ))
((MEMB (CDAR L) I) (COND
  ((MEMB (CAAR L) I)
    (STEP (CDR L) A I S M J))
  ((MEMB (CAAR L) J)
    (STEP (CDR L) A I S M J))
  (T (STEP (CDR L) A I
    (CONS (CAR L) S) M
    (CONS (CAAR L) J) )) ))
((MEMB (CAAR L) I) (COND
  ((MEMB (CDAR L) J)
    (STEP (CDR L) A I S M J))
  (T (STEP (CDR L) A I (CONS
    (CONS (CDAR L) (CAAR L)) S)
    M (CONS (CDAR L) J) )) ))
  (T (STEP (CDR L) A I S
    (CONS (CAR L) M) J)) )))
(SEXPR PATH1 (LAMBDA (L A B) (COND
  ((EQ A B) (CONS A NIL))
  (T (STEP L A (CONS B NIL)
    NIL NIL NIL)) )))
(PATH1 (QUOTE ((O . D) (P . Q) (N . M)
  (G . D) (H . D) (H . E) (H . Q) (D . C) (C . M) (C . K)
  (A . F) (B . E) (Q . R) (M . L) (E . I) (I . R)
  (J . T) (R . S) (E . K) (E . J) (L . K) (L . U)))
  (QUOTE T) (QUOTE A))

```

Программа отпечатает следующие значения входящих в нее выражений:

```

MEMB
P
STEP
PATH1
(T I E H D C K F A)

```

В данном случае механизм рекурсивных описаний не дает никаких видимых преимуществ перед механизмом *PROG*.

Алгоритм, реализуемый функциями *PATH* или *PATH1*, находит кратчайший путь от входа к центру лабиринта, если под длиной пути понимать число ходов, из которых этот путь состоит. Однако этот метод мало соответствует ситуации, возникающей при поиске пути в лабиринте, план которого неизвестен. В этих условиях, попав в очередную точку лабиринта, не центр, мы должны выбрать один

из ходов, выходящих из этой точки, по которому мы еще не ходили, и перейти по нему в следующую точку. Если же таких ходов нет, то надо вернуться в предыдущую точку. Если же строящийся путь привел нас в точку, встречавшуюся ранее, то надо также возвращаться. Действительно, поскольку эта точка лежит на пройденной части пуги, то в пути образовалась петля. Путь к центру может вести от одной из промежуточных точек этой петли, поэтому и необходимо вернуться, чтобы обследовать все возможные ответвления. Если все эти ответвления не приводят к центру, то мы все равно вернемся к точке, в которой замкнулась петля.

На основе этих соображений составлена следующая программа. Функции *MEMB* и *JOIN* — вспомогательные. Вторая из них возвращает в список *L* те его элементы, которые были перенесены в список *M*. Основная часть алгоритма — функции *UP* и *DOWN*. Первая разбирается в ситуации, встречающейся в точке *X*, куда мы только что пришли. Вторая осуществляет поиск в списке *L* ходов, ведущих из точки *X*. Ходы, не относящиеся к этой точке, накапливаются в списке *M*. Когда список ходов *L* исчерпан, функция *DOWN* возвращается к предыдущей точке. В списке *R* хранится путь, приведший в точку, предшествующую точке *X*. Ходы, по которым мы прошли хотя бы один раз, исключаются из списка ходов. Благодаря этому, попасть в заведомо бесперспективную точку невозможно.

В приводимой ниже программе выражение, вводящее функцию *MEMB*, и обращение к функции *PATH2* опущены. Они ничем не отличаются от предыдущих.

```
(SEXPR JOIN (LAMBDA (M L) (COND
  ((NULL M) L)
  (T (JOIN (CDR M)
    (CONS (CAR M) L) )) )))
(SEXPR UP (LAMBDA (L X B M R) (COND
  ((EQ X B) (CONS B R))
  ((MEMB X R) (DOWN L
    (CAR R) B M (CDR R)))
  (T (DOWN (JOIN M L) X B NIL R)) )))
(SEXPR DOWN (LAMBDA (L X B M R) (COND
  ((NULL L) (COND ((NULL R) NIL)
    (T (DOWN M (CAR R)
      B NIL (CDR R))) ))
  ((EQ X (CAAR L)) (UP (CDR L)
    (CDAR L) B M (CONS X R)))
  ((EQ X (CDAR L)) (UP (CDR L)
    (CAAR L) B M (CONS X R)))
```

```

(T (DOWN (CDR L) X B
      (CONS (CAR L) M) R)) )))
(SEXPR PATH2 (LAMBDA (L A B) (UP L A B NIL NIL)))

```

Программа находит следующий путь в лабиринте:

```
(A F K C D H E I T)
```

И наконец, вариант этого же метода, описанный с помощью аппарата *PROG*. Описание функции *MEMB* и обращение к *PATH3* также опущены. Результат работы программы прежний.

```

(SEXPR PATH3 (LAMBDA (L A B) (PROG (M R X)
  UP (COND ((EQ A B) (RETURN (CONS B R)))
    ((MEMB A R) (GO D1)))
  JOIN (COND ((NULL M) (GO DOWN)))
    (SETQ L (CONS (CAR M) L))
    (SETQ M (CDR M)) (GO JOIN)
  D1 (SETQ A (CAR R)) (SETQ R (CDR R))
  DOWN (COND ((NULL L) (GO D2)))
    (SETQ X (CAR L)) (SETQ L (CDR L))
    (COND ((EQ (CAR X) A) (GO U1))
      ((EQ (CDR X) A) (GO U2)))
    (SETQ M (CONS X M) (GO DOWN)
  U1 (SETQ X (CDR X)) (GO U3)
  U2 (SETQ X (CAR X))
  U3 (SETQ R (CONS A R)) (SETQ A X) (GO UP)
  D2 (COND ((NULL R) (RETURN NIL)))
    (SETQ L M) (SETQ M NIL) (GO D1) )))

```

ГЛАВА 2

РЕАЛИЗАЦИЯ ЯЗЫКА ЛИСП

В этой главе будут описаны принципы устройства программ для ЭВМ, способных реализовать язык, т. е. выполнять программы. Основное внимание будет уделено реализации лиспа на машине БЭСМ-6, впервые описанной в [7]. Попугно в некоторых случаях будут рассматриваться другие возможные способы реализации.

2.1. Внутреннее представление выражений

В разд. 1.31 было отмечено, что любое допустимое в языке выражение может быть записано с помощью лишь двух понятий: атома и пары. Именно этот способ записи выражений удобно положить в основу их представления в памяти вычислительной машины.

Каждому атому, встречающемуся в программе или возникающему в ходе ее выполнения, ставится в соответствие ячейка, называемая *информационной ячейкой* этого атома. Сам атом замещается во внутреннем представлении выражений адресом его информационной ячейки.

Каждой паре, содержащейся в исходной программе или возникающей при выполнении обращений к функции *CONS*, ставится в соответствие ячейка, в которой выделяются две группы разрядов, называемые *a*-указателем и *d*-указателем. При этом *a*-указатель содержит адрес ячейки, поставленной в соответствие левому элементу пары, а *d*-указатель — адрес ячейки, соответствующей правому элементу пары. Адрес ячейки, поставленной в соответствие паре, представляет эту пару внутри машины.

Таким образом, для каждого выражения, хранящегося в памяти машины, существует представляющий его адрес. В дальнейшем мы будем его называть просто *адресом* выражения. Полное словесное описание совокупности ячеек, поставленной в соответствие даже не очень сложному выражению, было бы слишком громоздким. Поэтому

обычно прибегают к графическому изображению. Оно составляется из фрагментов вида, представленного на рис. 2.1.1. Прямоугольник, разделенный на две клетки, изображает ячейку с выделенными в ней a -указателем (левая клетка) и d -указателем (правая клетка).

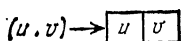


Рис. 2.1.1.

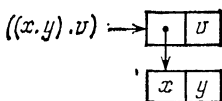


Рис. 2.1.2.

Буква u , вписанная в a -указатель (точнее, в изображающую его клетку), означает, что этот указатель содержит адрес, представляющий выражение u . Аналогичный смысл имеет буква v в правой клетке. Весь фрагмент, изображенный на рис. 2.1.1, соответствует,

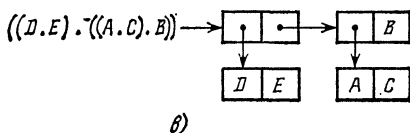
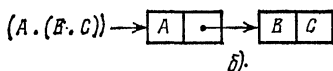
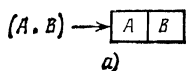


Рис. 2.1.3.

следовательно, паре $(u.v)$, что и показано стрелкой, ведущей от этой пары к ячейке. Если выражение u — не атом, а пара, то и для него можно построить соответствующий фрагмент, как это показано на рис. 2.1.2. Стрелка, ведущая от клетки к ячейке, означает, что клетка содержит адрес ячейки. Информационные ячейки атомов на схемах обычно не изображают, оставляя в клетках, содержащих адреса этих ячеек, обозначения самих атомов. Для пар, входящих в состав других выражений, как, например, для пары $(x.y)$ в выражении $((x.y).v)$, обычно поступают наоборот — не вписывают в клетку обозначение этой пары, а ограничиваются лишь стрелкой, ведущей от этой клетки к ячейке, соответствующей паре. Законченные схемы представления в памяти некоторых конкретных выражений изображены на рис. 2.1.3, а, б, в.

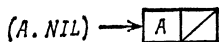


Рис. 2.1.4.

При изображении пар, одним из элементов которых является атом *NIL*, соответствующую клетку обычно перечеркивают наискосок вместо того, чтобы вписывать в нее наименование *NIL* (рис. 2.1.4). Этим подчеркивается особая роль атома *NIL* в языке. Из разд. 1.31 известно, что список $(e_1 e_2 \dots e_n)$ в точечных обозначениях принимает вид

$$(e_1 \cdot (e_2 \cdot \dots \cdot (e_n \cdot NIL) \dots))$$

Поэтому графическое изображение этого списка выглядит так, как это показано на рис. 2.1.5. Разумеется, если элемент e_i — не атом,

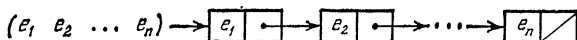


Рис. 2.1.5.

то обозначение e_i в соответствующей клетке на этом рисунке можно заменить стрелкой, ведущей от этой клетки к графическому представлению выражения e_i . Примеры развернутого изображения списков приведены на рис. 2.1.6, а, б.

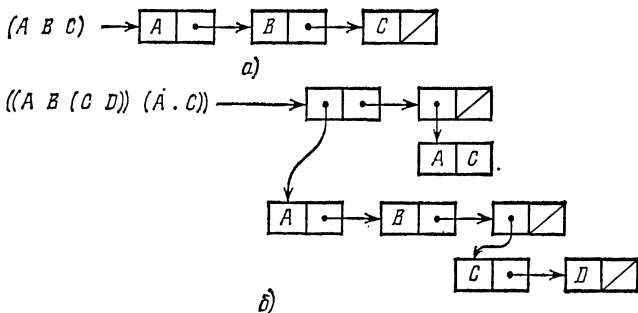


Рис. 2.1.6.

В соответствии со сказанным, в памяти машины должны быть выделены по крайней мере две группы ячеек — группа информационных ячеек атомов и группа ячеек, соответствующих парам. Эти группы (наряду с другими) действительно выделяются — обычно в виде связанных участков памяти. Участок, на котором размещаются информационные ячейки атомов, по традиции называется *списком объектов*, а участок, на котором размещаются пары, — *областью списочной памяти*. Оба названия не совсем удачны. Список объектов не представляет собой списка в том смысле, в каком этот термин используется в языке. Ячейки области списочной памяти могут быть

связаны не только в цепочки, изображающие списки, по схеме, изображенной на рис. 2.1.5, но и в иные структуры. Обычно эти структуры (например, простая пара атомов — рис. 2.1.3, а) могут быть описаны в точечных обозначениях. Но возможны также структуры, не допускающие описания языковыми средствами. Простейший пример изображен на рис. 2.1.7. Это ячейка, *a*-указатель которой содержит адрес информационной ячейки атома *A*, а *d*-указатель — адрес самой ячейки. При попытке отпечатать такую структуру программа печати будет работать безостановочно, печатая последовательность литер

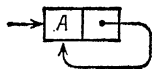
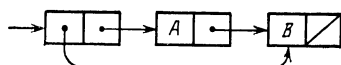


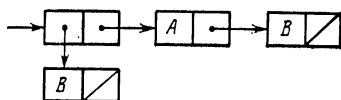
Рис. 2.1.7.

(A A A A A ...

В дальнейшем под списочными структурами мы будем понимать любые структуры, представимые в области списочной памяти, независимо от того, допускают ли они запись на языке. Заметим,



а)



б)

Рис. 2.1.8.

что структуры, в которых цепочки стрелок расходятся, а потом снова сходятся на одной ячейке, но не образуют циклов, допускают лисповскую запись. Так, например, структуре, изображенной на рис. 2.1.8, а, соответствует выражение $((B) A B)$. В этом отношении она полностью эквивалентна структуре, изображенной на рис. 2.1.8, б.

Ни списочные, ни информационные ячейки не обязаны совпадать с машинными ячейками. Может случиться, что списочная ячейка занимает лишь часть машинной ячейки или, наоборот, под списочную ячейку отводятся две машинные ячейки. Для организации уборки мусора (разд. 2.13) весьма желательно, чтобы машинная ячейка (или группа ячеек), соответствующая списочной или информационной ячейке, содержала, кроме *a*- и *d*-указателей, свободные разряды.

2.2. Списки свойств атомов

Мы уже знаем, что атомы могут быть наделены различными свойствами — атом может быть наименованием переменной, константы или функции или же числом. Функции и числа в свою очередь делятся на классы. Все это должно быть отражено в памяти в виде

так называемого *списка свойств* атома. Среди прочих свойств в этом списке должно содержаться и *внешнее представление* атома — последовательность литер, изображающая его в программе.

Доступ к списку свойств всегда открывается через информационную ячейку данного атома. Сам список свойств может быть организован разными способами. Мы опишем два способа: самый простой и самый экономный в смысле расходования памяти.

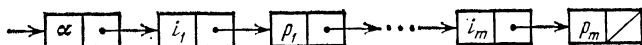


Рис. 2.2.1.

При первом способе весь список свойств располагается в области списочной памяти. Список имеет структуру, изображенную (вместе с информационной ячейкой) на рис. 2.2.1. На этом рисунке α обозначает специальный адрес (может быть, даже не относящийся к допустимому диапазону адресов ячеек памяти), по которому могут быть распознаны информационные ячейки. Сам список свойств состоит из звеньев — по две ячейки в каждом звене. Первая ячейка каждого звена содержит в своем α -указателе так называемый *индикатор* i_k — наименование свойства. Индикаторами могут быть, например, *EXPR*, *FSUBR*, *FIX* и др. Более точно, индикатор — это адрес информационной ячейки атома, используемого в качестве наименования свойства. В следующей ячейке списка свойств α -указатель содержит адрес p_k самого свойства — определяющего выражения функции (для свойства *EXPR*), начала машинной подпрограммы (для *FSUBR*) и т. д.

Одним из индикаторов в списке свойств должен быть индикатор *PNAME* (сокращение от *print name* — печатное наименование). Если бы в языке разрешалось пользоваться лишь короткими наименованиями, содержащими не больше литер, чем их вмещает ячейка, то само свойство могло бы быть представлено ячейкой, в которой упакованы коды литер внешнего наименования атома. На рис. 2.2.2, α изображен соответствующий фрагмент списка свойств атома *NIL* в предположении, что ячейка машины вмещает 6 кодов литер, причем недостающие литеры заменяются кодами, отличающимися от всех кодов литер (на рисунке такая литера изображена перечеркнутым прямоугольником). Если же не ограничивать длину наименований, то ячейки, в которых размещается внешнее наименование, можно связать в список, который и будет представлять свойство с индикатором *PNAME*. Примеры для атомов *NIL* и *FUNCTION* приведены на

рис. 2.2.2, б, в. Разумеется, при этом для коротких атомов уже нельзя пользоваться схемой рис. 2.2.2, а.

Описанная структура списков свойств атомов удобна тем, что число свойств у атома не ограничивается, наряду со стандартными свойствами атому можно приписывать любые дополнительные свойства, которые могут понадобиться в программе. Для исследования и преобразования списков свойств можно пользоваться теми же

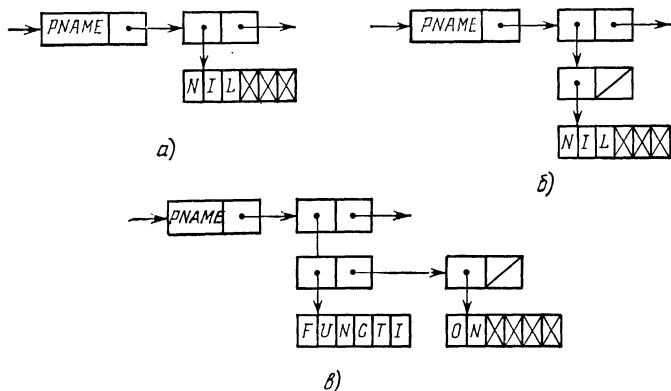


Рис. 2.2.2.

средствами, которые применяются для обработки любых других списков. Соответствующие функции описаны в разд. 3.10. Недостаток этой структуры в том, что списки свойств занимают много места в памяти, а на их просмотр может уходить много времени. По этой причине в реализации языка на машине БЭСМ-6 принята другая структура списка свойств атома. Информационная ячейка атома подобно ячейкам списочной памяти содержит *a*- и *d*-указатели. Но если у ячеек списочной памяти остальные разряды заполнены нулями и почти никакой информации не несут, то у информационных ячеек атома в этих разрядах содержится большая часть сведений об атоме. Один из этих разрядов (43-й) должен содержать 1, являющуюся признаком информационной ячейки. Группа разрядов (47-й — 44-й) содержит код наименования свойства, которым наделен данный атом. Предусмотрены следующие стандартные наименования свойств: *SUBR* — встроенная обычная функция, *FSUBR* — встроенная специальная функция, *EXPR* — обычная функция, определенная в выполняемой программе, *FEXPR* — специальная функция, определенная в программе, *APVAL* — константа (безразлично, встроенная или введенная в программе), *FIX* — целое (десятичное)

число, *BITS* — строка битов (восьмеричное число). Разряды с 39 по 25-й отведены под *a*-указатель, который содержит адрес свойства: адрес начала машинной подпрограммы для свойств *SUBR* и *FSUBR*, адрес определяющего выражения функции в списочной памяти для свойств *EXPR* и *FEXPR*, адрес значения константы для свойства *APVAL*, адрес самой информационной ячейки числа для свойств *FIX* и *BITS*. Как мы увидим впоследствии (см. разд. 2.8), именно такое использование *a*-указателя информационной ячейки числа позволяет записывать числа в лисповских программах, не прибегая к помощи *QUOTE* (см. разд. 1.32).



Рис. 2.2.3.

В *d*-указателе (разряды с 15-го по 1-й) информационной ячейки атома, не являющегося числом, находится начальный адрес группы из одной или нескольких ячеек, в которых упаковано (по 6 литер в ячейке) внешнее наименование атома. Количество ячеек в группе (не более 5) записывается в разрядах с 18-го по 16-й. Для атомов-чисел в *d*-указателе информационной ячейки помещается адрес

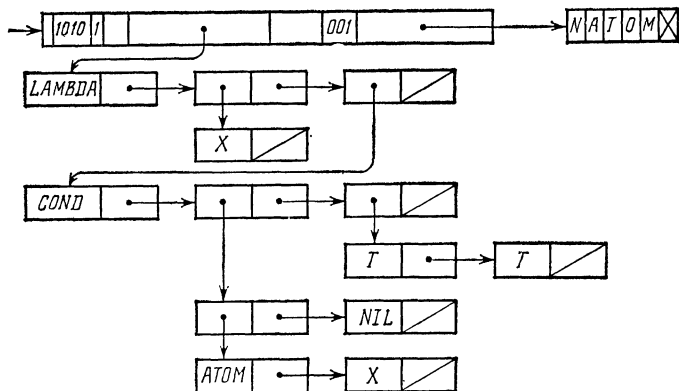


Рис. 2.2.4.

ячейки, в которой находится соответствующее числовое значение в машинном двоичном представлении. Внешнее представление числа, когда в нем возникает потребность (для вывода на печать), получается переводом машинного представления.

Для иллюстрации на рис. 2.2.3 изображен список свойств атома *NIL*, а на рис. 2.2.4 — список свойств атома *NATOM*, в предположении, что он определен в результате выполнения выражения

$$(SEXPR\ NATOM\ (LAMBDA\ (X)\ (COND\ ((ATOM\ X)\ NIL)\ (T\ T))))$$

Двоичное число 0110 в разрядах 47—44 информационной ячейки атома *NIL* на рис. 2.2.3 — это код свойства *APVAL*, а число 1010 на рис. 2.2.4 — код свойства *EXPR*. Число 0000 в тех же разрядах означает, что у атома нет ни одного из стандартных свойств. В этом случае атом может быть снабжен списком нестандартных свойств, адрес которого помещается в *a*-указателе информационной ячейки. Сам этот список имеет структуру, изображенную на рис. 2.2.1 (не считая начальной ячейки).

2.3. Язык для описания реализации

Машинную реализацию языка мы постараемся описать в виде, не зависящем от свойств и характеристик конкретной вычислительной машины, на которой осуществляется эта реализация. Для этого мы воспользуемся алголоподобным языком, в котором используются понятия переменной, оператора присваивания, метки, оператора перехода, условного и составного операторов. Значением каждой переменной или выражения является, как правило, адрес списочной ячейки или информационной ячейки атома. Редкие исключения будут особо оговариваться. Выражения большей частью будут иметь вид указателей функций, т. е. обращений к процедурам, вычисляющим значения функций. Это естественно, так как в лиспе понятие функции является одним из основных. Для простейших функций мы дадим в этом разделе их словесные описания, т. е. введем их на правах стандартных функций. Для многих других обычных и специальных встроенных функций и ряда функций, не имеющих эквивалента в лиспе, будут описаны процедуры вычисления их значений.

Описания этих процедур будут состоять из символа **function** (заменяющего описатель типа значения функции и символ **procedure**, с которых начинаются алгольные описания процедур-функций), идентификатора функции, списка формальных параметров, заключенного в круглые скобки, и оператора — тела функции. Формальные параметры будут обозначаться, как правило, идентификаторами *a1*, *a2*, ... Ни спецификаций, ни списка значений для этих параметров давать не будем. Условимся, что каждому формальному параметру обычной

встроенной функции (класса *SUBR*) присваивается при обращении к функции значение соответствующего фактического параметра. Как было сказано, этим значением является адрес некоторой ячейки. Таким образом, можно считать, что все формальные параметры любой из процедур вызываются значением. Все исключения будут оговариваться словесно. В телах процедур будут использоваться вспомогательные переменные (рабочие ячейки) *r*, *r1*, *r2*, ... Будем пока считать, что как параметры *a1*, *a2*, ... , так и переменные *r*, *r1*, *r2*, ... локализованы в телах процедур и, следовательно, их значения не портятся при обращении к другим процедурам или при рекурсивном обращении к той же самой процедуре. Рекурсивными будут многие из описываемых процедур. Технику перевода таких описаний процедур на машинный язык мы рассмотрим в разд. 2.12.

Процедуры, которые реализуют встроенные функции, будут обозначаться теми же наименованиями, что и эти функции, но записанными строчными буквами. Для каждого атома идентификатор, совпадающий с внешним наименованием атома, но записанный строчными буквами, обозначает адрес информационной ячейки этого атома. Например, *nil* — это адрес информационной ячейки атома *NIL*. Таким образом, в нашем языке появляются идентификаторы-константы, имеющие фиксированное значение.

Для процедур, реализующих специальные встроенные функции (класса *FSUBR* — у них, как было сказано в разд. 1.19, число аргументов может быть произвольным), вводится лишь один формальный параметр *a1*. Ему при обращении к этой процедуре присваивается адрес списка аргументов обращения к данной встроенной функции. Например, обращению к функции *COND* вида

(*COND* ((*NULL X*) *NIL*) (*T T*))

будет соответствовать обращение к процедуре-функции *cond*, при котором параметру *a1* присваивается адрес, представляющий список (((*NULL X*) *NIL*) (*T T*)) (рис. 2.3.1).

Будем придерживаться следующей терминологии. Поскольку мы описываем машинную реализацию языка, термины «процедура-функция» и «подпрограмма», а также «переменная» и «ячейка» будем считать синонимами, отдавая даже предпочтение вторым вариантам.

Опишем работу подпрограммы *car*, реализующей функцию *CAR*. При обращении *car(x)* значение аргумента *x* (т. е. адрес, представляющий значение аргумента соответствующего обращения к *CAR*) помещается в ячейку *a1*. Подпрограмма *car* извлекает содержимое *a*-указателя ячейки, адрес которой находится в ячейке *a1*, и выдает его в качестве результата своей работы. Аналогично, под-

программа *cdr* вырабатывает результат, совпадающий с содержимым *d*-указателя ячейки, адрес которой помещен в *a1*.

Как правило, рабочая часть этих подпрограмм состоит из двух-трех операций: послыки адреса из ячейки *a1* на индекс-регистр,

сдвига (если нужно) содержимого ячейки с этим адресом и логического умножения для выделения содержимого интересующих нас разрядов.

Первая операция не нужна, если в качестве ячейки *a1* уже используется индекс-регистр. Сдвиг

обычно бывает нужен лишь в одной из этих подпрограмм.

Статистика показывает, что в текстах реальных программ обращения к *CAR* встречаются в среднем на 10% (иногда до 40%) чаще, чем обращения к *CDR*. Причину такого предпочтения можно объяснить тем, что функция *f*, работающая со списками, часто имеет следующую структуру определяющего выражения:

$$\begin{aligned} & (LAMBDA (L) (COND ((NULL L) NIL) \\ & ((p (CAR L)) (g (CAR L))) \\ & (T (f (CDR L))))) \end{aligned}$$

Поэтому целесообразнее выделить под *a*-указатель те разряды ячейки, в которые желательно поместить результат операции. Тогда сдвиг понадобится лишь в подпрограмме *cdr*. Результат же удобнее всего размещать в тех разрядах ячейки, откуда его легче послать на индекс-регистр (для машины БЭСМ-6 — в младших разрядах, для машин типа М-20 — в разрядах второго адреса и т. п.). Поэтому на машине БЭСМ-6 следовало бы отвести под *a*-указатель разряды 15 — 1, а под *d*-указатель — разряды 39 — 25, хотя на самом деле сделано наоборот.

Подпрограмма, реализующая предикат *ATOM*, должна проверить, является ли значение переменной *a1* адресом информационной ячейки некоторого атома. Если да, то процедура должна выработать значение *i* (т. е. адрес информационной ячейки атома *T*), если нет — то *nil*. Если информационные ячейки атомов устроены одним из способов, описанных в разд. 2.2, то проверка производится либо по содержимому *a*-указателя (для информационной ячейки оно должно быть равно α), либо по значению выделенного (для БЭСМ-6 — 43-го) разряда ячейки с адресом, взятым из *a1*.

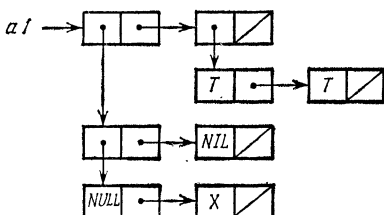


Рис. 2.3.1.

Первый вариант подпрограммы *atom* можно описать на принятом нами языке в виде процедуры

```
function atom (a1);  
    if car(a1) = alpha then atom := t  
    else atom := nil
```

Здесь *alpha* — константа, значением которой является характерный адрес α .

Сходным образом могут быть описаны подпрограммы *eq* и *null*:

```
function eq(a1, a2);  
    if a1 = a2 then eq := t  
    else eq := nil  
  
function null (a1);  
    if a1 = nil then null := t  
    else null := nil
```

Формально при таком описании процедур *atom*, *eq* и *null* мы уже не можем пользоваться выражениями *atom*(*x*), *eq*(*x*, *y*) или *null*(*x*) в качестве логических выражений (следует писать *atom*(*x*) = *t* и т. п.). Но мы будем все же прибегать к записи вида **if** *atom*(*x*) **then**, имея в виду, что при этом проверяется то же условие, что и при вычислении значения *atom*(*x*).

Подпрограмма *cons* выделяет ячейку в свободной части области списочной памяти, помещает в *a*- и *d*-указатели этой ячейки адреса, взятые из ячеек *a1* и *a2*, и выдает адрес этой ячейки в качестве значения функции. Ячейка после этого причисляется к занятой части области списочной памяти.

Может случиться, что в области списочной памяти не было ни одной свободной ячейки. В этом случае подпрограмма *cons* обращается к подпрограмме *reclaim*, называемой также *мусорщиком*. Мусорщик исследует содержимое всех участков памяти, которые распределяются динамически, т. е. во время работы программы. Если на этих участках выявляется *мусор* — ячейки, содержимое которых уже не может понадобиться программе, то он передается в резерв свободной памяти и происходит возврат в подпрограмму *cons*. Если и после уборки мусора в области списочной памяти не образовалось свободных ячеек, то выполнение программы прекращается и печатается сообщение о нехватке памяти — причине, по которой программа не может работать дальше. Работа мусорщика подробно описана в разд. 2.13.

Опишем еще две подпрограммы, преобразующие списочные структуры в памяти машины. Эти подпрограммы имеют по два аргумента. Подпрограмма *rplaca* заменяет содержимое *a*-указателя ячейки,

адрес которой находится в $a1$, адресом, помещенным в ячейку $a2$. За результат принимается (оставшееся неизменным) значение из $a1$. Подпрограмма $rplacd$ также выдает в качестве результата значение $a1$, но помещает адрес, хранящийся в ячейке $a2$, в d -указатель ячейки, адрес которой находится в $a1$. Подпрограмма $rplaca$ не меняет значения разрядов, не входящих в a -указатель, а подпрограмма $rplacd$ — разрядов, не входящих в d -указатель ячейки с адресом из $a1$. Если перед выполнением этих подпрограмм состояние памяти было таким, как показано на рис. 2.3.2, a , то после выполнения подпрограммы $rplaca$ или $rplacd$ оно становится таким, как показано на рис. 2.3.2, b или 2.3.2, $в$ соответственно.

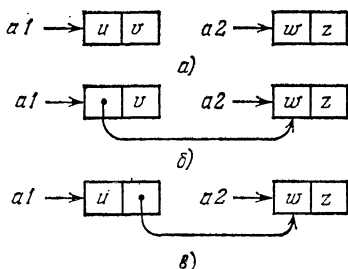


Рис. 2.3.2.

С помощью подпрограмм $rplaca$ и $rplacd$ можно выполнять любые преобразования списочных структур в памяти машины, в том числе и такие, которые не могут быть описаны известными нам средствами языка. В связи с этим в язык обычно включают функции $RPLACA$ и $RPLACD$ класса $SUBR$, с двумя аргументами каждая, позволяющие обращаться к этим подпрограммам. Значение выражения

$(RPLACA\ x\ y)$, если оно представимо в языке, совпадает со значением выражения $(CONS\ y\ (CDR\ x))$, а значение выражения $(RPLACD\ x\ y)$ при том же условии — со значением выражения $(CONS\ (CAR\ x)\ y)$. Однако при обращении к $CONS$ строится новая ячейка в области списочной памяти, а содержимое ячеек, выделенных в этой области ранее, не меняется (поэтому не может измениться значение ни одной

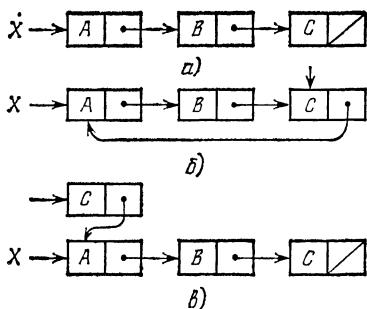


Рис. 2.3.3.

из переменных программы). При обращении к $RPLACA$ или $RPLACD$, наоборот, меняется содержимое одной из ранее занятых ячеек области списочной памяти (а именно, ячейки, соответствующей первому аргументу этих функций), новых же ячеек не создается. Побочным эффектом обращения к $RPLACA$ или $RPLACD$ может

оказаться изменение значений всех выражений, во внутреннем представлении которых участвует преобразованная ячейка.

На рис. 2.3.3, б, в изображены структуры, возникающие в памяти в результате выполнения выражений (*RPLACD (CDDR X) X*) и (*CONS (CAR (CDDR X)) X*) при условии, что значением *X* был список (*A B C*), как это показано на рис. 2.3.3, а. Во втором случае (рис. 2.3.3, в) значение *X* не изменилось, а вычисленное выражение имеет значение (*C A B C*). В первом же случае (рис. 2.3.3, б) не только значение выражения, но и значение *X* после выполнения выражения не имеют эквивалента в языке, так как представляют собой циклические списочные структуры.

Таким образом, применять функции *RPLACA* и *RPLACD* следует с большой осторожностью. Если значение первого аргумента этих функций — атом, то в результате их выполнения изменится содержимое информационной ячейки атома. Это может привести к существенному нарушению работы системы. Примеры полезного применения функции *RPLACD* для преобразования списка свойств атома содержатся в разд. 3.10. Однако во многих реализациях языка в подпрограммах *rplaca* и *rplacd* предусмотрены проверки, разрешающие применять их лишь к ячейкам списочных структур.

2.4. Распределение памяти

Оперативная память вычислительной машины должна быть разбита на несколько участков, различающихся по назначению и характеру их использования. Это следующие участки:

область машинных программ,
область полных слов,
магазин,
ассоциативный список,
список объектов,
область списочной памяти.

В области машинных программ размещаются подпрограммы, реализующие встроенные функции языка, вспомогательные подпрограммы и управляющая программа, объединяющая все эти подпрограммы в единую систему. В этой же области располагаются рабочие ячейки этих программ, в частности ячейки *a1*, *a2*, ..., для аргументов встроенных функций, буферы ввода и вывода и массив *pnbuf*, используемые подпрограммами функций *PRINT* и *READ* (см. разд. 2.6).

В области полных слов размещаются отдельные ячейки или группы ячеек, в которых хранятся внешние наименования нечисло-

вых атомов, значения атомов-чисел, а также располагаются машинные подпрограммы, составленные компилятором.

Магазин предназначен для организации работы рекурсивных подпрограмм. В нем могут храниться адреса возвратов (т. е. адреса переходов после завершения подпрограммы), а также содержимое рабочих ячеек подпрограммы, которое должно быть восстановлено после возврата (см. разд. 2.12).

Ассоциативный список служит для запоминания связи между наименованиями переменных и их активными и пассивными значениями. Во многих реализациях языка ассоциативный список организуется так, как это описано в разд. 3.7, и размещается в области списочной памяти.

Список объектов обеспечивает доступ к информационным ячейкам всех существующих в системе атомов. Он используется при появлении нового атома, чтобы можно было выяснить, действительно ли это новый атом или он уже встречался в программе. Такая потребность возникает при вводе новых выражений, при вычислении арифметических функций и при генерации новых наименований функций *GENSYM* (см. разд. 2.6, 2.11 и 1.27). Список объектов может быть размещен частично в области машинных программ (где хранится его оглавление, см. разд. 2.5), частично в области списочной памяти.

О назначении и структуре списочной памяти уже было сказано в разд. 2.1.

Обычно существует некоторое стандартное распределение памяти между этими областями. В некоторых реализациях пользователю разрешается перед началом работы программы задать желательное ему нестандартное распределение. Если, например, большая часть определяемых в программе функций компилируется, то бывает необходимо расширить область полных слов за счет области списочной памяти. Широкое использование аппарата *PROG* уменьшает, как правило, глубину рекурсий и позволяет поэтому обойтись магазином меньшего размера. Информацию о нехватке места в той или иной конкретной области памяти пользователь получает из сообщения, которое печатается, если в этой области не осталось свободного места (возможно, даже после уборки мусора).

2.5. Список объектов

Проще всего было бы организовать список объектов, выделив участок памяти и размещая на нем информационные ячейки новых атомов по мере их появления. Однако включение нового атома в список было бы при этом довольно трудоемкой операцией. Пришлось бы просматривать весь список объектов и сличать внешнее наимено-

вание (или значение — для атомов-чисел) каждого из них с наименованием (значением) нового атома. Так как список объектов обычно содержит несколько сотен атомов (только наименований встроенных функций и констант в нем больше 100), то на его просмотр уходило бы много времени. Существуют простые способы существенно сократить просмотр ценой дополнительных затрат памяти.

Обычно список объектов организуют, используя *функцию расстановки* (см. [6], разд. 2.6). Функция расстановки — это функция, определенная на множестве всех возможных наименований (значений) атомов и принимающая целочисленные значения из диапазона $1, \dots, m$. В памяти выделяется так называемое *оглавление* списка объектов — массив *hashtable*, состоящий из m ячеек (рис. 2.5.1).

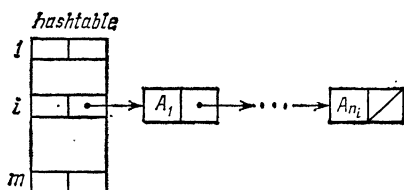


Рис. 2.5.1.

Каждая ячейка оглавления содержит в своем d -указателе адрес начала (первого звена) списка, в который объединены все атомы с одним и тем же значением функции расстановки, равным номеру i этой ячейки в массиве. Каждое звено такого списка содержит в своем a -указателе адрес информационной ячейки одного из атомов A_j , попавших в этот список, а в d -указателе — адрес следующего звена списка. Разумеется, d -указатель последнего звена содержит адрес *nil*. Если же список пуст, то адрес *nil* помещается в d -указателе ячейки оглавления. В машинных программах вместо порядкового номера i ячейки оглавления удобнее пользоваться ее адресом, равным $h + i - 1$, где h — адрес первой ячейки массива *hashtable* (его базовый адрес).

Работу со списком объектов можно описать в виде следующей ниже подпрограммы *intern*, не имеющей эквивалента в языке. Параметрами этой подпрограммы являются: *buf* — массив, в ячейках которого размещается внешнее наименование атома, k — число ячеек, занятых в массиве *buf* наименованием атома, c — номер первой из этих ячеек. Результат работы подпрограммы *intern* — адрес информационной ячейки атома с данным наименованием. Эту ячейку подпрограмма *intern* либо находит в списке объектов, либо создает сама. Подпрограмма *hash (buf, c, k)* вычисляет значение функции

расстановки — адрес одной из ячеек оглавления списка объектов. К нему, как к любому адресу ячейки, в которой выделены a - и d -указатели, применимы функции *car* и *cdr*. Подпрограмма *newatom* выполняется тогда, когда обнаруживается, что атом, наименование которого хранится в ячейках *buf* [c], ..., *buf* [$c + k - 1$], ранее не встречался в программе, т. е. не содержится в списке объектов. Эта подпрограмма создает для этого атома информационную ячейку и список свойств и вырабатывает в качестве значения адрес этой ячейки. В списке свойств (или в самой ячейке) должен содержаться начальный адрес группы ячеек области полных слов, в которую копируется содержимое указанных ячеек массива *buf*.

Логическая функция *comp*, которой в качестве аргументов задаются адрес информационной ячейки некоторого атома, а также значения *buf*, c , k , сравнивает внешнее наименование атома с наименованием, размещенным в массиве *buf*. В случае совпадения вырабатывается ответ «да» (**true**). Остальные подпрограммы, обращения к которым содержатся в приводимом ниже описании процедуры *intern*, были определены в предыдущих разделах.

```
function intern (buf, c, k);
begin r1 := hash (buf, c, k);
  A: if null (cdr (r1)) then
    begin r := newatom; rplacd (r1, cons (r, nil)) end
  else
    begin r1 := cdr (r1); r := car (r1);
      if  $\neg$  comp (r, buf, c, k) then go to A end;
    intern := r
end
```

В подпрограмме *intern* должны быть учтены, кроме того, различия между нечисловыми и числовыми атомами, а также между числами разных типов (*FIX*, *BITS*). Для этого у подпрограммы должен быть дополнительный параметр — признак атома. Подпрограмма *newatom* должна заносить этот признак в список свойств атома, а подпрограмма *comp* — сравнивать признаки атомов перед сравнением их наименований.

Функция расстановки *hash* не обязательно должна зависеть от содержимого всех k ячеек массива *buf*, хранящих наименование атома. Можно использовать только ячейку *buf* [c]. В этом случае все атомы, у которых несколько первых литер (столько, сколько их может быть упаковано в одной машинной ячейке) совпадают, заведомо попадут в один и тот же из списков, подвешенных к позициям оглавления. Но таких атомов обычно бывает немного в одной программе и их поиск в списке не будет слишком долгим.

Один из простейших способов вычисления функции расстановки таков. Содержимое ячейки *buf* [*c*] делится на группы двоичных разрядов одной и той же длины. Эти группы суммируются (после соответствующих сдвигов) как целые числа. Из двоичного представления суммы выделяются *s* разрядов. Целое число, представленное этими разрядами, добавляется к базовому адресу *h* оглавления. Полученная сумма принимается за значение функции расстановки. При этом оглавление должно состоять из $m = 2^s$ ячеек с адресами от *h* до $h + m - 1$.

Массив *buf* можно совместить с областью полных слов, точнее — использовать в качестве ячеек *buf* [*c*], ..., *buf* [*c* + *k* — 1] очередные *k* свободных ячеек этой области. Тогда подпрограмма *newatom* будет лишь завершать перевод этих ячеек из свободных в занятые без копирования, о котором говорилось выше. Если же функция *comp* выработает ответ «да», то эти ячейки вновь следует считать свободными.

Возможно (так сделано в реализации на БЭСМ-6) накапливать внешнее наименование атома в специальном массиве *pnbuf*. В этом случае параметр *s* — лишний, его значение всегда равно 1.

2.6. Подпрограммы функций *PRINT* и *READ*

Прежде чем описывать эти подпрограммы, введем несколько вспомогательных понятий, обозначений и функций. Подпрограмма функции *PRINT* использует массив, называемый *буфером вывода*, в ячейках которого накапливаются упакованные литеры текста, подлежащего выводу на печать. Заносит литеры в буфер вывода подпрограмма *prinl*. Обращение к ней имеет вид *prinl* (*x*), где *x* — адрес информационной ячейки некоторого атома. Все литеры, составляющие внешнее наименование атома, одна за другой переносятся из области полных слов, где они хранятся, в буфер вывода. В случае заполнения буфера до конца его содержимое распечатывается и буфер очищается, так что следующие литеры накапливаются в нем, начиная с первой ячейки. Если аргумент подпрограммы *prinl* представляет атом-число, то оно предварительно переводится в десятичное или восьмеричное (в зависимости от типа числа) представление и в буфер заносятся литеры этого представления. Значение функции *prinl* — это печатаемый атом (т. е. адрес его информационной ячейки).

Содержимое заполненной части буфера можно отпечатать также, обратившись к подпрограмме без параметров *terpri*. Буфер при этом тоже очищается.

С помощью подпрограммы *prinl* можно заносить в буфер вывода и ограничители, если в списке объектов иметь атомы, имеющие

эти ограничители своими внешними наименованиями. Обозначим идентификаторами *lpar*, *rpar*, *period*, *blank* адреса информационных ячеек атомов с внешними наименованиями «(», «)», «.», « \square » (пробел). В некоторых реализациях существуют, кроме того, встроенные константы *LPAR*, *RPAR*, *PERIOD*, *BLANK*, имеющие эти атомы своими значениями.

Структура машинной подпрограммы, реализующей функцию *PRINT*, описывается следующими процедурами:

```
function print (a1);
```

```
begin print := prin0 (a1); terpri end
```

Главную работу здесь выполняет подпрограмма *prin0*, которая заполняет буфер вывода литерами внешнего представления выражения *a1*, печатая его по частям, если оно слишком длинное. Обращение к *terpri* необходимо, чтобы отпечатать накопившийся в буфере остаток выражения

```
function prin0 (a1);
```

```
begin   if atom (a1) then prin1 (a1) else
```

```
    begin prin1 (lpar); r1 := a1;
```

```
    A: prin0 (car (r1)); r1 := cdr (r1);
```

```
    if  $\neg$  null (r1) then
```

```
    begin prin1 (blank);
```

```
        if  $\neg$  atom (r1) then go to A;
```

```
        prin1 (period); prin1 (blank);
```

```
        prin1 (r1)
```

```
    end;
```

```
    prin1 (rpar)
```

```
end;
```

```
prin0 := a1
```

```
end
```

Здесь, например, оператор *prin1* (*lpar*) заносит в буфер вывода литеру «(», а оператор *prin1* (*a1*), который выполняется, только если *a1* — атом, заносит в буфер внешнее наименование этого атома. Подпрограмма *prin0* — рекурсивная; если ее аргумент *a1* — список, то каждый элемент этого списка заносится в буфер в результате рекурсивного обращения к *prin0* с помощью оператора процедуры *prin0* (*car* (*r1*)).

Если наряду с уже упомянутыми выше встроенными константами *LPAR* и другими включить в язык аппарат для обращения к подпрограммам *prin1* и *terpri* (в виде встроенных функций *PRIN1*, за значение которой естественно принять печатаемый атом, и *TERPRI* со значением *NIL*), то программист получит возможность печатать

любые тексты и управлять их расположением на бумаге. Например, текст

$$1) F(X_1, \dots, X_N) = 0$$

с отступлением на 15 позиций от начала печатной строки может быть отпечатан с помощью выражения

```
(PROG (A) (SETQ A 15)
      B (COND ((ZEROP A) (GO C)))
      (PRIN1 BLANK) (SETQ A (SUB1 A))
      (GO B)
      C (SETQ A (LIST 1 RPAR BLANK
                     BLANK (QUOTE F)
                     LPAR (QUOTE X1)
                     PERIOD PERIOD PERIOD
                     (QUOTE ,XN) RPAR
                     (QUOTE =0) ) )
      D (COND ((NULL A) (RETURN (TERPRI))))
      (PRIN1 (CAR A)) (SETQ A (CDR A)) (GO D))
```

если атомы в реализации определены так, как в разд. 1.2, т. е. «X1», «XN» и «=0» допускаются в качестве атомов. Если же такие литеры, как «,», «=» и др., запрещено использовать в составе атомов, то в реализацию необходимо включить дополнительные константы: *КОММА*, значением которой является атом с внешним наименованием «,», *EQSIGN* с атомом «=» в качестве значения и т. п.

Перейдем к подпрограмме, реализующей функцию *READ*. Она использует массив — *буфер ввода*, который содержит в своих ячейках упакованные коды литер выражений, подлежащих вводу. Подпрограмма ввода обрабатывает содержимое буфера первый раз с его начала, а при последующих обращениях — с того места, на котором закончилось чтение выражения при предыдущем обращении. Выборка литер из буфера осуществляется подпрограммой, обращение к которой мы будем записывать в виде указателя функции без параметров *reada*. Задача этой подпрограммы — выбрать из буфера ввода и перенести в массив *pnbuf* (*буфер внешних наименований*) литеры, составляющие один атом. Атомы-числа переводятся в машинное представление и заносятся в первую ячейку массива *pnbuf*. При этом фиксируется тип числа, который учитывается подпрограммой *intern*, хотя это и не было указано явно. Кроме собственно атомов вводимого выражения, подпрограмма *reada* выделяет из исходного текста ограничители (кроме пробела), превращая их в атомы с внешними наименованиями «(,)», «.». Таким же способом возможно трактовать и некоторые другие литеры, если входной язык конкрет-

ной реализации не допускает использовать эти литеры (назовем их *особыми*) в составе внешних наименований.

Итак, подпрограмма *reada* должна работать следующим образом. Она пропускает все пробелы, стоящие в начале того участка текста, который она обрабатывает. Управляющие литеры (перевод строки, возврат каретки и т. п.) приравниваются к пробелам. Если первая содержательная литера (т. е. не пробел и не управляющая литера) — не особая, то она и все следующие за ней литеры того же класса переносятся в массив *pnbuf*. Следующий за этими литерами ограничитель (включая пробел) или особая литера остается в буфере ввода и с нее начнется выборка литер при следующем обращении к *reada*. Если последняя из занятых ячеек массива *pnbuf* осталась незаполненной, то она дополняется кодами, не совпадающими ни с одним из возможных кодов литер. Если в процессе выборки литер будет исчерпано все содержимое буфера ввода, то буфер очищается и заполняется новой порцией вводимого текста и выборка продолжается с начала буфера. Если же первая содержательная литера оказывается ограничителем или особой литерой, то в первой ячейке *pnbuf* формируется внешнее наименование, состоящее из одной этой литеры, и работа подпрограммы *reada* на этом заканчивается. Значение функции *reada* — это число ячеек массива *pnbuf*, занятых внешним наименованием атома.

Подпрограмма *read1* с описанием

```
function read1;  
begin k := reada; read1 := intern (pnbuf, 1, k) end
```

помимо выборки атома, осуществляет поиск выбранного атома в списке объектов, возможно, с занесением его в этот список, если ранее он не встречался. Значение функции *read1* — это выработанный функцией *intern* адрес информационной ячейки атома, выбранного из буфера.

Основная подпрограмма, реализующая функцию *READ*, определяется следующим описанием:

```
function read;  
begin r1 := read1;  
  if eq (r1, lpar) then r1 := read0  
  else if eq (r1, rpar) ∨ eq (r1, period) then error1 (r1);  
  read := r1  
end
```

Правильное лисповское выражение либо является атомом, либо начинается с открывающей скобки. В первом случае все необходимое для ввода атома выполняет оператор *r1 := read1* и значение *r1*

принимается за значение функции *read*. Во втором случае выполняется обращение к подпрограмме *read0*, которая завершает ввод выражения вида

$$(e_1 \ e_2 \ \dots \ e_n) \quad (1)$$

или

$$(e_1 \ e_2 \ \dots \ e_n \cdot e_{n+1}) \quad (2)$$

(открывающая скобка уже прочитана).

Если же в самом начале вводимого текста обнаруживается закрывающая скобка или точка, то выполняется подпрограмма *error1*, которой в качестве аргумента передается адрес атома, представляющего этот незаконный символ. Подпрограмма *error1* должна отпечатать сообщение о характере обнаруженной ошибки и прекратить выполнение программы или же, если программа выполняется в режиме диалога, запросить для ввода новое выражение.

Осталось описать подпрограмму *read0*:

```
function read0;
begin r := cons (nil, nil); r1 := r;
  A: r2 := read1;
    if eq (r2, lpar) then r2 := read0
    else if eq (r2, rpar) then go to E
    else if eq (r2, period) then go to B;
    rplacd (r1, cons (r2, nil));
    r1 := cdr (r1); go to A;
  B: if eq (r1, r) then error2 (cdr (r));
    r2 := read1;
    if eq (r2, lpar) then r2 := read0
    else if eq (r2, rpar) ∨ eq (r2, period)
      then error 2 (cdr (r));
    rplacd (r1, r2); r2 := read1;
    if ⊃ eq (r2, rpar) then error2 (cdr (r));
  E: read0 := cdr (r)
end
```

Во время работы цикла, представленного группой операторов от метки *A* до оператора *go to A* (точнее, в момент прихода на оператор с меткой *A*), ячейки *r* и *r1* содержат адреса начальной и конечной ячеек списочной структуры, изображенной на рис. 2.6.1, а в буфере ввода остается невыбранным текст

$$e_{i+1} \ \dots \ e_n)$$

или

$$e_{i+1} \ \dots \ e_n \cdot e_{n+1})$$

в зависимости от того, имел ли первоначальный текст форму (1) или (2). Оператор $r2 := read0$ выполняется при $i < n$, если выражение e_{i+1} начинается с открывающей скобки. Здесь обращение к $read0$ — рекурсивное. Переход на метку E означает, что $i = n$ и имел место случай (1), возможно, с $n = 0$. Переход на метку B происходит при $i = n$ в случае исходного выражения вида (2). При $i < n$ списочная структура, адрес которой хранится в ячейке r , наращивается новым звеном и в ячейку $r1$ посылается адрес этого звена.

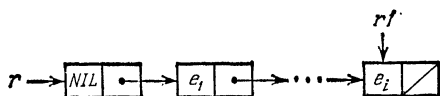


Рис. 2.6.1.

Операторы, следующие за меткой B , анализируют структуру выражения e_{n+1} . На оператор $rplacd(r1, r2)$ мы придем либо, если это выражение — атом, либо после еще одного рекурсивного обращения к $read0$ для чтения выражения e_{n+1} . Этот оператор завершает построение списочной структуры, изображенной на рис. 2.6.1. Затем проверяется, что за выражением e_{n+1} в буфере находится закрывающая скобка. Если в тексте исходного выражения обнаружится ошибка, то один из операторов $error2(cdr(r))$ отпечатает введенную часть выражения и сообщение о характере обнаруженной ошибки.

Операторы $r := cons(nil, nil)$ в начале подпрограммы и $read0 := cdr(r)$ в конце позволяют не выделять особо случай пустого списка, представленного парой стоящих рядом скобок $()$.

2.7. Ассоциативный список

Ассоциативный список, как уже было сказано в разд. 2.4, служит для хранения наименований переменных вместе с их значениями. Традиционная структура ассоциативного списка и функции для работы с ним описаны ниже (см. разд. 3.7). При желании нетрудно перевести описания этих функций на машинный язык. В этом разделе мы опишем другую структуру ассоциативного списка, требующую почти вдвое меньше памяти для размещения списка и во многих отношениях более удобную в работе. Эта структура представляет собой магазин, ячейки которого организованы подобно ячейкам списочной памяти. Большинство ячеек ассоциативного списка содержат в своем a -указателе адрес переменной (т. е. адрес информационной ячейки атома, представляющего эту переменную), а в d -указателе — адрес значения, связанного с этой переменной. Каждая

переменная может встречаться в ассоциативном списке любое число раз. При поиске значения переменной список просматривается от конца к началу, поэтому находится то значение, которое было связано с этой переменной последним. Это и есть активное значение переменной, остальные значения, если они есть в списке, пассивные (см. разд. 1.25).

Среди ячеек, содержащих пары переменная — значение, могут быть вкраплены ячейки, *a*-указатель которых содержит адрес *nil*, а *d*-указатель — адрес одной из предшествующих ячеек ассоциативного списка. Если в процессе поиска значения переменной встречается такая ячейка, то поиск продолжается с места, указанного в ее *d*-указателе. Благодаря этому создается возможность обходить некоторые участки ассоциативного списка. Это необходимо в некоторых случаях при работе с функционалами (см. разд. 2.10).

Опишем две подпрограммы для работы с ассоциативным списком. Пусть ячейка *aptr* содержит адрес первой свободной ячейки ассоциативного списка, ячейка *abeg* — наименьший возможный адрес (адрес начала) этого списка, ячейка *aend* — наибольший возможный адрес.

Для массива, занятого ассоциативным списком, можно не вводить никакого обозначения. Если *a* — адрес одной из ячеек этого списка, то функции *car(a)* и *cdr(a)* открывают доступ к содержимому этой ячейки, а функции *rplaca* и *rplacd* позволяют изменять это содержимое.

Аргументы *a1* и *a2* подпрограммы *pairlis* — это адреса соответственно списка переменных и списка значений, которые мы хотим с ними связать. Подпрограмма *pairlis* используется только как процедура, никакого значения она не вырабатывает.

```

procedure `pairlis (a1, a2);
begin r1 := a1; r2 := a2;
  A: if  $\neg$  null (r1) then
    begin if null (r2) then error3;
      if aptr > aend then
        begin reclaim; if aptr > aend then error4 end;
        rplaca (aptr, car (r1)); rplacd (aptr, car (r2));
        r1 := cdr (r1); r2 := cdr (r2);
        aptr := aptr + 1; go to A
      end
    end
end

```

Здесь подпрограмма *reclaim* — это мусорщик (см. разд. 2.13). В результате обращения к нему могут измениться значения переменных *aptr* и (или) *aend*, так что появится возможность продолжить работу

подпрограммы *pairlis*. Процедуры *error3* и *error4*, как обычно, фиксируют нарушение нормальной работы системы (список значений короче списка переменных или не хватает памяти).

Аргумент *a1* подпрограммы *assoc* — это адрес информационной ячейки некоторого атома, рассматриваемого как наименование переменной. Подпрограмма *assoc* описанным выше способом разыскивает ячейку, содержащую *a1* в своем *a*-указателе. Если поиск успешен, то в качестве результата выдается адрес этой ячейки, в противном случае происходит обращение к процедуре *error5*.

```
function assoc (a1);
begin r1 := aptr;
  A: if r1 ≤ abeg then error5;
      r1 := r1 - 1;
      if null (car (r1)) then r1 := cdr (r1)
      else if eq (car (r1), a1) then go to E;
      go to A;
  E: assoc := r1
end
```

Подпрограммы *pairlis* и *assoc* не являются реализацией функций *PAIRLIS* и *ASSOC*, описанных ниже в разд. 3.7, и не допускают прямого обращения в лисповской программе.

2.8. Интерпретатор

В широком смысле слова *интерпретатор* — это вся система встроенных подпрограмм, позволяющая выполнять лисповские программы без их предварительной компиляции — перевода на язык машинных команд. В этом разделе мы будем понимать термин «интерпретатор» более узко — как программу функции *EVAL* и ряд других тесно связанных с ней подпрограмм.

Большинство из этих подпрограмм выделены в самостоятельные единицы лишь для большей наглядности, в действительности они являются лишь частями одной большой подпрограммы, не имеют самостоятельного значения и не допускают прямого обращения на лиспе.

Подпрограмма *eval* вычисляет значение своего единственного аргумента. Фактически она лишь проверяет, является ли аргумент атомом, после чего дальнейшее вычисление осуществляется подпрограммой *evalatom* или *apply*:

```
function eval (a1);
  if atom (a1) then eval := evalatom (a1)
  else eval := apply (car (a1), cdr (a1))
```

Работу подпрограммы *evalom* можно описать в виде процедуры

```
function evalom (a1);  
  if get (a1, apval, r1) then evalom := r1  
  else evalom := cdr (assoc (a1))
```

Здесь используется логическая функция *get*, которая исследует список свойств атома, указанного в качестве ее первого аргумента. Если в этом списке есть свойство с индикатором, указанным в качестве второго аргумента, то это свойство присваивается переменной, заданной третьим аргументом, а значением функции считается **true**. В противном случае функция вырабатывает значение **false**, а значение переменной, указанной в качестве третьего аргумента, не меняется.

Таким образом, если аргумент подпрограммы *evalom* наделен свойством с индикатором *APVAL* (индикаторы *FIX* и *BITS* в данном случае следует рассматривать как частные случаи индикатора *APVAL*), то в качестве значения функции выдается это свойство. Как было сказано в разд. 2.2, для чисел этим свойством служит адрес информационной ячейки числа. Атом, не имеющий в своем списке свойств индикатора *APVAL* (*FIX*, *BITS*), рассматривается как переменная. Ее значение разыскивается в ассоциативном списке и извлекается из *d*-указателя ячейки, адрес которой находит функция *assoc*.

Подпрограмме *apply* соответствует описание

```
function apply (a1, a2);  
  if atom (a1) then apply := evfun (a1, a2) else  
begin r1 := car (a1) ; r2 := cdr (a1) ;  
  if eq (r1, lambda) then  
    apply := evlam (r2, evlis (a2))  
  else if eq (r1, funarg) then  
    apply := evfarg (r2, a2)  
  else error6  
end
```

Первый аргумент подпрограммы *apply* может быть либо наименованием функции, либо определяющим выражением, либо функциональным аргументом, обработанным специальной функцией *FUNCTION* (см. разд. 2.10). Дальнейшее вычисление осуществляется соответственно подпрограммой *evfun*, *evlam* или *evfarg*. Работа подпрограммы *evfarg* будет разобрана в разд. 2.10. Во всех случаях второй аргумент — это список аргументов интерпретируемой функции. При обращении к *evlam* он заменяется списком значений этих

аргументов, вычисляемым с помощью подпрограммы *evalis*:

```
function evalis (a1) ;  
begin save (aptr0) ; aptr0 := nil ;  
      evalis := list (a1) ;  
      restore (aptr0) ; bridge  
end
```

Главную работу здесь выполняет оператор *evalis := list (a1)*, остальные операторы нужны лишь для обслуживания механизма функциональных аргументов (см. разд. 2.10). Подпрограмма *list* реализует специальную функцию *LIST*. Она строит список значений выражений, из которых состоит список, переданный ей как аргумент.

```
function list (a1) ;  
begin r := cons (nil, nil) ; r1 := a1 ; r2 := r ;  
  A : if  $\neg$  null (r1) then  
    begin rplacd (r2, cons (eval (car (r1)), nil)) ;  
          r1 := cdr (r1) ; r2 := cdr (r2) ; go to A end ;  
    list := cdr (r)  
end
```

Подпрограмма *evfun* исследует список свойств атома, заданного в качестве наименования интерпретируемой функции. Если в списке свойств обнаруживается индикатор *EXPR* или *FEXPR*, то соответствующим свойством является определяющее выражение функции, из которого выброшен первый элемент — атом *LAMBDA*. Это выражение вместе со списком аргументов (для *FEXPR*) или их значений (для *EXPR*) передается подпрограмме *evalam* для завершения интерпретации. В случае обнаружения индикаторов *SUBR* или *FSUBR* управление передается машинной подпрограмме, начальный адрес которой извлекается из списка свойств. Для *FSUBR* в ячейку *a1* предварительно помещается адрес списка аргументов функции, для *SUBR* подпрограмма *spread* размещает в ячейках *a1*, *a2*, ... значения аргументов. Здесь, в отличие от сказанного в разд. 2.3, ячейки *a1*, *a2*, ... считаются глобальными. Подробнее характер использования этих ячеек разъяснен ниже, в разд. 2.12.

Если названных индикаторов в списке свойств наименования функции нет, то это наименование рассматривается как переменная. Соответствующее ей значение извлекается из ассоциативного списка и передается вместе со списком аргументов подпрограмме *apply*:

```
function evfun (a1, a2);  
  if get (a1, expr, r1) then  
    evfun := evalam (r1, evalis (a2))  
  else if get (a1, fexpr, r1) then  
    evfun := evalam (r1, a2)
```



```

else if get (a1, subr, r1) then
begin spread (evalis (a2)); call (r1);
    evfun := result end
else if get (a1, fsubr, r1) then
begin a1 := a2 ; call (r1);
    evfun := result end
else if get (a1, apval, r1) then error?
else begin r1 := assoc (a1);
    evfun := apply (cdr (r1), a2) end

```

Подпрограмма *evalam* работает следующим образом:

```

function evalam (a1, a2);
begin r1 := cadr (a1) ; save (aptr) ; pairlis (car (a1), a2);
    evalam := eval (r1) ; restore (aptr) end

```

Она извлекает из определяющего выражения адрес тела *e* интерпретируемой функции, запоминает в магазине значение указателя *aptr*, связывает в ассоциативном списке наименования переменных интерпретируемой функции с их значениями, вычисляет выражение *e* при получившемся состоянии ассоциативного списка и, наконец, восстанавливает значение *aptr*, возвращаясь тем самым к исходному состоянию ассоциативного списка.

Процедуры *save* и *restore*, используемые для запоминания в магазине нужных значений и для их последующего восстановления, будут описаны в разд. 2.12.

Подпрограмму *spread* мы не описываем. Ее аргумент — это список значений аргументов интерпретируемой функции. Первый элемент этого списка она помещает в ячейку *a1*, второй — в ячейку *a2*. Заметим, что в описанной выше версии языка нет функций класса *SUBR* с более чем двумя аргументами. Ответ на вопрос — откуда подпрограмма *spread* узнает адреса ячеек *a1* и *a2*, которые в разд. 2.3 были объявлены локальными для каждой встроенной функции, мы отложим до разд. 2.12.

Оператор *call (r1)* означает передачу управления (с возвратом) машинной подпрограмме, начальный адрес которой находится в ячейке *r1*. Предполагается, что эта подпрограмма помещает результат своей работы (значение функции) в ячейку *result*.

Рассмотрим схемы подпрограмм некоторых специальных встроенных функций. Мы уже говорили, что перед передачей управления этим подпрограммам адрес списка их аргументов помещается в ячейку *a1*. Так, для функции *QUOTE* в ячейке *a1* будет находиться список из одного элемента — аргумента *QUOTE*. Работа подпрограммы *quote* сводится к выделению этого единственного элемента:

```

function quote (a1) ; quote := car (a1)

```

Более сложной, но вполне естественной оказывается подпрограмма функции *COND*:

```
function cond (a1);
begin r1 := a1;
  A : if null (r1) then cond := nil else
    begin r2 := car (r1);
      if null (eval (car (r2))) then
        begin r1 := cdr (r1) ; go to A end;
        cond := eval (cadr (r2))
      end
    end
end
```

Эта подпрограмма поочередно помещает в ячейку *r2* адреса аргументов функции *COND*, имеющих вид $(p_i e_i)$, вычисляет значение p_i : если оно равно *NIL*, то переходит к следующему аргументу, а если нет, то выдает в качестве результата значение e_i . В некоторых реализациях вместо присваивания *cond := nil* фиксируется ошибка в условном выражении (т. е. в обращении к *COND*), если все выражения p_i имеют значение *NIL*.

Функция *SEXPR* реализуется в виде подпрограммы

```
function sexpr (a1);
begin r1 := car (a1); r2 := cadr (a1) ;
  if  $\neg atom (r1) \vee \neg eq (car (r2), lambda)$  then error8;
  put (r1, expr, cdr (r2)); sexpr := r1 end
```

Если нарушены правила обращения к *SEXPR* (первый аргумент — не атом или второй — не определяющее выражение), то оператор *error8* сообщает о невозможности выполнить это обращение. Процедура *put (a, i, p)* помещает в список свойств атома *a* свойство *p* под индикатором *i*. В данном случае *p* — это определяющее выражение, из которого выбрасывается атом *LAMBDA*, а индикатор *i* — атом *EXPR*. Реализация *SFEXPR* отличается лишь тем, что в качестве индикатора берется атом *FEXPR*.

Предикат *AND* реализуется по следующей схеме:

```
function and (a1);
begin r1 := a1;
  A : if null (r1) then and := t
    else if null (eval (car (r1))) then and := nil
    else begin r1 := cdr (r1); go to A end
end
```

Аналогично реализуется предикат *OR* (см. также определения функций *AND1* и *OR1* в разд. 1.30).

2.9. Интерпретация *PROG*

Следующее описание отображает структуру подпрограммы, реализующей функцию *PROG*:

```
function prog (a1) ;
begin save (aptr) ; pairlis (car (a1), nillist);
      stlist := cdr (a1); lablist := segment (stlist);
      switch := 0;
  A : if null (stlist) then prog := nil else
      begin r1 := car (stlist);
          if  $\neg$ atom (r1) then
              begin r1 := eval (r1);
                  if switch  $\neq$  0 then
                      begin if switch < 0 then go to B;
                          stlist := lassoc (car (r1), lablist);
                          switch := 0
                      end
                  end;
                  stlist := cdr (stlist); go to A;
              B : switch := 0; prog := r1
          end;
      restore (aptr)
  end
```

Здесь *nillist* — адрес циклической списочной структуры, имитирующей бесконечный список, составленный из атомов *NIL* (рис. 2.9.1).



Эта структура используется в обращении к *pairlis*, чтобы связать в ассоциативном списке каждую программную переменную со значением *NIL*. Старое значение указателя ассоциативного списка предварительно запоминается в магазине, а перед выходом из подпрограммы *prog* восстанавливается.

В ячейку *stlist* помещается адрес списка операторов, а в ячейку *lablist* — адрес списка, элементами которого являются отрезки списка операторов от каждой из меток до конца. Этот список строит подпрограмма *segment* с описанием

```
function segment (a1);
begin r1 := a1; r2 := nil;
  A : if null (r1) then segment := r2 else
      begin if atom (car (r1)) then r2 := cons (r1, r2);
          r1 := cdr (r1); go to A end
  end
```

С метки *A* в теле подпрограммы *prog* начинается основной цикл интерпретации обращения к *PROG*. Из этого цикла мы выходим, когда

список операторов будет исчерпан, а также если (после выполнения оператора *RETURN*) управление будет передано на метку *B*. Продвижение по списку операторов происходит, если очередной оператор — метка или если после выполнения этого оператора значение переменной *switch* равно нулю — это означает, что выполненный оператор — не *RETURN* и не *GO*. Подпрограмма оператора (*RETURN e*) присваивает переменной *switch* значение —1, а подпрограмма оператора (*GO l*) — значение 1. Значением самого оператора (*GO l*) является список (*l*). После этого подпрограмма *lassoc* просматривает список, присвоенный переменной *lablist*, и выделяет из него отрезок списка операторов, начинающийся с атома-метки *l*.

Далее, работа основного цикла подпрограммы *prog* продолжается над этим отрезком списка операторов (метка *l* предварительно выбрасывается).

Описание подпрограммы *lassoc* таково:

```
function lassoc (a1, a2);
begin r1 := a2;
  A : if null (r1) then error9 (a1);
      r2 := car (r1);
      if eq (a1, car (r2)) then lassoc := r2
      else begin r1 := cdr (r1); go to A end
end
```

Описанная выше схема интерпретации *PROG* обладает достаточной гибкостью. Она допускает, например, включать в программу операторы типа

(*EVAL (LIST (QUOTE GO) X)*)

для перехода на метку, являющуюся значением переменной *X*. Можно также помещать операторы *GO* и *RETURN* внутри операторов *COND*, которые в свою очередь могут вкладываться друг в друга на любую глубину. Для этого не требуется ничего менять в подпрограмме *cond* из предыдущего раздела. Целесообразно, впрочем, сделать механизм условных операторов несколько более общим, допуская во входном языке условные конструкции вида

(*COND (p₁ e₁₁ ... e_{1m₁}) ... (p_n e_{n1} ... e_{nm_n})*) (1)

где $m_i \geq 0$ для $i = 1, \dots, n$. Если p_i — первое среди выражений p_1, \dots, p_n , имеющее значение, отличное от *NIL*, то последовательно вычисляются выражения e_{i1}, \dots, e_{im_i} . В качестве значения выражения (1) принимается либо значение e_{im_i} , либо значение,работанное оператором (*GO l*) или (*RETURN e*), если он встретится

в последовательности выражений e_{ij} или внутри другого обращения к *COND*, являющегося членом этой последовательности (следующие выражения при этом уже не вычисляются). Если же $m_i = 0$ или все p_i имеют значение *NIL*, то значением выражения (1) считается *NIL*. Подпрограмму *cond* следует переделать следующим образом:

```
function cond (a1);
begin r1 := a1;
  A : if null (r1) then cond := nil else
    begin r2 := car (r1);
      if null (eval (car (r2))) then
        begin r1 := cdr (r1); go to A end;
        r1 := cdr (r2);
        if null (r1) then cond := nil else
      B : begin r2 := eval (car (r1)); r1 := cdr (r1);
            if null (r1) ∨ switch ≠ 0 then cond := r2
            else go to B end
    end
end
```

Встроенная функция *RETURN* относится к классу *SUBR*, поэтому при обращении (*RETURN e*) значение v выражения e вычисляется и помещается в ячейку *a1*. Оно же принимается за значение *RETURN*, поэтому все, что остается сделать, отражено в описании

```
function return (a1);
begin switch := -1; return := a1 end
```

При обращении (*GO l*) к функции *GO* класса *FSUBR* в ячейку *a1* засылается адрес списка (*l*). Поэтому подпрограмма *go* должна иметь вид

```
function go (a1);
begin switch := 1; go := a1 end
```

Функция *SETQ* класса *FSUBR* реализуется по схеме

```
function setq (a1);
begin r1 := eval (cadr (a1)); r2 := assoc (car (a1));
  rplacd (r2, r1); setq := r1 end
```

Эта подпрограмма находит в ассоциативном списке ячейку, содержащую активную связь заданной переменной, и заменяет содержимое *d*-указателя этой ячейки адресом значения данного выражения.

Реализации языка обычно содержат также функцию *SET* класса *SUBR*, отличающуюся от *SETQ* тем, что переменная, с которой надо связать новое значение, задается не явно, а как значение

первого аргумента функции, так что операторы (*SETQ v e*) и (*SET (QUOTE v) e*) эквивалентны. Функция *SET* реализуется по схеме

```
function set (a1, a2);
begin rplacd (assoc (a1), a2); set := a2 end
```

2.10. Функциональные аргументы

В разд. 1.36 было сказано, что у переменных, свободно входящих в функциональный аргумент (т. е. не являющихся его связанными переменными), должны сохраняться те значения, которыми они обладали в момент задания функционального аргумента. Поэтому в ассоциативном списке необходимы мостики, по которым при поиске значения переменной обходится часть содержимого этого списка. О возможном существовании таких мостиков и их использовании подпрограммой *assoc* говорилось в разд. 2.7. Теперь рассмотрим, как они возникают.

Функциональный аргумент (по крайней мере, если он может использовать свободные переменные) должен быть задан выражением

(*FUNCTION fn*) (1)

где *fn* — определяющее выражение или наименование функции. При вычислении его значения происходит обращение к функции *FUNCTION* класса *FSUBR*, работающей по следующей схеме:

```
function function (a1);
begin r1 := car (a1);
  if atom (r1) then
    begin if get (r1, expr, r1) then go to A
          else if get (r1, subr, r2) ∨ get (r1, fsubr, r2) then
            begin function := r1; go to E end
          else error14 end
        else if eq (car(r1), lambda) then r1 := cdr (r1)
        else error15;
```

A : function := cons (funarg, cons (r1, cons (aptr, nil)));

E : end

где, как обычно, *funarg* обозначает адрес информационной ячейки атома *FUNARG*. Таким образом, значением выражения (1) является либо *fn*, если это — наименование встроенной функции, либо список

(*FUNARG del ass₀*) (2)

где *del* отличается от *fn* или от определяющего выражения функции с наименованием *fn* лишь отсутствием первого элемента — атома *LAMBDA*. В этом списке фиксируется также текущее значение *ass₀*

указателя *aptr* ассоциативного списка в момент задания функционального аргумента — первый устой моста. Полученное значение — наименование *fn* или выражение (2) — связывается в ассоциативном списке с функциональной переменной *fv*. Как всякая переменная, она не должна быть наделена ни одним из свойств *SUBR*, *FSUBR*, *EXPR*, *FEXPR* или *APVAL*. Поэтому, когда дойдет очередь до вычисления обращения к функции с наименованием *fv* (см. описание подпрограммы *evfun* в разд. 2.8), произойдет обращение к подпрограмме *assoc*, которая найдет значение, связанное с этим наименованием. Затем возникнет обращение к подпрограмме *apply* с этим значением в качестве первого аргумента (вторым аргументом будет список аргументов обращения к *fv*). Подпрограмма *apply* в свою очередь обратится либо снова к *evfun* с первым аргументом — атомом *fn*, либо к подпрограмме *evfarg*, передав ей в качестве первого аргумента список (*del ass₀*)

Подпрограмма *evfarg* имеет следующую структуру:

```
function evfarg (a1, a2);
begin aptr0 := cdr(a1); save(aptr); r1 := evlis(a2);
      evfarg := evlam(car(a1), r1); restore(aptr)
end
```

Она присваивает переменной *aptr0* значение (*ass₀*) и обращается к *evlis* для вычисления списка значений аргументов. Из *evlis* произойдет обращение к подпрограмме *bridge*:

```
procedure bridge;
  if  $\neg$  null (aptr0) then
    begin pairlis (cons(nil, nil), aptr0);
      aptr0 := nil end
```

Эта подпрограмма с помощью *pairlis* создает второй устой моста — ячейку ассоциативного списка с адресом *nil* в *a*-указателе и адресом *ass₀* в *d*-указателе. Только после этого в *evfarg* возникнет обращение к *evlam*. Эта подпрограмма, связав значения аргументов обращения к *fv* с переменными из *del*, начнет вычислять тело определяющего выражения *del*. Благодаря построенному мосту в ассоциативном списке будут доступны только те связи, которые существовали в момент задания функционального аргумента (1) и, разумеется, те, которые возникнут в ходе выполнения обращения к *fv*.

При интерпретации функций, не являющихся функциональными аргументами, значение *aptr0* равно *nil* и подпрограмма *bridge* срабатывает вхолостую. Так как подпрограмма *bridge* меняет значение *aptr*, в подпрограмме *evfarg* предусмотрено его запоминание и восстановление.

Вернемся к примеру из разд. 1.36. При обращении (1.36.3) ассоциативный список (до этого пустой) перейдет в состояние

$$a_0 = ((F . de_1) (X . x_1)) \quad (3)$$

где для краткости введены обозначения

$$\begin{aligned} x_1 &= (A \ B \ (C \ D)) \\ de_1 &= (LAMBDA \ (X) \ (CONS \ X \ (QUOTE \ W))) \end{aligned}$$

Символы a_0 , a_1 , a_2 , ... обозначают различные состояния ассоциативного списка, представленные в списочной записи.

Начнется выполнение тела функции *MAPCAR*:

$$(MAPLIST \ X \ (FUNCTION \ (LAMBDA \ (X) \ (F \ (CAR \ X)))))$$

Для переменных X и F (теперь уже как связанных переменных функции *MAPLIST*) в ассоциативном списке возникнут новые связи:

$$a_1 = ((F . fa_1) (X . x_1) (F . de_1) (X . x_1))$$

где

$$\begin{aligned} fa_1 &= (FUNARG \ de_2 \ a_0) \\ de_2 &= ((X) \ (F \ (CAR \ X))) \end{aligned}$$

При этом состоянии ассоциативного списка начнет выполняться тело функции *MAPLIST*:

$$\begin{aligned} (COND \ ((NULL \ X) \ NIL) \\ (T \ (CONS \ (F \ X) \ (MAPLIST \ (CDR \ X) \ F))) \) \end{aligned}$$

Поскольку значение X — не *NIL*, значение этого условного выражения совпадает со значением выражения

$$(CONS \ (F \ X) \ (MAPLIST \ (CDR \ X) \ F)) \quad (4)$$

Остановимся подробнее на вычислении первого аргумента, т. е. $(F \ X)$. Последовательно возникнут обращения к подпрограммам:

apply с аргументами $a1 \rightarrow F$, $a2 \rightarrow (X)$,
evalfun с аргументами $a1 \rightarrow F$, $a2 \rightarrow (X)$,
apply с аргументами $a1 \rightarrow fa_1$, $a2 \rightarrow (X)$,
evalfarg с аргументами $a1 \rightarrow (de_2 \ a_0)$, $a2 \rightarrow (X)$.

Подпрограмма *evalfarg* присвоит переменной *aptr0* значение (a_0) и обратится к *evalis* с аргументом (X) . Функция *evalis* выработает (с помощью *list*) значение (x_1) , используя активную связь переменной X , и запустит подпрограмму *bridge* (впервые при значении *aptr0*, отличном от *nil*). Эта подпрограмма переведет ассоциативный список

в состоянии

$$a_2 = ((NIL . a_0) (F . fa_1) (X . x_1) (F . de_1) (X . x_1))$$

Далее из *evfarg* произойдет обращение к *evlam* с аргументами

$$a1 \rightarrow ((X) (F (CAR X))), a2 \rightarrow (x_1)$$

В ассоциативном списке возникнет еще одна связь для X :

$$a_3 = ((X . x_1) (NIL . a_0) \overbrace{(F . fa_1) (X . x_1) (F . de_1) (X . x_1)}^{\downarrow}) \quad (5)$$

и при таком состоянии этого списка станет вычисляться выражение

$$(F (CAR X)) \quad (6)$$

Благодаря мостику, изображенному стрелкой в формуле (5), для F теперь будет найдено значение de_1 , так что выражение (6) как бы заменится выражением

$$(de_1 (CAR X)) \quad (7)$$

Значением $(CAR X)$ будет атом A , так что состояние ассоциативного списка перед вычислением тела определяющего выражения de_1 окажется таким

$$a_4 = ((X . A) (X . x_1) (NIL . a_0) (F . fa_1) (X . x_1) (F . de_1) (X . x_1))$$

Значением этого тела

$$(CONS X (QUOTE W))$$

будет выражение $(A . W)$. С этим же значением завершится вычисление обращения (7) и ассоциативный список вернется в состояние a_3 . Таким же окажется и значение выражения (6). Когда завершится упомянутое выше обращение к *evlam*, ассоциативный список вновь примет вид a_2 . Затем оператор

$$restore (aptr)$$

вернет ассоциативный список в состояние a_1 . Немедленно вслед за этим завершатся все перечисленные обращения к *apply*, *evfun* и *evfarg*, возникшие при вычислении значения первого аргумента выражения (4). После этого начнет вычисляться второй аргумент

$$(MAPLIST (CDR X) F)$$

и в ассоциативном списке появятся такие связи

$$a_5 = ((F . fa_1) (X . x_2) (F . fa_1) (X . x_1) (F . de_1) (X . x_1))$$

где

$$x_2 = (B (C D))$$

Читателю предлагается в качестве упражнения завершить разбор интерпретации обращения (1.36.3).

Теперь можно объяснить, почему функционалом не может быть функция класса *FEXPR* (см. разд. 1.36). У таких функций переменные связываются с аргументами, а не с их значениями. Поэтому, если даже предпринять попытку вычислить значение функционального аргумента после входа в тело функционала, то момент будет уже упущен и в полученном выражении вида (2) значение *ass₀* не будет соответствовать условиям, в которых был задан функциональный аргумент.

Что касается функционального аргумента вида (*FUNCTION fn*), где *fn* — наименование функции класса *FEXPR*, то его можно было бы допустить, потребовав, чтобы подпрограмма *function* вырабатывала для него значение, отличающееся от (2), например, значение (*FFARG del ass₀*). Подпрограмма *apply* должна обнаруживать атом *FFARG* наряду с *FUNARG* и *LAMBDA* и запускать в работу вместо *evfarg* другую подпрограмму:

```
function evffarg (a1, a2);  
begin save(aptr); pairlis(cons(nil, nil), cdr(a1));  
      evffarg := evalam(car(a1), a2); restore (aptr)  
end
```

работа которой должна быть понятна по аналогии с *evfarg*.

2.11. Арифметические функции

Реализация арифметических функций довольно проста. Значениями их аргументов должны быть числовые атомы типа, соответствующего виду функции. Из списков свойств этих атомов извлекаются их числовые значения, над которыми и выполняется соответствующая операция. К полученному результату применяется функция *intern* (см. разд. 2.5), которая и вырабатывает значение функции — адрес информационной ячейки вновь созданного или ранее существовавшего атома.

Для примера опишем подпрограмму, реализующую функцию *PLUS*. Она относится к классу *FSUBR*, так как допускает произвольное число аргументов. Перед началом работы подпрограммы адрес списка аргументов помещается в ячейку *a1*.

```
function. plus (a1);  
begin r1 := a1; r2 := 0;  
  A : if  $\neg$  null (r1) then  
    begin r3 := eval (car (r1)); r1 := cdr (r1);
```

```

        if get (r3, fix, r3) then r2 := r2 + val (r3)
        else error10 (r3);
        go to A end;
    pnbuf [1] := r2; plus := intern (pnbuf, 1, 1)
end

```

Под выражением *val (r3)* понимается числовое значение, адрес которого содержится в списке свойств атома-числа *r3* (см. разд. 2.2). Предусмотрена проверка, что значение очередного аргумента в обращении к *PLUS* — это действительно число.

Остальные подпрограммы, как правило, еще проще. Подпрограмма предиката *ZEROP*, например, такова:

```

function zerop (a1);
    if a1 = zero then zerop := t else zerop := nil

```

где *zero* обозначает адрес информационной ячейки атома 0, который наряду с другими употребительными числами целесообразно постоянно иметь в системе.

Опишем еще подпрограмму *sadd1*, которая реализует полуарифметическую функцию (оператор) *SADD1* класса *FSUBR*.

```

function sadd1 (a1);
begin r1 := assoc (car (a1));
    if get (cdr (r1), fix, r2) then
        begin pnbuf [1] := val (r2) + 1;
            r2 := intern (pnbuf, 1, 1);
            rplacd (r1, r2); sadd1 := r2 end
        else error11 (r1)
end

```

Оператор (*SADD1 v*) проверяет, что переменная *v* имеет числовое значение, и присваивает ей новое значение, на 1 больше предыдущего. Полезна также функция *SSUB1*, уменьшающая значение своего аргумента-переменной на 1.

2.12. Организация рекурсивных подпрограмм

Многие процедуры, описанные в предыдущих разделах, были рекурсивными. Известно, что организация работы рекурсивных подпрограмм в общем случае оказывается довольно сложным делом. Во-первых, надо позаботиться о сохранении адреса возврата. Эта проблема встает даже для нерекурсивных подпрограмм, если они могут обращаться к другим подпрограммам, а адрес возврата при обращении к любой подпрограмме засылается в одну и ту же стандартную ячейку

или помещается на один и тот же индекс-регистр. Во-вторых, необходимо сохранить содержимое некоторых рабочих ячеек подпрограммы, если оно может понадобится после возврата из рекурсивного обращения к той же подпрограмме.

Первую задачу можно решить следующим образом. В начале подпрограммы, из которой возможны обращения к другим подпрограммам, следует запомнить в магазине адрес возврата.

Магазин представляет собой массив из s ячеек:

array *stack* [$1 : s$]

Часть этого массива заполнена информацией, а часть, начиная с ячейки с номером p , — свободна. Для запоминания в магазине содержимого ячейки x следует выполнить оператор *save* (x), обращающийся к подпрограмме

```
procedure save ( $x$ );  
begin if  $p > s$  then error12 ( $p$ );  
      stack [ $p$ ] :=  $x$ ;  $p := p + 1$  end
```

Условный оператор проверяет, не переполнен ли магазин. Если да, то процедура *error12* напечатает сообщение об этом и приостановит выполнение программы.

Для восстановления содержимого ячейки x , хранящегося в магазине, надо обратиться к процедуре

```
procedure restore ( $x$ );  
begin if  $p \leq 1$  then error13 ( $p$ );  
       $p := p - 1$ ;  $x := \textit{stack}$  [ $p$ ] end
```

Восстановление содержимого нескольких ячеек будет происходить правильно, если оно совершается в порядке, в точности обратном тому, в каком происходило запоминание. Поэтому, даже если выяснилось, что содержимое некоторой ячейки, сохраняющееся в магазине, нам уже не потребуется, все равно в подходящий момент следует выполнить оператор *restore* (x) или по крайней мере оператор $p := p - 1$, чтобы освободить магазин.

Каждая подпрограмма, в начале работы которой производилось запоминание адреса возврата, после завершения своей работы должна передавать управление блоку выхода. Этот блок восстанавливает хранившееся в магазине значение адреса возврата и передает управление по этому адресу, осуществляя тем самым возврат к месту обращения (точнее, по адресу, заданному при обращении).

Вторая задача — сохранение содержимого рабочих ячеек подпрограммы на время обращения к той же или другой подпрограмме —

может решаться разными способами. Один из них заключается в том, что при каждом обращении к подпрограмме для нее выделяется новая группа рабочих ячеек из резерва свободной памяти. После завершения работы подпрограммы эти ячейки возвращаются в резерв. Результат работы подпрограммы предварительно пересылается в стандартную ячейку. Этот способ обычно используется в подпрограммах, составленных компилятором, и подробнее будет описан в разд. 2.14. Заметим, что и при этом способе должна существовать по крайней мере одна ячейка, общая для всех обращений к подпрограммам, — ячейка, в которой хранится начальный адрес группы остальных рабочих ячеек (в разд. 2.14 она обозначена идентификатором *base*). Содержимое этой ячейки надо сохранять несколько иным способом. Кроме того, этот способ расходует больше памяти, чем это требуется. Может случиться, что значения одной или нескольких переменных подпрограммы уже не потребуются после возврата в данную подпрограмму. Тем не менее для этих переменных при рекурсивном обращении к подпрограмме будут выделены новые рабочие ячейки вместо того, чтобы использовать старые. Поэтому во встроенных подпрограммах чаще используется другой способ. Анализируется порядок использования рабочих ячеек подпрограммы, и перед тем, как обращаться к другой (или той же) подпрограмме, которая может испортить еще нужное содержимое некоторых из этих ячеек, оно запоминается в магазине. После возврата в данную подпрограмму восстанавливается содержимое тех рабочих ячеек, для которых производилось запоминание.

Для примера обратимся к процедурам *print* и *prin0*. Перепишем эти процедуры в виде последовательностей операторов без явной рекурсии. Вместо того, что говорилось в разд. 2.3 о стандартных ячейках *a1*, *a2*, ... для аргументов и *r*, *r1*, *r2*, ... для промежуточных результатов встроенных подпрограмм, будем теперь считать эти ячейки глобальными. Введем переменную *raddr*, принимающую значения типа «метка» и процедуру *jump (raddr)*, выполнение которой означает переход на метку, которая перед этим была присвоена переменной *raddr*. Будем считать, что группы операторов, заменяющие процедуры *car*, *cdr*, *atom*, *prin1* и *terpri*, не обращающиеся ни к каким другим процедурам, завершаются просто оператором *jump (raddr)* (а не передачей управления блоку выхода, как это необходимо в более сложных случаях). Итак, процедуре *atom* соответствует такая группа операторов:

atom: if car (a1) = alpha then go to true; go to false

Здесь *car (a1)* обозначает просто содержимое *a*-указателя ячейки с адресом, взятым из *a1*, которое извлекается без обращения к под-

программе *car* (иначе перед обращением к *car* необходимо было бы запомнить значение *raddr*, с которым произошло обращение к *atom*). Метками *true* и *false* помечены группы операторов, используемые и другими предикатами:

```
true : r := t; jump (raddr);  
false : r := nil; jump (raddr)
```

Здесь *r* — стандартная ячейка (или регистр), в которую каждая встроенная подпрограмма помещает вырабатываемое ею значение.

Каждая подпрограмма, обращающаяся к другим подпрограммам, должна начинаться с оператора *save* (*raddr*), запоминающего адрес возврата, и завершаться переходом на блок выхода. Этот блок состоит из операторов

```
ex: restore (raddr); jump (raddr)
```

Для печати выражения, адрес которого находится в ячейке *x*, с последующим возвратом на метку *m* надо выполнить операторы

```
a1 := x; raddr := m; go to print
```

Вместо процедуры *print* (см. разд. 2.6) следует написать операторы

```
print : save (raddr); save (a1); raddr := pr1; go to prin0;  
pr1 : raddr := pr2; go to terpri;  
pr2 : restore (r); go to ex;
```

У процедуры *prin0* разд. 2.6 две рабочие ячейки: *a1* и *r1*. Первая ячейка совпадает с ячейкой *a1* процедуры *print*. Ее содержимое уже было заслано в магазин написанными выше операторами. Поэтому операторы, соответствующие процедуре *prin0*, должны только в надлежащее время запоминать и восстанавливать значение переменной *r1*:

```
prin0 : save (raddr); r1 := a1; raddr := pr01; go to atom;  
pr01 : if r=nil then go to B;  
          a1 := r1; raddr := ex; go to print;  
          B : a1 := lpar; raddr := A; go to print;  
          A : a1 := r1; raddr := pr02; go to car;  
pr02 : save (r1); a1 := r; raddr := pr03; go to prin0;  
pr03 : restore (a1); raddr := pr04; go to cdr;  
pr04 : if r=nil then go to C; r1 := r;  
          a1 := blank; raddr := pr05; go to print;
```

```

pr05 : a1 := r1; raddr := pr06; go to atom;
pr06 : if r=nil then go to A;
        a1 := period; raddr := pr07; go to prin1;
pr07 : a1 := blank; raddr := pr08; go to prin1;
pr08 : a1 := r1; raddr := C; go to prin1;
C : a1 := rpar; raddr := ex; go to prin1

```

Полезно убедиться, что для избавления от рекурсии здесь использован тот же прием, что и в разд. 1.26. Предполагается, что подпрограммы *atom*, *car*, *cdr* не портят содержимое ячейки *r1*, но портят содержимое *a1* (на практике вместо ячеек *a1* и *r* используется один и тот же индекс-регистр). Вместо явного перехода на блок выхода начальный адрес *ex* этого блока засылается в нужный момент (см. последнюю строчку подпрограммы) в ячейку *raddr*.

На большинстве вычислительных машин экономнее включать в программу команды вычисления таких простых функций, как *car*, *cdr*, *eq* и т. п., чем команды обращения к соответствующим подпрограммам.

По-видимому, эти примеры достаточно полно иллюстрируют технику программирования встроенных функций, и в последующих разделах мы вновь вернемся к ранее использованному методу описания реализации этих функций.

2.13. Уборка мусора

Во время работы программы может случиться, что в одном из участков памяти не осталось свободного места. Чаще всего это случается с областью списочной памяти. Но именно в этой области наиболее вероятно наличие *мусора* — ячеек, не входящих ни в одну из списочных структур, доступных программе. Мусор может образовываться также в списке объектов и в области полных слов.

Избавиться от мусора, сделать занятые им ячейки вновь доступными программе — такова цель подпрограммы, называемой *мусорщиком*. Мусорщик работает в два этапа. На этапе *разметки* выявляются и помечаются все ячейки или группы ячеек, еще доступные программе. На этапе *сборки* все непомеченные ячейки (это и есть мусор) собираются в резерв свободной памяти.

Итак, общая структура мусорщика — подпрограммы *reclaim* — такова:

```

procedure reclaim;
begin mark; collect end

```

где *mark* и *collect* — подпрограммы разметки и сборки.

Указатели на списочные структуры и отдельные атомы, доступные программе, содержатся в магазине, в ассоциативном списке, в списках свойств атомов-констант и определяемых функций, а также атомов, относительно которых уже известно, что они не являются мусором, в некоторых рабочих ячейках подпрограмм встроенных функций и еще в одном массиве, который заполняется в ходе компиляции определяемых функций. В этот последний массив выносятся адреса выражений и списков, не подвергающихся компиляции и используемых в компилированных подпрограммах (см. разд. 2.14). Ниже будем называть этот массив *массивом внесенных* (из области полных слов) *указателей*.

Для каждого из перечисленных массивов производится разметка списочных структур, участков области полных слов и информационных ячеек атомов, адреса которых содержатся в данном массиве. Строение этих массивов различно, поэтому выделение адресов, с которых начинается разметка, происходит по-своему для каждого массива. Например, в магазине, кроме адресов выражений, могут встречаться адреса возвратов из подпрограмм (см. разд. 2.12), которые при разметке следует игнорировать. В ассоциативном списке начальными адресами для разметки могут служить только адреса, содержащиеся в *d*-указателях ячеек, составляющих этот список, за исключением, разумеется, адресов, служащих для образования мостиков (см. разд. 2.7). Подробнее алгоритмы выделения начальных адресов для разметки описывать не будем.

Рассмотрим процедуру разметки списочной структуры, заданной своим начальным адресом *a1*. Проще всего она описывается рекурсивно:

```

procedure marklist (a1);
  if  $\neg$  marked (a1) then
    begin markcell (a1);
      if  $\neg$  atom (a1) then
        begin marklist (car (a1));
          marklist (cdr (a1)) end
        end
    end

```

Здесь логическая процедура *marked* проверяет, не была ли ранее помечена ячейка *a1* (она может принадлежать сразу нескольким списочным структурам, или же к ней могут вести разные пути в одной структуре — см. рис. 2.1.8, а). Процедура *markcell* помечает одну ячейку с адресом *a1*. Далее, если это — не информационная ячейка атома, то следует пометить структуры, адреса которых содержатся в *a*- и *d*-указателях этой ячейки.

The diagram illustrates the construction of a B-tree. It shows a sequence of nodes (represented as boxes with pointers) being linked together. The final structure shows a root node (labeled β) pointing to several leaf nodes (labeled γ).

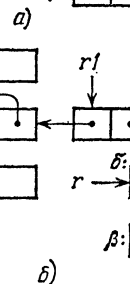


Рис. 2.13.1.

от начальной до текущей ячейки, и помнить лишь адрес непосредственно предыдущей ячейки на этом пути. Так, например, в момент, когда мы собираемся метить ячейку с адресом δ в структуре, изображенной на рис. 2.13.1, *а*, содержимое указателей должно быть преобразовано так, как это показано на рис. 2.13.1, *б*. Рабочие ячейки r и rl содержат соответственно адреса очередной помечаемой ячейки (т. е. адрес δ) и предыдущей ячейки. Когда мы будем метить ячейку β (или γ), то адрес, находящийся в rl , будет помещен в a -указатель (d -указатель) ячейки δ , в rl будет находиться адрес $\delta.a$ в r — адрес β (γ). Из этого примера видно, что для восстановления содержимого ячейки rl надо знать, в каком из указателей лисповской (списочной) ячейки хранится это содержимое.

Будем считать, что каждая лисповская ячейка, кроме a - и d -указателей, содержит еще два разряда: один из них — m -разряд — используется для пометки, другой — b -разряд — для фиксации ветви, по которой, исходя из данной ячейки, пошла дальнейшая разметка. Содержимое этих разрядов ячейки с адресом a будем обозначать $cm(a)$ и $cb(a)$, а для посылки в эти разряды двоичной цифры b будем пользоваться процедурами $rm(a, b)$ и $rb(a, b)$. Непомеченная (помеченная) ячейка содержит в m -разряде цифру 0 (1), цифра 0 (1) в b -разряде означает, что идет разметка подструктуры, адрес которой находился в a -указателе (d -указателе) данной ячейки. Таким образом, условие $marked(a)$ в этих обозначениях можно записать в виде $cm(a) = 1$, а оператор $markcell(a)$ — в виде $rm(a, 1)$.

Нерекурсивный алгоритм разметки списочной структуры с начальным адресом $a1$ описывается в виде следующей процедуры:

```

procedure marklist ( $a1$ );
begin  $r := a1$ ;  $r1 := nil$ ;
   $A$  : if  $cm(r) = 0$  then
    begin  $rm(r, 1)$ ;
      if  $\neg atom(r)$  then
        begin  $r2 := r1$ ;  $r1 := r$ ;  $r := car(r1)$ ;
           $rb(r1, 0)$ ;  $rplaca(r1, r2)$ ; go to  $A$  end
        end;
       $B$  : if  $\neg null(r1)$  then
        begin if  $cb(r1) = 0$  then
          begin  $r2 := car(r1)$ ;  $rplaca(r1, r)$ ;
             $rb(r1, 1)$ ;  $r := cdr(r1)$ ;
               $rplacd(r1, r2)$ ; go to  $A$  end;
             $r2 := cdr(r1)$ ;  $rplacd(r1, r)$ ;
               $rb(r1, 0)$ ;  $r := r1$ ;  $r1 := r2$ ;
                go to  $B$ 
            end
          end
        end
      end
    end
  end

```

На оператор с меткой A мы попадаем каждый раз, продвинувшись по a - или d -указателю. Если ячейка с адресом, находящимся в ячейке r , не помечена, то она метится, и если это — не информационная ячейка атома, то подготавливается и осуществляется продвижение по a -указателю. Переход к оператору с меткой B означает, что впереди уже метить нечего и надо возвращаться назад по списочной структуре. Если ячейка $r1$ содержит значение nil , присвоенное ей в самом начале, то возвращаться уже некуда и разметка структуры заканчивается. Если b -разряд ячейки с адресом из $r1$ содержит

нуль, то надо восстановить a -указатель этой ячейки, перенеся его содержимое в d -указатель, и продвинуться по прежнему адресу из d -указателя, сделав отметку в b -разряде. Если же в b -разряде уже находится 1, то надо восстановить d -указатель и вернуться к ячейке, адрес которой находится в этом указателе, заслав нуль в b -разряд.

На некоторых машинах размер ячейки не позволяет разместить в ней ничего, кроме a - и d -указателей. В этом случае m - и b -разряды размещаются в отдельном массиве. Это несколько усложняет и замедляет доступ к этим разрядам.

Процедура *marklist* отображает лишь алгоритм разметки структур в списочной памяти и информационных ячеек атомов. Несложно дополнить ее так, чтобы после пометки информационной ячейки размечался список свойств атома. Значения свойств (см. разд. 2.2) размещаются преимущественно в области списочной памяти и в области полных слов. На разметке последней мы остановимся чуть ниже.

Следует учесть также, что в указателях лисповских ячеек могут храниться адреса, не относящиеся к участкам памяти, подлежащим разметке. Поэтому перед проверкой условия $cm(r) = 0$ следует проверить, что адрес в ячейке r относится к этим участкам. В противном случае следует также переходить на метку B , т. е. возвращаться по размечаемой структуре.

Довольно прост алгоритм разметки списка объектов. Его задача — собрать в списках, подвешенных к ячейкам оглавления (см. разд. 2.5), лишь те звенья, которым соответствуют помеченные информационные ячейки. Эти звенья следует пометить и объединить в новый список. Разметку списка объектов следует производить лишь тогда, когда уже помечены информационные ячейки всех доступных атомов. Разметка описывается в виде следующей процедуры. В ней обозначено: h — базовый адрес оглавления списка объектов, m — длина оглавления.

procedure *markoblist*;

begin $i := h$;

$A: r := r1 := i$;

$B: r := cdr(r)$;

if $\neg null(r)$ **then**

begin $r2 := car(r)$;

if $cm(r2) = 1$ **then**

begin $rm(r, 1); rplacd(r1, r)$;

$r1 := r$ **end**;

go to B **end**;

```

    rplacd (r1, nil); i := i + 1;
    if i < h + m then go to A
end

```

Процедура *markoblist* рассчитана на структуру списка объектов, изображенную на рис. 2.5.1. С метки *A* начинается цикл просмотра ячеек оглавления списка объектов, с метки *B* — внутренний цикл, в котором анализируется список, подвешенный к одной из ячеек оглавления. Указатель *r* содержит адрес очередного звена этого списка, указатель *r1* — адрес последнего из просмотренных звеньев, которое должно быть сохранено, так как его *a*-указатель содержит адрес ранее помеченной информационной ячейки атома. Каждое такое звено метится и подвешивается либо к ячейке оглавления (если $r1 = i$), либо к предыдущему помеченному звену. Таким образом, список объектов не только размечается, но и перестраивается — ненужные звенья из него выбрасываются.

Разметку области полных слов желательно свести к пометке первой ячейки каждого из участков, занятых в этой области и еще доступных программе. Это можно делать по ходу разметки списков свойств помечаемых атомов, так как каждый участок представляет собой одно из свойств некоторого атома. Но если участок может содержать указатели на другие списочные структуры, используемые в программе, то эти структуры также следует разметить. Обнаружить такие указатели в области полных слов нелегко, а скорее всего — невозможно, так как содержимое участков области полных слов может оказаться произвольным. Поэтому целесообразно при первоначальном заполнении области полных слов выносить все такие указатели в упомянутый выше массив. Такие указатели должны содержать, в частности, адреса аргументов обращений к функции *QUOTE* из тел компилируемых функций, а также некоторые другие адреса (см. ниже разд. 2.14). При компиляции эти адреса заносятся в упомянутый массив, причем вместе с каждым из них запоминается адрес начала компилированной подпрограммы в области полных слов и положение вынесенного указателя относительно этого начала. Выражение, адрес которого хранится в этом массиве, размечается лишь в том случае, если соответствующая подпрограмма помечена, т. е. если компилированная функция доступна программе.

Переходим ко второму этапу — к собственно сборке мусора. Он был бы очень прост, если бы освободившиеся участки памяти (бывший мусор) можно было оставлять там, где они обнаружились. В области списочной памяти это вполне приемлемо, но в области полных слов могут возникать осложнения при попытке вновь занять свободный участок, если длина его мала. Поэтому мы сразу рассмот-

рим вариант, когда сборка мусора совмещается с уплотнением занятой памяти. Обычно области списочной памяти и полных слов размещаются в одном массиве, соответственно в его конце и в начале. Середина этого массива — резерв свободной памяти для наращивания обеих областей.

Будем считать, что первая ячейка каждого участка, выделенного в области полных слов, играет роль паспорта участка. Паспорт содержит m -разряд, используемый для пометки, a -указатель, который обычно содержит адрес паспорта, и еще одну группу разрядов, которую мы назовем n -полем, предназначенную для хранения длины участка. Для обращения к содержимому m -разряда и a -указателя ячейки с адресом a по-прежнему будем использовать функции $cm(a)$ и $car(a)$, а для изменения этого содержимого — процедуры $rm(a, b)$ и $rplaca(a, x)$. Содержимое n -поля будем обозначать $cn(a)$.

Опишем теперь важнейшие части процедуры *collect*. Для уплотнения памяти некоторые ячейки списочной памяти и некоторые участки области полных слов приходится переносить на новое место. В связи с этим сборка мусора должна выполняться в следующей последовательности: вычисление новых начальных адресов помеченных участков и ячеек, замена старых адресов на новые во всех указателях и перемещение участков по новым адресам. В списочной памяти перемещение содержимого помеченных ячеек можно осуществлять сразу, как только выбрано их новое место (в одной из непомеченных ячеек), так как длины всех ячеек одинаковы. В области полных слов необходимо строго придерживаться указанной последовательности фаз сборки мусора. Итак, общая структура процедуры *collect* такова:

```

procedure collect;
begin calculate addresses;
      reallocate; correct;
      move end

```

В процедуре *calculate addresses*, где определяются новые начальные адреса помеченных участков области полных слов, использованы обозначения: $mbeg$ — начальный адрес области, $fwppt$ — указатель на первую ячейку вслед за концом области, т. е. на начало свободной памяти, r — адрес очередного участка, $r1$ — новый адрес.

```

procedure calculate addresses;
begin  $r := r1 := mbeg$ ;
      A: if  $r < fwppt$  then
        begin if  $cm(r) = 1$  then
          begin  $rplaca(r, r1); rm(r, 0)$ ;
             $r1 := r1 + cn(r)$  end;
        end

```

```

     $r := r + cn(r)$ ; go to  $A$  end;
     $fwpt0 := fwpt$ ;  $fwpt := r1$ 

```

end

Новые адреса запоминаются в a -указателях паспортов участков. Оператор $rm(r, 0)$ снимает пометку с участка. После просмотра всех участков определяется новое значение $fwpt$. Старое значение запоминается.

Процедура *reallocate* просматривает область списочной памяти с двух концов. Сверху (со стороны меньших адресов) разыскивается очередная помеченная ячейка, а снизу — очередная непомеченная. Как только будут найдены помеченная ячейка с адресом r и непомеченная ячейка с адресом $r1$, причем $r < r1$, содержимое помеченной ячейки переносится по адресу $r1$, а в a -указателе ячейки r запоминается этот новый адрес. Обозначения: *memory* — массив, в котором, как говорилось выше, располагаются область полных слов и область списочной памяти, *mend* — адрес последней ячейки этого массива, т. е. адрес конца списочной памяти, *lptr* — указатель на начало (первую занятую ячейку) списочной памяти.

```

procedure reallocate;
  begin  $r := lptr$ ;  $r1 := mend$ ;
     $A$ : if  $r \leq r1$  then
      begin if  $cm(r) = 0$  then  $r := r + 1$  else
        begin if  $cm(r1) = 0$  then
          begin  $memory[r1] := memory[r]$ ;
             $rplaca(r, r1)$ ;  $rm(r, 0)$ ;
             $r := r + 1$  end;
           $r1 := r1 - 1$  end;
        go to  $A$  end;
       $lptr := r$ 
    end

```

Заметим, что процедура *reallocate* оставляет на прежнем месте все помеченные ячейки, адреса которых заключены между новым значением *lptr* и *mend*. Поэтому при работе процедуры *correct* следует изменять значение указателя лишь в том случае, если оно не меньше *mbeg* и строго меньше значения *lptr*. Указатели, подлежащие корректировке, располагаются в ячейках списочной памяти, в магазине, в ассоциативном списке, в массиве указателей, вынесенных из области полных слов, и в рабочих ячейках системы. Так как корректировка всюду выполняется однотипно, опишем ее применительно лишь к списочной памяти.

```

procedure correct;
  for  $r := lptr$  step 1 until mend do
    begin if  $mbeg \leq car(r) \wedge car(r) < lptr$  then
       $rplaca(r, car(car(r)))$ ;
    if  $mbeg \leq cdr(r) \wedge cdr(r) < lprr$  then
       $rplacd(r, car(cdr(r)))$ ;
     $rm(r, 0)$  end

```

Последняя процедура мусорщика *move* сдвигает на свои места участки полных слов.

```

procedure move;
begin  $r := mbeg$ ;
   $A : \text{if } r < fwpt0 \text{ then}$ 
    begin  $r1 := car(r)$ ;  $n := cn(r)$ ;
      if  $r1 \neq r$  then
        for  $i := 0$  step 1 until  $n - 1$  do
           $memory[r1 + i] := memory[r + i]$ ;
         $r := r + n$ ; go to  $A$  end;
    end

```

Сдвигать нужно не все помеченные участки, а только те, у которых новый начальный адрес отличается от старого. Поэтому пометка снижалась с участков уже во время вычисления новых адресов.

2.14. Компилятор

Компиляция лисповских функций, т. е. перевод их определяющих выражений в машинные подпрограммы, значительно ускоряет вычисление их значений. Здесь трудно привести подробное описание компилятора, потому что компилятор — это довольно большая программа, а ее детали сильно зависят от машинного языка, на который происходит перевод. Поэтому более или менее подробно будет описана лишь структура машинных подпрограмм, возникающих в результате компиляции (обычно называемых *рабочими* подпрограммами), и их составных частей, соответствующих частям определяющих выражений. Две главные части определяющего выражения

$$(LAMBDA (x_1 \dots x_n) e) \quad (1)$$

— это список связанных переменных $(x_1 \dots x_n)$ и тело e . Для работы с переменными x_1, \dots, x_n используется несколько видоизмененный механизм ассоциативного списка (разд. 2.7). Перед началом выполнения рабочей (компилированной) подпрограммы значения, которые должны быть связаны с переменными x_1, \dots, x_n (т. е. значения v_1, \dots, v_n аргументов обращения, если функция — обычная, или сами

аргументы a_1, \dots, a_n , если функция — специальная), размещаются в d -указателях первых n свободных ячеек ассоциативного списка, т. е. там же, куда их помещает интерпретатор. В a -указатели этих ячеек, так же как в режиме интерпретации, заносятся наименования переменных, точнее — представляющие их адреса информационных ячеек атомов. Но значение, связанное с i -й переменной x_i , как правило, разыскивается не по ее наименованию, а по номеру i . Чтобы это было возможно, используется специальная переменная $base$, которой присваивается значение $aptr$ указателя ассоциативного списка непосредственно перед размещением переменных. При этом значение переменной x_i попадает в d -указатель ячейки ассоциативного списка с адресом $base + c_i$, где c_i линейно зависит от i , так что доступ к этому значению возможен с помощью операции $cdr (base + c_i)$, а точнее — с помощью соответствующих ей машинных команд. Здесь i может лежать в границах от 1 до n . В дальнейшем будем считать, как мы это уже делали в разд. 2.7, что $c_i = i - 1$, хотя на самом деле коэффициент зависимости адреса ячейки от ее порядкового номера может быть и не равен 1. Если в теле e рассматриваемого определяющего выражения (1) содержатся новые списки переменных, то им ставятся в соответствие очередные ячейки ассоциативного списка и очередные номера: $n + 1, n + 2, \dots$. Например, если это тело имеет вид

$$(PROG (u_1 \dots u_k) s_1 \dots s_r),$$

то программная переменная u_j получает номер $n + j - 1$, соответственно вычисляется ее адрес. В операторах s_1, \dots, s_r могут содержаться новые определяющие выражения или обращения к $PROG$. С их списками переменных мы поступаем таким же образом. Точнее, если для определяющих выражений или обращений к $PROG$, охватывающих очередное выражение одного из этих типов, использованы номера с 1 до n_1 , то переменным u_1, \dots, u_{m_1} этого последнего выражения ставятся в соответствие номера $n_1 + 1, \dots, n_1 + m_1$. В дальнейшем относительный адрес ячейки, поставленной в соответствие переменной x , будем обозначать c_x , абсолютный адрес равен при этом $base + c_x$. Для выражений, не содержащихся одно в другом, номера ячеек, соответствующих переменным этих выражений, могут совпадать (частично или полностью). Например, в выражении

$$(COND ((...) (PROG (u_1 \dots) ...)) \\ ((...) (PROG (v_1 \dots) ...)) ...)$$

переменным u_1 и v_1 будет соответствовать одна и та же ячейка. Во время компиляции номера, поставленные в соответствие переменным, можно хранить в ассоциативном списке так, как во время выпол-

нения программы хранятся значения переменных. Заполнение ассоциативного списка происходит во время обработки списка переменных, удаление этих переменных из ассоциативного списка — в момент завершения компиляции выражения, содержащего список переменных.

Все переменные, о которых шла речь, назовем *внутренними* переменными данного определяющего выражения (1) и рабочей подпрограммы, получающейся при его компиляции. Остальные переменные, встречающиеся в теле e выражения (1), назовем *внешними*. Более строго, вхождение переменной в тело e следует называть *внутренним* или *внешним* в зависимости от того, содержится ли оно или не содержится в одном из списков переменных в определяющих выражениях и обращениях к *PROG*, охватывающих данное вхождение и содержащихся в компилируемом выражении (1).

Теперь опишем структуру участка рабочей подпрограммы, соответствующего телу e компилируемого определяющего выражения. Впрочем, эта структура подчиняется одним и тем же правилам как для самого тела, так и для большинства выражений, входящих в его состав. Поэтому всякий раз, когда будет встречаться оборот «подпрограмма выражения», имеется в виду, что эта подпрограмма составляется по тем же правилам, что и для тела e . Структура этих подпрограмм будет описана на том же алголоподобном языке, которым мы пользовались в предыдущих разделах, но, конечно, вместо операторов и выражений этого языка всегда следует иметь в виду соответствующие конструкции машинного языка.

Пусть очередное компилируемое выражение — атом (для тела e , в отличие от его внутренних выражений, эта возможность чисто гипотетическая). Если этот атом a — внутренняя переменная, то соответствующая подпрограмма такова:

$$r := cdr (base + c_a)$$

Здесь c_a — относительный адрес, соответствующий данной переменной, r — переменная (ячейка), которой присваивается значение каждого очередного выражения. По окончании выполнения всей подпрограммы, соответствующей телу e , в ячейке r окажется значение функции для данного обращения к ней.

Любому другому атому (константе или внешней переменной) соответствует подпрограмма

$$r := evalom (a)$$

где a — адрес информационной ячейки атома, а подпрограмма *evalom* описана в разд. 2.8.

Если компилируемое выражение — не атом, то оно представляет собой обращение к функции

$$(fn\ a_1 \dots a_n) \quad (2)$$

Здесь fn может быть атомом-наименованием функции или определяющим выражением.

Пусть fn — атом. Структура рабочей подпрограммы зависит от класса функции с наименованием fn . Кроме четырех известных нам классов, следует ввести еще два: $COMP$ и $FCOMP$ для функций, полученных компиляцией определяющих выражений функций классов $EXPR$ и $FEXPR$ соответственно. Следует считаться с тем, что во время компиляции выражения (2) сама функция fn может быть еще не компилирована, но окажется прокомпилированной к моменту выполнения рабочей подпрограммы для этого выражения. Поэтому структуры рабочих подпрограмм для функций классов $EXPR$ и $COMP$ (а также для пары классов $FEXPR$ и $FCOMP$) должны быть схожими и первая должна легко преобразовываться во вторую, если обнаружится, что к моменту выполнения рабочей подпрограммы функция fn также подверглась компиляции. Это соображение было одной из причин, по которой мы приняли описанную выше схему расположения аргументов компилированных функций в ассоциативном списке.

Рабочая подпрограмма, соответствующая выражению (2), состоит из двух частей: подготовительной, обрабатывающей аргументы a_1, \dots, a_n , и исполнительной, вычисляющей собственно значение функции. Подготовительные части для функций классов $EXPR$ и $COMP$ одинаковы и имеют вид

```
подпрограмма выражения  $a_1$ ;  
   $rplacd\ (aptr, r); aptr := aptr + 1$ ;  
  .....  
подпрограмма выражения  $a_n$ ;  
   $rplacd\ (aptr, r); aptr := aptr + 1$ 
```

Что такое «подпрограмма выражения a_i », было объяснено выше. Два оператора, следующие за каждой такой подпрограммой, заносят значение аргумента a_i , вычисленное в ячейке r , в d -указатель очередной ячейки ассоциативного списка и продвигают этот указатель по списку.

Исполнительная часть рабочей подпрограммы для функции класса $COMP$ состоит из оператора

$$evcomp\ (fn, n) \quad (3)$$

обращающегося к подпрограмме

```
procedure evcomp (fn, n); procedure fn; integer n;  
begin save (base); base := aptr - n;  
  pairlist (base, car (fn)); call (cadr (fn));  
  aptr := base; restore (base) end
```

Подпрограмма *evcomp* запоминает текущее значение переменной *base* и присваивает ей значение, равное значению указателя *aptr* перед началом выполнения подготовительной части. Подпрограмма

```
procedure pairlist (a, a1);  
begin r := a; r1 := a1;  
  A: if  $\neg$  null (r1) then  
    begin rplaca (r, car (r1)); r := r + 1;  
      r1 := cdr (r1); go to A end  
end
```

заносят адреса информационных ячеек связанных переменных функции *fn* в *a*-указатели последовательных ячеек ассоциативного списка. Адрес первой из этих ячеек задается первым аргументом при обращении к *pairlist*, адрес списка переменных — вторым аргументом. Предполагается, что после компиляции определяющего выражения (1) в информационную ячейку наименования функции заносится адрес *del* этого выражения, преобразованного к виду

$$((x_1 \dots x_n) a_e)$$

где a_e — адрес начала рабочей подпрограммы тела определяющего выражения в области полных слов. Оператор *call* (*f*) означает обращение (переход с возвратом) к подпрограмме с начальным адресом *f*. В данном случае это — только что упомянутый адрес a_e . Два последних оператора восстанавливают значение *aptr* и *base*.

Исполнительная часть рабочей подпрограммы для функции класса *EXPR* имеет вид

$$evexpr (fn, n) \tag{4}$$

Подпрограмма *evexpr* прежде всего проверяет (по признаку в информационной ячейке атома *fn*), не подверглась ли компиляции функция *fn*. Если да, то обращение (4) в рабочей подпрограмме заменяется обращением (3), которое тут же выполняется. Если нет, то выполняются операции

```
r := aptr - n; save (r); pairlist (r, car (fn));  
eval (cadr (fn)); restore (aptr)
```

Для функций классов *FCOMP* и *FEXPR* исполнительные части имеют такой же вид, как для *COMP* и *EXPR* соответственно.

Подготовительная же часть состоит лишь из одного оператора

pairlis2 (*args*)

где *args* — адрес списка ($a_1 \dots a_n$) аргументов обращения, а подпрограмме *pairlis2* соответствует описание

```
procedure pairlis2 (a);  
begin r := a;  
  A: if  $\neg$  null (r) then  
    begin rplacd (aptr, car (r)); r := cdr (r);  
        aptr := aptr + 1; go to A end  
end
```

Адрес *args* должен храниться в массиве указателей, вынесенных из области полных слов (разд. 2.13).

Если в выражении (2) вместо *fn* стоит определяющее выражение

(*LAMBDA* ($y_1 \dots y_n$) *e*)

то подготовительная часть рабочей подпрограммы остается такой же, как для функций *fn* класса *COMP* или *EXPR*, а исполнительная часть принимает вид

```
pairlis1 (aptr — n, vars);  
подпрограмма выражения e;  
aptr := aptr — n
```

Здесь *vars* — адрес списка переменных ($y_1 \dots y_n$), хранящийся в массиве вынесенных указателей. В подпрограмме выражения *e* этим переменным ставятся в соответствие очередные ячейки, нумеруемые относительно текущего значения переменной *base*, как было описано в начале раздела.

Осталось рассмотреть случаи, когда *fn* — наименование встроенной функции. Стандартные приемы компиляции обращений к таким функциям очень просты. Так, обращение к функции *fn* класса *SUBR* без параметров переводится в оператор

call (*fn*)

с одним параметром a_1 — в участок программы

```
подпрограмма выражения  $a_1$ ;  
 $a_1 := r$ ; call (fn)
```

и с двумя параметрами a_1 и a_2 — в участок программы

```
подпрограмма выражения  $a_1$ ;  
save (r);
```

подпрограмма выражения a_2 ;
 $restore(a1); a2 := r; call(fn)$

Здесь $a1$ и $a2$ — стандартные ячейки для аргументов функций класса *SUBR*.

Обращению (2) к любой функции fn класса *FSUBR* соответствуют операторы

$a1 := args; call(fn)$

где $args$ имеет тот же смысл, что и выше.

Однако в ряде случаев вместо обращений к подпрограммам встроенных функций целесообразно включать в рабочую программу операторы, реализующие эти функции. Так, выражение (*CAR a1*) проще перевести в операторы

подпрограмма выражения a_1 ;
 $r := car(r)$

чем компилировать стандартным образом, когда вместо последнего оператора стоят операторы

$a1 := r; call(car)$

(напомним, что в данном разделе оператор $r := car(r)$ обозначает команды, помещающие в ячейку r содержимое a -указателя ячейки, адрес которой ранее находился в r , а $call(car)$ — команды обращения к подпрограмме функции car). Правда, по числу команд стандартный способ может оказаться более экономным (на машинах с примитивными средствами косвенной адресации), но компиляция вообще предпринимается с целью экономии не команд, а времени работы программы. Нестандартный способ выгоден также для таких функций, как *CDR*, *ATOM*, *EQ*, *NULL*, *RPLACA*, *RPLACD*.

Но особое внимание следует уделить нестандартным способам компиляции функций класса *FSUBR*, в которых главным образом и воплощена специфика языка.

Выражению (*QUOTE e*) соответствует оператор

$r := e$

причем адрес e должен быть помещен в массив вынесенных указателей.

Выражение

$$(COND (p_1 e_{11} \dots e_{n_{11}}) \dots (p_m e_{1m} \dots e_{n_{mm}})) \quad (5)$$

переводится в следующую последовательность операторов:

```

begin  $L_j$ : подпрограмма выражения  $p_1$ ;
      if null ( $r$ ) then go to  $L_1$ ;
      подпрограмма выражения  $e_{11}$ ;
      . . . . .
      подпрограмма выражения  $e_{n_11}$ ;
      go to  $L_m$ ;
      . . . . .
 $L_{i-1}$ : подпрограмма выражения  $p_i$ ;
      if null ( $r$ ) then go to  $L_i$ ;
      подпрограмма выражения  $e_{1i}$ ;
      . . . . .
      подпрограмма выражения  $e_{n_i i}$ ;
      go to  $L_m$ ;
      . . . . .
 $L_{m-1}$ : подпрограмма выражения  $p_m$ ;
      if null ( $r$ ) then go to  $L_m$ ;
      подпрограмма выражения  $e_{1m}$ ;
      . . . . .
      подпрограмма выражения  $e_{n_m m}$ ;
 $L_m$ : end

```

Однако, если выражение (5) представляет собой тело e компилируемого определяющего выражения (1) функции f класса $EXPR$, а какое-нибудь из выражений $e_{n_i i}$ ($i = 1, \dots, m$) является обращением к этой же функции

$(f \ b_1 \dots b_n)$

то вместо подпрограммы выражения $e_{n_i i}$ (рекурсивного обращения к f) целесообразно поместить в рабочую программу операторы

```

подпрограмма выражения  $b_1$ ;
rplacd ( $base + n, r$ );  $aptr := aptr + 1$ ;
. . . . .
подпрограмма выражения  $b_{n-1}$ ;
rplacd ( $base + 2 \times n - 2, r$ );  $aptr := aptr + 1$ ;
подпрограмма выражения  $b_n$ ;
rplacd ( $base + n - 1, r$ );  $aptr := aptr - n + 1$ ;
for  $i := 1$  step 1 until  $n - 1$  do
  rplacd ( $base + i - 1, cdr (base + n + i - 1)$ );
go to  $L_0$ 

```

Эти операторы вычисляют новые значения аргументов обращения b_1, \dots, b_n , помещая их сначала (кроме последнего) в очередные свободные ячейки ассоциативного списка, чтобы не портить до завершения вычислений предыдущие значения, а затем заменяют эти предыдущие значения новыми и передают управление на начало рабочей подпрограммы функции f . Этот прием позволяет несколько сократить время вычислений, а главное, — существенно уменьшить количество памяти, занимаемой в ассоциативном списке.

Еще один прием повышения эффективности рабочей подпрограммы для выражения (5) связан с появлением логических функций $NULL$ (или NOT), AND и OR в условиях, т. е. на месте выражений p_i . Вместо

подпрограмма выражения p ;
if null (r) then go to L

следует включать в рабочую программу следующие операторы:

- а) если p имеет вид ($NULL\ q_1$), то
 подпрограмма выражения q_1 ;
if \neg null (r) then go to L
- б) если p имеет вид ($AND\ q_1 \dots q_k$), то
 подпрограмма выражения q_1 ;
if null (r) then go to L;

 подпрограмма выражения q_k ;
if null (r) then go to L
- в) если p имеет вид ($OR\ q_1 \dots q_k$), то
begin подпрограмма выражения q_1 ;
if \neg null (r) then go to L'_1 ;

 подпрограмма выражения q_{k-1} ;
if \neg null (r) then go to L'_1 ;
 подпрограмма выражения q_k ;
if null (r) then go to L;
 L'_1 : end

Аналогично, вместо операторов

подпрограмма выражения p ;
if \neg null (r) then go to L

в рабочую программу следует помещать операторы:

- а) если p имеет вид ($NULL\ q_1$), то
 подпрограмма выражения q_1 ;
if null (r) then go to L

- б) если p имеет вид $(AND\ q_1 \dots q_k)$, то
- ```

begin
 подпрограмма выражения q_1 ;
 if null (r) then go to L'_1 ;

 подпрограмма выражения q_{k-1} ;
 if null (r) then go to L'_1 ;
 подпрограмма выражения q_k ;
 if \neg null (r) then go to L ;
 L'_1 : end

```
- в) если  $p$  имеет вид  $(OR\ q_1 \dots q_k)$ , то
- ```

  подпрограмма выражения  $q_1$ ;
  if  $\neg$  null ( $r$ ) then go to  $L$ ;
  . . . . .
  подпрограмма выражения  $q_k$ ;
  if  $\neg$  null ( $r$ ) then go to  $L$ 

```

Ясно, что если одно из выражений q_j также является обращением к функции *NULL*, *AND* или *OR*, то эти правила должны применяться повторно.

Выражению $(PROG\ (u_1 \dots u_k)\ s_1 \dots s_r)$ соответствует рабочая подпрограмма

```

begin pairlis3 (vars);
  подпрограмма оператора  $s_1$ ;
  . . . . .
  подпрограмма оператора  $s_r$ ;
   $r := nil$ ;
EX:  $aptr := aptr - k$  end

```

Оператор $r := nil$ необходим лишь в том случае, если оператор s_r не является оператором *GO* или *RETURN*. Он вырабатывает значение *NIL* функции *PROG*, если не было выработано какое-либо иное значение. Метка *EX* используется в подпрограммах операторов *RETURN*. Следующий за ней оператор восстанавливает исходное значение указателя *aptr*, измененное в результате работы подпрограммы *pairlis3*. Эта подпрограмма

```

procedure pairlis3 (a);
begin  $r := a$ ;
  A: if  $\neg$  null ( $r$ ) then
    begin  $rplaca\ (aptr, car\ (r))$ ;
       $rplacd\ (aptr, nil)$ ;  $r := cdr\ (r)$ ;
       $aptr := aptr + 1$ ; go to A end
end

```


связывает в ассоциативном списке программные переменные со значением *NIL*. Адрес списка программных переменных *vars* должен храниться в массиве вынесенных указателей.

Подпрограммы операторов, специфических для аппарата *PROG*, таковы. Для оператора (*SETQ u e*), где *u* — внутренняя переменная:

подпрограмма выражения *e*;

rplacd (base + c_u, r)

где *c_u* — относительный адрес ячейки, поставленной в соответствие переменной *u*. Если же *u* — внешняя переменная, то оператору (*SETQ u e*) соответствуют операторы

подпрограмма выражения *e*;

r1 := r; r := assoc (u); rplacd (r, r1)

где *assoc* — функция, описанная в разд. 2.7, а адрес (информационной ячейки) *u* хранится в массиве вынесенных указателей. Аналогично строятся подпрограммы операторов *PUSH*, *POPUP* и *POP*, описанных в разд. 3.9.

Подпрограмма оператора (*RETURN e*) имеет вид

подпрограмма выражения *e*;

go to EX

Меткам соответствуют начальные адреса операторов рабочей подпрограммы, реализующих те операторы *PROG*a, перед которыми стоят эти метки. Оператор (*GO l_i*) переводится в

go to A_i

где *A_i* — адрес, соответствующий метке *l_i*.

Перейдем к компиляции функциональных аргументов. Выражение (*FUNCTION (LAMBDA (y₁ ... y_n) e)*) переводится в последовательность операторов:

go to L1;

L_e: подпрограмма выражения *e*;

L1: r := cons (funarg1, cons (base, cons (aptr, del)))

где *del* — хранящийся в массиве вынесенных указателей адрес преобразованного определяющего выражения

((NIL y₁ ... y_n) L_e)

Таким образом, соответствующая функциональная переменная получает значение

(FUNARG1 b₀ ass₀ (NIL y₁ ... y_n) L_e)

где b_0 и ass_0 — значения указателей $base$ и $aptr$ в момент задания функционального аргумента. Адрес nil добавляется к началу списка переменных, чтобы при обращении к функциональному аргументу в ассоциативном списке возник мостик (см. разд. 2.10). Подпрограмма выражения e должна составляться с учетом того, что перед ячейками, соответствующими переменным y_1, \dots, y_n , в ассоциативном списке под мостик будет занята еще одна ячейка, и предусматривать возврат к месту обращения к ней.

К уже рассмотренным случаям компиляции выражения (2) надо добавить случай, когда fn — атом, не наделенный признаком функции или константы, т. е. функциональная переменная. При этом подготовительная часть остается почти такой же, как для функций классов $COMP$ или $EXPR$, с тем отличием, что в самом начале ее добавляется оператор

$$aptr := aptr + 1$$

резервирующий место под мостик. Исполнительная часть рабочей подпрограммы состоит из операторов:

подпрограмма атома fn ;
 $eval(r, n + 1)$

Подпрограмму $eval$ мы детально не описываем, так как она довольно громоздка, но основана на уже известных принципах. Анализируя значение своего первого аргумента, она должна создать мостик (возможно, тривиальный — на предыдущую ячейку ассоциативного списка), занести в ассоциативный список наименования связанных переменных функционального аргумента, выполнить его тело (с помощью $call$ или $eval$, а может быть, — $evcomp$ или $evexpr$, если функциональный аргумент был задан не определяющим выражением, а наименованием функции) и восстановить значения указателей $aptr$ и $base$, если они были испорчены.

Обращения к не рассмотренным здесь функциям классов $SUBR$ и $FSUBR$ целесообразно компилировать стандартным образом, так как для них нестандартная компиляция (ее легко осуществить по приведенным здесь образцам) мало что дает.

2.15. Ограничения

При интерпретации анализ структуры выражения и свойств его составных частей происходит параллельно с вычислением значения выражения. При компиляции эти процессы разделены во времени. Так как свойства объектов программы, прежде всего — атомов, могут меняться во время ее выполнения, то, строго говоря, безошибочная компиляция почти невозможна или, во всяком случае,

требует, чтобы рабочая подпрограмма предусматривала проверку динамических свойств объектов. В описанной выше схеме компиляции такая проверка предусмотрена лишь в одном случае — при обращении к функции класса *EXPR* (*FEXPR*), которая может к моменту выполнения этого обращения перейти в класс *COMP* (*FCOMP*). Это изменение свойств связано с самим процессом компиляции, и пренебрегать им нельзя. Например, если две функции обращаются одна к другой (а это совсем не редкость), то во время компиляции определяющего выражения первой (по порядку компиляции) из этих функций вторая еще сохраняет свой прежний класс. Можно, правда, потребовать, чтобы при обращении к компилятору задавался список всех компилируемых функций, и при компиляции учитывать не только текущий класс функции, но и присутствие ее наименования в этом списке. Это хотя и не очень существенное, но все же ограничение на входной язык (в данном случае — в правилах обращения к компилятору).

Но есть и другие ситуации, когда неограниченное использование всех возможностей входного языка, вполне доступных в режиме интерпретации, приводит к неверной работе компилированных программ. Например, интерпретатор разрешает замену одних определений функций другими (при условии, что число и смысл аргументов остаются прежними). Компилятор, безусловно, не допускает замены определения функции, связанной с изменением ее класса (например, *COMP* на *EXPR*, *EXPR* на *FEXPR* или *SUBR* на *EXPR*), кроме оговоренного выше случая (замены *EXPR* на *COMP*). Однако замена, скажем, прокомпилированного определения класса *COMP* на новое определение класса *EXPR* с последующей компиляцией (при сохранении числа и смысла аргументов) возможна. Чувствителен компилятор и к изменению свойств атомов — переменных и констант. Компилятор считает константами лишь те атомы, которые были наделены свойством *APVAL* во время компиляции. Если внутренняя переменная к моменту выполнения программы стала константой, то компилированная программа в отличие от интерпретатора этого не заметит. Если же, наоборот, атом-константа лишится свойства *APVAL*, то его значение будет найдено хотя и правильно (т. е. так же, как при интерпретации), но медленнее, чем значение обычной внутренней переменной, так как будет разыскиваться по наименованию, а не по относительному адресу.

Для интерпретатора мы запретили использование функциональных аргументов вида (*FUNCTION fn*), где *fn* — функция класса *FEXPR*. Для компилятора *fn* может принадлежать только классам *COMP* или *EXPR*, так как функциональная переменная *fv* не несет признака класса соответствующего ей функционального аргумента

и при компиляции подготовительной части обращения к функции вида $(fv\ e_1 \dots e_n)$ приходится ориентироваться на наиболее распространенный класс.

Изложенная выше схема компиляции допускает почти неограниченное совместное использование компилированных и интерпретируемых определений функций. Однако и здесь некоторые возможности режима интерпретации недоступны компилированным программам. Например, оператор

$(EVAL\ (LIST\ (QUOTE\ GO)\ e))$ (1)

— для перехода к метке, являющейся значением выражения e , не будет переведен правильно. Возможно, конечно, несколько усложнить способ компиляции функции *PROG* и оператора *GO* (в частности, использовать переменную *llis*, присваивая ей в начале выполнения рабочей подпрограммы обращения к функции *PROG* список, составленный из меток *PROG*а в паре с соответствующими им адресами перехода). Но, по-видимому, отказ от использования подобных конструкций является разумной платой за преимущества, даваемые компиляцией.

Более того, для многих применений языка можно вообще отказаться от интерпретации, потребовав, чтобы все определения функций компилировались и чтобы они не содержали внешних переменных. Вместо последних можно использовать константы и константы-магазины (значение константы-магазина — это список, первый элемент которого соответствует активному, а остальные — пассивным значениям переменной). Тогда отпадает надобность в поиске значений переменных по их наименованиям, а вместе с ним — в *a*-указателях ячеек ассоциативного списка, в хранении списков наименований переменных, в обращениях к *pairlist*, в мостиках и т. п. Ассоциативный список превращается в обычный магазин значений внутренних переменных. Рабочие подпрограммы становятся короче и работают быстрее.

Хотя такая реализация и приемлема для многих приложений, ее все же нельзя считать удовлетворительной. Во-первых, константы и даже константы-магазины не являются полным эквивалентом внешних переменных, так как простые константы не имеют пассивных значений, а все операции над константами-магазинами программист должен выписывать явно. Аналогичные операции над ассоциативным списком, где хранятся значения внешних переменных, выполняются автоматически как при интерпретации, так и при компиляции по описанной выше схеме. В то же время вообще отказаться от внешних (иначе говоря, глобальных) переменных тоже нежелательно, как показывает следующий простой пример.

Допустим, функция F обращается к функциям G и H , у которых есть сходные по своему назначению переменные. Над этими переменными при выполнении функций G и H требуется произвести одни и те же действия, которые могут быть описаны с помощью функции K . Если требуется, чтобы при этих действиях значения переменных не только использовались, но и менялись, то они могут быть только внешними в K . При этом в G и H они должны быть обозначены одинаково, так что при компиляции функции K даже нельзя установить, переменные какой из двух функций участвуют в операциях. Можно, правда, сделать их внутренними (локальными) в F , но тогда они окажутся внешними не только в K , но и в G и H . Если каждый атом, выступающий в роли внешней переменной в какой-нибудь функции, является внутренней переменной в точности одной функции, то в принципе во время компиляции можно реализовать механизм, определяющий адрес хранения значения этой переменной, но едва ли этот механизм будет проще ассоциативного списка. Кроме того, соблюдение указанного условия потребует от программиста дополнительных усилий и искусственных приемов.

Вторая причина, по которой желательно сохранить аппарат интерпретации, — это создаваемая этим аппаратом возможность формировать новые части программы в процессе ее выполнения, как это показано выше на примере (1) формирования оператора GO . Хотя злоупотреблять этой возможностью и не рекомендуется (она делает действия, выполняемые программой, трудно обозримыми и плохо контролируемыми), но все же она выделяет лисп среди большинства языков программирования и в некоторых случаях очень удобна.

Краткий вывод из сказанного в этом разделе таков. Программы можно сделать более быстро работающими или экономными в других отношениях, но почти неизбежно это связано с ограничениями входного языка. И наоборот, дополнительные удобства для программиста связаны, как правило, с потерей эффективности программ. К этому можно добавить, что идейная и структурная простота лиспа делает его чрезвычайно удобным объектом для экспериментов по методам реализации. Из этих экспериментов можно извлечь много полезного для совершенствования структуры новых вычислительных машин и их программного (математического) обеспечения.

2.16. Переходы

При организации передач управления в рабочих подпрограммах надо считаться с тем, что эти подпрограммы не занимают фиксированного места в памяти, а могут перемещаться во время уборки мусора. Поэтому адрес перехода должен задаваться в виде суммы

двух слагаемых — начального адреса (базы) участка памяти, занятого подпрограммой, и смещения адреса относительно базы. Смещение может быть вычислено во время компиляции, оно остается после этого неизменным на всем протяжении работы программы и может храниться в самой команде перехода. Текущее значение базы известно только во время выполнения программы. Оно хранится в виде адреса, обозначенного a_e , в составе преобразованного определяющего выражения компилированной функции (см. разд. 2.14). При уборке мусора адрес a_e корректируется автоматически.

Необходимо различать три вида передач управления: переходы внутри одной рабочей подпрограммы, обращения из одной подпрограммы к другой, возвраты после завершения выполнения подпрограммы.

Переходы внутри подпрограммы можно программировать двумя способами. Если конструкция машины позволяет фиксировать адрес A выполняемой команды, то истинный адрес B , по которому передается управление, можно вычислить, добавляя к адресу A разность смещений адресов B и A , известную во время компиляции. Более универсальный способ таков. В ячейке (регистре) *sbase* хранится база выполняемой в данный момент подпрограммы. Эта ячейка входит в число рабочих ячеек, содержимое которых корректируется во время уборки мусора. Адрес перехода вычисляется добавлением известного смещения к содержимому ячейки *sbase*.

Обращение из одной рабочей подпрограммы к другой происходит через специальный блок вызова (обозначенный идентификатором *call* в разд. 2.14). Аргументами обращения к блоку *call* являются адрес a_e базы вызываемой подпрограммы и адрес или смещение адреса возврата. Блок *call* запоминает в магазине это смещение, а также текущее значение *sbase*, заносит в *sbase* значение a_e и передает управление вызываемой подпрограмме по адресу, отличающемуся от a_e на небольшое смещение, стандартное для каждой системы компиляции. Немного сложнее происходит обращение к подпрограмме функционального аргумента. Для него аргументом обращения к блоку *call* вместо a_e является адрес, обозначенный выше L_e . В действительности \dot{L}_e должен состоять из двух отдельно хранимых компонент: базы a_e подпрограммы, содержащей функциональный аргумент, и смещения этого аргумента. Блок *call* должен добавлять к a_e вместо стандартного именно это индивидуальное смещение, а в остальном работать так же, как было описано выше. В этой схеме в магазине запоминаются как базы, так и смещения адресов возврата. Надо позаботиться, чтобы первые корректировались при уборке мусора, а вторые — нет.

Возврат из подпрограммы также целесообразно осуществлять через специальный блок *return*, который восстанавливает из магазина

значение *sbase*, извлекает оттуда же смещение адреса возврата и передает управление по соответствующему адресу.

Теперь можно окончательно уточнить состав рабочей подпрограммы, соответствующей определяющему выражению компилируемой функции. Каждая такая подпрограмма представляет собой отдельный участок области полных слов. Поэтому в начале этого участка должна содержаться ячейка — паспорт участка (см. разд. 2.13). Далее следует подпрограмма тела определяющего выражения, а за ней — обращение к блоку *return*. Для нужд конкретных реализаций к этому могут быть добавлены дополнительные ячейки со служебной информацией. Подпрограмма функционального аргумента также должна завершаться обращением к блоку *return* (имеется в виду подпрограмма, помеченная меткой L_c , см. разд. 2.14).

С переходами связана еще одна проблема, относящаяся уже не к структуре рабочих подпрограмм, а к процессу компиляции. Команда перехода и команда, которой передается управление, могут находиться в разных, далеких друг от друга частях рабочей подпрограммы. Иногда адресу перехода соответствует метка в исходной программе, иногда — нет. Адрес перехода может быть меньше любого из адресов команд перехода, тогда он известен в момент включения команды перехода в рабочую подпрограмму. Если это условие нарушается, то в некоторые команды перехода адрес перехода приходится вставлять лишь тогда, когда он определится.

В этих условиях формировать команды перехода можно следующим образом. Создается список пар, у которых первый элемент — настоящая или фиктивная метка, а второй — соответствующий ей адрес перехода или список адресов неокончательно сформированных команд перехода по этому адресу. Настоящая метка — это метка в *PROG*. Она представлена в списке адресом информационной ячейки атома-метки. Фиктивная метка изображается произвольным адресом, лишь бы он не совпадал с адресом какой-либо информационной ячейки. Адреса переходов и команд перехода представлены своими смещениями. В начале компиляции определяющего выражения этот список пуст. Если в компилируемом выражении встречается некоторая метка, то одновременно становится известным соответствующее ей смещение. Если эта метка еще не была представлена в списке, то в список добавляется новая пара, составленная из метки и смещения. Если же метка уже содержится в списке, то в паре с ней может быть только список адресов несформированных до конца команд перехода. Во все эти команды подставляется известное теперь смещение и второй элемент пары заменяется этим смещением.

Если компиляция достигает места, где должна стоять команда перехода, а метки (настоящей) в списке еще нет, то к списку доба-

вляется пара из этой метки и списка, содержащего пока адрес лишь одной этой команды перехода. Если пара с нужной меткой уже есть в списке и эта пара содержит смещение адреса перехода, то это смещение заносится в команду, на чем ее формирование заканчивается. Наконец, если в паре с меткой находится список недоформированных команд, то к нему добавляется адрес текущей команды перехода, а в ней адрес перехода остается пока незаполненным.

Сходным образом компилируются переходы к фиктивным меткам. Когда начинается компиляция выражения, за которым должна стоять фиктивная метка и в котором возможны переходы к этой метке, для нее выбирается новый, еще не использованный адрес, а в список заносится пара из этой фиктивной метки (т. е. выбранного адреса) и пустого списка команд перехода к ней. Одновременно этот адрес запоминается в магазине. Структура рабочей подпрограммы такова, что всякий раз, когда нужно сформировать команду перехода к фиктивной метке, представляющий ее адрес оказывается в вершине магазина. Он извлекается оттуда (без изменения указателя магазина), по нему в списке разыскивается соответствующая пара и к списку несформированных команд этой пары добавляется адрес текущей команды. Когда компиляция такого выражения заканчивается, из магазина окончательно выталкивается адрес, представляющий фиктивную метку, из списка удаляется соответствующая пара и ко всем командам, адреса которых содержатся в списке — втором элементе этой пары, приформировывается адрес перехода, который к этому моменту уже известен (до него дошло формирование рабочей подпрограммы).

Здесь не предусмотрен переход к фиктивной метке, стоящей раньше команды перехода. Но такой переход возможен лишь в одной ситуации — на начало подпрограммы тела определяющего выражения. Смещение адреса в этом случае стандартное и его нет нужды хранить в списке.

На этом мы завершаем описание компилятора, так как формирование всех остальных команд рабочих подпрограмм происходит достаточно просто.

ГЛАВА 3

БИБЛИОТЕКА ВСПОМОГАТЕЛЬНЫХ ФУНКЦИЙ

В этой главе даны определения функций, которые могут быть использованы в программах решения практических задач. Знакомство с определениями этих функций может способствовать овладению рядом приемов программирования.

Для некоторых функций даны несколько определений. Обычно первое определение (рекурсивное) более компактно, второе (через *PROG*) обеспечивает более быстрое выполнение обращений к функции в случае ее компиляции.

3.1. Операции над списками

Функция *COPY* создает в списочной памяти второй экземпляр произвольного выражения. Это может оказаться полезным, если один из этих экземпляров подвергается преобразованиям с помощью функций *RPLACA* и *RPLACD* или других функций, меняющих структуру внутреннего представления своих аргументов (см. разд. 3.2)

```
(SEXPR COPY (LAMBDA (X) (COND
  ((ATOM X) X)
  (T (CONS (COPY (CAR X))
            (COPY (CDR X)) )) )))
```

Функция *COPY*, как и многие другие, неприменима к циклическим списочным структурам, например к структуре, изображенной на рис. 2.3.3, б. Если же в списочной структуре есть сходящиеся на одной ячейке цепочки указателей, то при копировании возникает структура с тем же внешним представлением, но без таких цепочек (ср. рис. 2.1.8, а с рис. 2.1.8, б).

Функция APPEND присоединяет к одному списку другой, сохраняя относительный порядок элементов.

- а) (SEXPR APPEND (LAMBDA (X Y) (COND
((NULL X) Y)
(T (CONS (CAR X)
 (APPEND (CDR X) Y))))))
- б) (SEXPR APPEND (LAMBDA (X Y) (PROG (S)
 (SETQ X (REVERSE X))
 A (COND ((NULL X) (RETURN Y)))
 (SETQ S X) (SETQ X (CDR X))
 (SETQ Y (RPLACD S Y)) (GO A))))
- в) (SEXPR APPEND (LAMBDA (X Y) (PROG (U V)
 (SETQ U (SETQ V (CONS NIL Y)))
 A (COND ((NULL X) (RETURN (CDR U)))
 (RPLACD V (SETQ V (CONS (CAR X) Y)))
 (SETQ X (CDR X)) (GO A))))
- г) (SEXPR APPEND (LAMBDA (X Y) (COND
 (X (CONS (CAR X) (APPEND (CDR X) Y)))
 (T Y))))

Вариант б) обращается к функции *REVERSE* (см. ниже).

Вариант г) основан на том, что любое значение, отличное от *NIL*, изображает значение «истина».

Приемы программирования, продемонстрированные на примере функции *APPEND*, могут быть применены и при определении многих других функций.

П р и м е р:

(APPEND (QUOTE (A B)) (QUOTE ((A B) (C D) E)))
→ (A B (A B) (C D) E)

Функция REVERSE переставляет элементы списка в обратном порядке, не меняя самих элементов.

- а) (SEXPR REVERSE (LAMBDA (X)
 (REVERSE1 X NIL)))
 (SEXPR REVERSE1 (LAMBDA (X Y) (COND
 ((NULL X) Y)
 (T (REVERSE1 (CDR X) (CONS (CAR X) Y))))))
- б) (SEXPR REVERSE (LAMBDA (X) (PROG (U)
 A (COND ((NULL X) (RETURN U)))
 (SETQ U (CONS (CAR X) U))
 (SETQ X (CDR X)) (GO A))))

П р и м е р:

```
(REVERSE (QUOTE ((A B) (C D) F)))  
→ (E (C D) (A B))
```

Функция REMOVE из списка, заданного в качестве значения ее второго аргумента, исключает все элементы, совпадающие со значением первого аргумента. Преобразованный список выдается в качестве значения функции.

```
а) (SEXPR REMOVE (LAMBDA (X L) (COND  
  ((NULL L) NIL)  
  ((EQUAL X (CAR L)) (REMOVE X (CDR L)))  
  (T (CONS (CAR L) (REMOVE X (CDR L)))))) )  
б) (SEXPR REMOVE (LAMBDA (X L) (PROG (Y)  
  A (COND ((NULL L) (RETURN (REVERSE Y)))  
    ((NOT (EQUAL X (CAR L)))  
      (SETQ Y (CONS (CAR L) Y)) ) )  
  (SETQ L (CDR L)) (GO A) )))
```

П р и м е р:

```
(REMOVE (QUOTE A) (QUOTE (A B C A A C A))) →  
→ (B C C)
```

Функция REMOVEF из списка, являющегося значением ее второго аргумента, удаляет первый по порядку элемент, совпадающий со значением первого аргумента.

```
(SEXPR REMOVEF (LAMBDA (X Y) (COND  
  ((NULL Y) NIL)  
  ((EQUAL X (CAR Y)) (CDR Y))  
  (T (CONS (CAR Y) (REMOVEF X (CDR Y)))))) )
```

П р и м е р ы:

```
(REMOVEF (QUOTE (A B)) (QUOTE (A B (A B) C (A B)))) →  
→ (A B C (A B))  
(REMOVEF (QUOTE X) (QUOTE (A B C))) → (A B C)
```

Функция LAST выделяет последний элемент списка.

```
а) (SEXPR LAST (LAMBDA (X) (COND  
  ((NULL X) NIL)  
  ((NULL (CDR X)) (CAR X))  
  (T (LAST (CDR X)))) )  
б) (SEXPR LAST (LAMBDA (X) (PROG (U)  
  A (COND ((ATOM X) (RETURN U)))  
    (SETQ U (CAR X)) (SETQ X (CDR X))  
    (GO A) )))
```

Эти два варианта не вполне эквивалентны. Читателю рекомендуется подумать — почему, как сделать их эквивалентными и какому варианту следует отдать предпочтение.

Примеры:

$(LAST\ NIL) \rightarrow NIL$

$(LAST\ (QUOTE\ ((A\ B)\ C\ (D\ E)))) \rightarrow (D\ E)$

$(LAST\ (QUOTE\ (A\ B\ .\ C))) \rightarrow B$ (только для варианта б)).

Функция *LENGTH* вычисляет число элементов списка. Если аргумент — не список, а атом, то значение функции равно нулю.

а) $(SEXPR\ LENGTH\ (LAMBDA\ (X)\ (COND\ ((ATOM\ X)\ 0)\ (T\ (ADD1\ (LENGTH\ (CDR\ X))))))$

б) $(SEXPR\ LENGTH\ (LAMBDA\ (X)\ (PROG\ (U)\ (SETQ\ U\ 0)$

$A\ (COND\ ((ATOM\ X)\ (RETURN\ U)))$

$(SETQ\ X\ (CDR\ X))\ (SADD1\ U)\ (GO\ A)))$

Второй вариант использует функцию *SADD1* (см. разд. 2.11). Вместо $(SADD1\ U)$ можно написать $(SETQ\ U\ (ADD1\ U))$.

Примеры:

$(LENGTH\ (QUOTE\ (A\ B\ (C\ D)))) \rightarrow 3$

$(LENGTH\ (QUOTE\ A)) \rightarrow 0$

$(LENGTH\ (QUOTE\ (A\ .\ B))) \rightarrow 1$

Функция *ADDIFNONE* проверяет, содержится ли заданный элемент (значение первого аргумента) в заданном списке (в значении второго аргумента), и если нет, то добавляет этот элемент к списку.

$(SEXPR\ ADDIFNONE\ (LAMBDA\ (X\ L)\ (COND\ ((MEMBER\ X\ L)\ L)\ (T\ (CONS\ X\ L))))$

Примеры:

$(ADDIFNONE\ (QUOTE\ A)\ (QUOTE\ (A\ B\ C))) \rightarrow (A\ B\ C)$

$(ADDIFNONE\ NIL\ (QUOTE\ (A\ B\ C))) \rightarrow (NIL\ A\ B\ C)$

$(ADDIFNONE\ (QUOTE\ A)\ NIL) \rightarrow (A)$

Функция *COLLECT* перегруппировывает элементы заданного списка так, чтобы одинаковые элементы, если они есть в списке, стояли все подряд.

```
(SEXPR COLLECT (LAMBDA (L) (COND
  ((NULL L) NIL)
  (T (CONS (CAR L) (COLLECT (COND
    ((MEMBER (CAR L) (CDR L))
      (CONS (CAR L) (REMOVEF (CAR L) (CDR L)))) )
    (T (CDR L)) )))) ))
```

Пр и м е р:

```
(COLLECT (QUOTE (A B D A A C B E C B))) →
→ (A A A B B B D C C E)
```

Функция FLATTEN устраняет в произвольном выражении все внутренние скобки, а в точечных выражениях — и точки, превращая его в список атомов. Количество и относительный порядок атомов в выражении сохраняются.

```
а) (SEXPR FLATTEN (LAMBDA (X) (COND
  ((NULL X) NIL)
  ((ATOM X) (LIST X))
  (T (APPEND (FLATTEN (CAR X))
    (FLATTEN (CDR X)) ) ) )))
б) (SEXPR FLATTEN (LAMBDA (X) (PROG (U V)
  A (COND ((NULL X) (GO C))
    ((ATOM X) (GO B)))
  (SETQ V (CONS (CAR X) V))
  (SETQ X (CDR X)) (GO A)
  B (SETQ U (CONS X U))
  C (COND ((NULL V) (RETURN U)))
  (SETQ X (CAR V)) (SETQ V (CDR V))
  (GO A) )))
```

Вариант а) использует функции *LIST* и *APPEND*.

Пр и м е р ы:

```
(FLATTEN (QUOTE (A ((B C)) (D . E) ((A B) . C)
  (((2 3) 4)) )) → (A B C D E A B C 2 3 4)
(FLATTEN (QUOTE (NIL . A))) → (A)
```

3.2. Функции с побочным эффектом

В этом разделе представлены некоторые функции, в определении которых имеются обращения к функциям *RPLACA* или *RPLACD*. Поэтому они, как правило, изменяют значение выражений, заданных в качестве их аргументов. Более того, они могут создавать объ-

екты, внутренние представления которых в памяти имеют общие части. Последующее изменение значения одного из таких объектов влечет изменение значения и другого объекта.

Пусть, например, выполняется следующий участок программы:

```
(CSETQ C1 (QUOTE (A B)))
(CSETQ C2 (NCONC (QUOTE (C D)) C1))
```

(см. ниже определение функции *NCONC*). Значением константы *C2* станет список *(C D A B)*, но два его последних элемента — общие с элементами списка *(A B)*, являющегося значением константы *C1*.

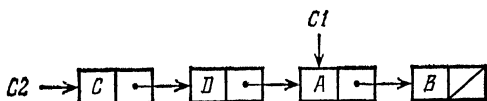


Рис. 3.2.1.

В памяти образуется структура, изображенная на рис. 3.2.1. Поэтому последующее выполнение выражения

```
(CSETQ C2 (NCONC C2 (QUOTE (E))))
```

приведет к тому, что значением *C1* станет список *(A B E)*.

Функция *ATTACH* вырабатывает то же значение, что и *CONS*, но, в отличие от *CONS*, она заставляет обладать этим значением свой второй аргумент.

```
(SEXPR ATTACH (LAMBDA (X Y)
  (RPLACA (RPLACD Y (CONS
    (CAR Y) (CDR Y))) X) ))
```

Пример. Пусть константа *C* обладает значением *(A B)*. Тогда

```
(ATTACH (QUOTE C) C) → (C A B)
```

и значение *C* изменится на *(C A B)*. После этого

```
(ATTACH (QUOTE (D E)) (CDR C))
→ ((D E) A B)
```

а значение *C* станет равно

```
(C (D E) A B)
```

Функция DREVERSE вырабатывает то же значение, что и REVERSE (см. разд. 3.1), но разрушает свой аргумент.

```
(SEXPR DREVERSE (LAMBDA (X) (PROG (U V)
  A (COND ((NULL X) (RETURN U)))
  (SETQ V X) (SETQ X (CDR X))
  (SETQ U (RPLACD V U)) (GO A) )))
```

Пр и м е р. Пусть значение константы C равно $(A B (C D))$. Тогда

$$(DREVERSE C) \rightarrow ((C D) B A)$$

и после вычисления этого выражения значением C станет (A) .

Функция NCONC вырабатывает то же значение, что и APPEND (см. разд. 3.1), но одновременно она заставляет обладать этим значением свой первый аргумент.

```
а) (SEXPR NCONC (LAMBDA (X Y) (COND
  ((NULL X) Y)
  (T (RPLACD X (NCONC (CDR X) Y))))) )))
б) (SEXPR NCONC (LAMBDA (X Y) (PROG (U)
  (COND ((NULL X) (RETURN Y)))
  (SETQ U X)
  A (COND ((NULL (CDR U)) (GO B)))
  (SETQ U (CDR U)) (GO A)
  B (RPLACD U Y) (RETURN X) )))
```

Пр и м е р. Пусть значение константы C равно $((A B) (C D) E)$. Тогда

$$(NCONC (QUOTE (A B)) C) \rightarrow (A B (A B) (C D) E)$$

$$(NCONC C (QUOTE (A B))) \rightarrow ((A B) (C D) E A B)$$

После выполнения первого из этих выражений значение C не изменится, а после второго — станет равным

$$((A B) (C D) E A B)$$

Функция TCONS. Очередью будем называть списочную структуру, состоящую из некоторого списка и лисповской ячейки, содержащей указатели на первый и последний элементы этого списка. На рис. 3.2.2 изображены очереди, состоящие из элементов: а) A ; б) A, B .

Функция TCONS помещает значение своего первого аргумента в конец очереди, представленной вторым аргументом. Если эта очередь пуста, то формируется очередь, состоящая из одного элемента,

```
(SEXPR TCONC (LAMBDA (X Q) (COND
  ((NULL Q) (CONS (SETQ Q (CONS X NIL)) Q))
  (T (RPLACD Q (CDR (RPLACD (CDR Q)
    (CONS X NIL) )))))))
```

Пусть, например, одно за другим выполняются следующие два выражения:

```
(CSETQ C (TCONC (QUOTE A) NIL))
(TCONC (QUOTE B) C)
```

Внутреннее представление значения константы *C* после вычисления первого выражения изображено на рис. 3.2.2, а, а после второго — на рис. 3.2.2, б.

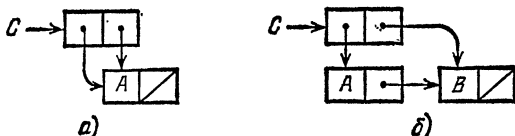


Рис. 3.2.2.

Заметим, что доступ к первому элементу очереди *Q* дает функция (*CAAR Q*), а к последнему — (*CADR Q*). Исключить из очереди *Q* первый (но не единственный) элемент можно с помощью выражения

```
(RPLACA Q (CDAR Q))
```

Функция *EFFACE*. Значением второго аргумента функции *EFFACE* должен быть список. Если этот список содержит хотя бы один элемент, совпадающий со значением первого аргумента, то первый по порядку из этих элементов исключается из списка, в противном случае список не меняется. Значением функции является преобразованный список. Если выброшенный элемент не был самым первым в списке, то значением второго аргумента также становится преобразованный список.

```
(SEXPR EFFACE (LAMBDA (X Y) (COND
  ((NULL Y) NIL)
  ((EQUAL X (CAR Y)) (CDR Y))
  (T (RPLACD Y (EFFACE X (CDR Y))))))
```

Примеры. Пусть значение константы *C* равно ((*A B*) *B* (*A B*)). Тогда

```
(EFFACE (QUOTE (A B)) C) → (B (A B))
```

Значение *C* не меняется.

$(\text{EFFACE } (\text{QUOTE } B) C) \rightarrow ((A B) (A B))$

Константа C получает значение $((A B) (A B))$. После этого

$(\text{EFFACE } (\text{QUOTE } A) C) \rightarrow ((A B) (A B))$

Значение C не меняется.

Функция $DREMOVE$, в отличие от $EFFACE$, выбрасывает из списка, являющегося значением ее второго аргумента, все элементы, совпадающие со значением первого аргумента функции.

$(\text{SEXPR } DREMOVE (\text{LAMBDA } (X Y) (\text{COND}$
 $((\text{NULL } Y) \text{NIL})$
 $((\text{EQUAL } X (\text{CAR } Y)) (DREMOVE X (\text{CDR } Y))$
 $(\text{T } (\text{RPLACD } Y (DREMOVE X (\text{CDR } Y))))))$)

П р и м е р ы. Пусть значение константы C равно $((A B) B (A B))$. Тогда

$(DREMOVE (\text{QUOTE } (A B)) C) \rightarrow (B)$

Константа C получает значение $((A B) B)$.

$(DREMOVE (\text{QUOTE } B) C) \rightarrow ((A B) (A B))$

Константа C получает значение $((A B) (A B))$.

Следующие две функции служат для распознавания циклических списочных структур, которые могут возникнуть в результате применения функций $RPLACD$ и $RPLACA$ (см. разд. 2.3).

Предикат $LCYCLEP$ вырабатывает значение T , если в значении аргумента есть цикл по цепочке d -указателей, и NIL в противном случае, т. е. если в результате многократного применения функции CDR к значению аргумента можно получить атом.

$(\text{SEXPR } LCYCLEP (\text{LAMBDA } (X) (\text{AND}$
 $(\text{NOT } (\text{ATOM } X)) (\text{NOT } (\text{ATOM } (\text{CDR } X)))$
 $(\text{LCYCLEI } (\text{CDR } X) (\text{CDDR } X)))$)

$(\text{SEXPR } LCYCLEI (\text{LAMBDA } (X Y) (\text{OR}$
 $(\text{EQ } X Y) (\text{AND } (\text{NOT } (\text{ATOM } Y))$
 $(\text{NOT } (\text{ATOM } (\text{CDR } Y)))$
 $(\text{LCYCLEI } (\text{CDR } X)$
 $(\text{CDDR } Y)))$)

П р и м е р ы. Если X имеет значение, изображенное на рис. 2.3.3, а (или, вообще, любое значение, представимое на лиспе), то $(LCYCLEP X) \rightarrow NIL$.

Если же значение X таково, как показано на рис. 2.3.3, б, то

$(LCYCLEP X) \rightarrow T$

$(LCYCLEP (\text{CDR } X)) \rightarrow T$

однако

$$(LCYCLES (CONS X NIL)) \rightarrow NIL$$

Предикат CYCLES вырабатывает значение *T*, если значение аргумента — циклическая списочная структура, и значение *NIL*,

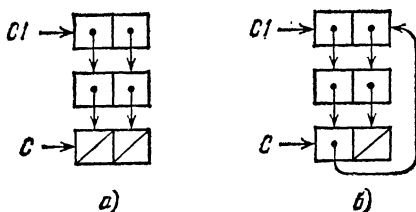


Рис. 3.2.3.

если в значении аргумента нет циклов (циклических цепочек указателей). Использует функцию *CYCLES1*, первый аргумент которой — одна из вершин (подструктур) списочной структуры, исследуемой в *CYCLES*, второй аргумент — путь от этой вершины к исходной вершине, третий — список уже обследованных вершин.

```
(SEXPR CYCLES (LAMBDA (X)
  (NULL (CYCLES1 X NIL T)) ))
(SEXPR CYCLES1 (LAMBDA (X U V) (COND
  ((ATOM X) V)
  ((MEMB X U) NIL)
  ((MEMB X V) V)
  ((NULL (SETQ V (CYCLES1 (CAR X)
    (SETQ U (CONS X U)) V))) NIL)
  ((NULL (SETQ V (CYCLES1 (CDR X) U V))) NIL)
  (T (CONS X V)) )))
```

Примеры. Для структуры, изображенной на рис. 3.2.3, *а*:

```
(CYCLES C) → NIL
(CYCLES CI) → NIL
```

Если изменить содержимое одного из указателей ячейки *C*, как показано на рис. 3.2.3, *б*, то

```
(CYCLES C) → T
(CYCLES CI) → T
```

3.3. Предикаты

Предикаты *FORALL*, *FORSOME* и *FORODD*. У этих предикатов по два аргумента. Значением первого аргумента должен быть некоторый список *l*, а второго (функционального) аргумента — наименование или определяющее выражение функции *p*. Предикат *FORALL* принимает значение *T* лишь в том случае, если функция *p* принимает значение «истина» (т. е. не *NIL*) на всех элементах списка *l*. Предикат *FORSOME* принимает значение *T*, если функция *p* принимает значение «истина» хотя бы на одном элементе списка *l*. Предикат *FORODD* принимает значение *T*, если число элементов списка *l*, на которых функция *p* принимает значение «истина», нечетно.

```
(SEXPR FORALL (LAMBDA (L P) (COND
  ((NULL L) T)
  ((P (CAR L)) (FORALL (CDR L) P))
  (T NIL) )))
(SEXPR FORSOME (LAMBDA (L P) (COND
  ((NULL L) NIL)
  ((P (CAR L)) T)
  (T (FORSOME (CDR L) P)) )))
(SEXPR FORODD (LAMBDA (L P) (COND
  ((NULL L) NIL)
  ((P (CAR L)) (NOT (FORODD (CDR L) P)))
  (T (FORODD (CDR L) P)) )))
```

Примеры. Пусть константы *C*, *C1* и *C2* имеют соответственно значения $(A\ B\ C)$, $(A\ (B\ C)\ (D\ E))$ и $((A\ B)\ (C\ D))$. Тогда

```
(FORALL C (FUNCTION ATOM)) → T
(FORALL C1 (FUNCTION ATOM)) → NIL
(FORSOME C1 (FUNCTION ATOM)) → T
(FORSOME C2 (FUNCTION ATOM)) → NIL
(FORODD C1 (FUNCTION ATOM)) → T
(FORODD C1 (FUNCTION (LAMBDA (X)
  (NOT (ATOM X)) ))) → NIL
```

Предикат *ATOMLIST* проверяет, является ли его аргумент списком (возможно, пустым), составленным лишь из атомов.

```
а) (SEXPR ATOMLIST (LAMBDA (X) (COND
  ((NULL X) T)
  ((ATOM X) NIL)
  ((ATOM (CAR X)) (ATOMLIST (CDR X)))
  (T NIL) )))
```

```

6) (SEXPR ATOMLIST (LAMBDA (X)
    (OR (NULL X)
        (AND (NOT (ATOM X))
              (ATOM (CAR X))
              (ATOMLIST (CDR X)) ) ) ))

```

Пр и м е р ы:

```

(ATOMLIST NIL) → T
(ATOMLIST (QUOTE A)) → NIL
(ATOMLIST (QUOTE (A))) → T
(ATOMLIST (QUOTE ((A B) (C D) E))) → NIL

```

Предикат LISTP принимает значение *NIL*, если заданное выражение является атомом, отличным от *NIL*, или выражением, которое может быть записано только в точечных обозначениях (см. разд. 1.31). В противном случае заданное выражение является списком, который можно записать, не прибегая к точечным обозначениям ни на одном из уровней, и предикат принимает значение *T*.

```

(SEXPR LISTP (LAMBDA (X) (COND
    ((NULL X) T)
    ((ATOM X) NIL)
    ((OR (ATOM (CAR X)) (LISTP (CAR X)))
        (LISTP (CDR X)))
    (T NIL) )))

```

Пр и м е р ы:

```

(LISTP (QUOTE ((A B) (B . C) C))) → NIL
(LISTP (QUOTE ((A . (B C)) (B . (C A) )))) → T

```

3.4. Порядок и упорядочивание

В этом разделе собраны функции и предикаты, связанные с проверкой выполнения некоторых отношений порядка между данными объектами и упорядочиванием списков.

Предикат ORDER проверяет, в каком порядке два заданных элемента встречаются в данном списке. Список (упорядочивающая последовательность) задается в качестве значения третьего аргумента функции *ORDER*. Если при просмотре элементов этого списка слева направо встречается элемент, совпадающий со значением первого аргумента, а ни один из ранее просмотренных элементов не

совпал со значением второго аргумента, то значение предиката равно *T*, во всех остальных случаях оно равно *NIL*.

```
(SEXPR ORDER (LAMBDA (X Y L) (COND
  ((NULL L) NIL)
  ((EQUAL X (CAR L)) T)
  ((EQUAL Y (CAR L)) NIL)
  (T (ORDER X Y (CDR L))) )))
```

Примеры:

```
(ORDER (QUOTE A) (QUOTE E)
  (QUOTE (A B C D E))) → T
(ORDER (QUOTE D) (QUOTE B)
  (QUOTE (A B C D E))) → NIL
(ORDER (QUOTE C) (QUOTE F)
  (QUOTE (A B C D E))) → T
(ORDER (QUOTE F) (QUOTE C)
  (QUOTE (A B C D E))) → NIL
(ORDER (QUOTE F) (QUOTE G)
  (QUOTE (A B C D E))) → NIL
```

Предикат *ORDER1* вычисляется так же, как и *ORDER*, однако, если ни один из заданных элементов не содержится в данном списке, то в качестве значения предиката выдается атом *ORDERUNDEF*.

```
(SEXPR ORDER1 (LAMBDA (X Y L) (COND
  ((NULL L) (QUOTE ORDERUNDEF))
  ((EQUAL X (CAR L)) T)
  ((EQUAL Y (CAR L)) NIL)
  (T (ORDER1 X Y (CDR L))) )))
```

Если в первых четырех примерах на применение предиката *ORDER* использовать предикат *ORDER1*, то значения останутся прежними. В пятом примере выработается значение *ORDERUNDEF*.

Предикат *LEXORDER* сравнивает — элемент за элементом — два списка, заданные как значения первого и второго аргументов. Если в какой-либо позиции обнаруживаются различные элементы, то они сравниваются между собой с помощью предиката *ORDER1*, причем в качестве третьего аргумента (упорядочивающей последовательности) указывается третий аргумент обращения к *LEXORDER*. Результат сравнения (*T*, *NIL* или *ORDERUNDEF*) выдается в качестве значения предиката *LEXORDER*. Если раньше, чем встретятся различные элементы, исчерпается первый список, то выдается

результат *T*; если первым исчерпается второй список, то в качестве результата выдается *NIL*.

```
(SEXPR LEXORDER (LAMBDA (X Y L) (COND
  ((NULL X) T) ((NULL Y) NIL)
  ((EQUAL (CAR X) (CAR Y))
    (LEXORDER (CDR X) (CDR Y) L))
  (T (ORDER1 (CAR X) (CAR Y) L)) )))
```

П р и м е р ы:

```
(LEXORDER (QUOTE (B A)) (QUOTE (B C))
  (QUOTE (A B C))) → T
(LEXORDER (QUOTE (B A)) (QUOTE (A C))
  (QUOTE (A B C))) → NIL
(LEXORDER (QUOTE (B)) (QUOTE (B A))
  (QUOTE (A B C))) → T
(LEXORDER (QUOTE (A C)) (QUOTE (B))
  (QUOTE (A B C))) → T
```

Предикат *LEXORDER1* отличается от *LEXORDER* тем, что значением третьего аргумента для него должен быть список, каждая позиция которого в свою очередь является списком, который используется в случае необходимости для сравнения соответствующих позиций списков, заданных в качестве значений первых двух аргументов.

```
(SEXPR LEXORDER1 (LAMBDA (X Y L) (COND
  ((NULL X) T) ((NULL Y) NIL)
  ((NULL L) (QUOTE LEXORDUNDEF))
  ((EQUAL (CAR X) (CAR Y))
    (LEXORDER1 (CDR X)
      (CDR Y) (CDR L)))
  (T (ORDER1 (CAR X) (CAR Y) (CAR L)))))
```

Если во всех примерах применения предиката *LEXORDER* воспользоваться предикатом *LEXORDER1*, заменив третий аргумент на *(QUOTE ((A B C) (A B C)))*, то значение предиката не изменится. Если в качестве третьего аргумента взять *(QUOTE ((A B C)))*, то в первом примере будет получен результат *LEXORDUNDEF*, а в трех других примерах значение останется прежним. Если третий аргумент заменить на *(QUOTE ((1 2 3) (A B C)))*, то в первом и третьем примерах результат сохранится, а во втором и четвертом примерах выработается результат *ORDERUNDEF*.

Функция *FIRST* среди элементов списка, заданного в качестве значения первого аргумента, выбирает тот, который раньше

встречается в списке, заданном в качестве значения второго аргумента. Если ни один из элементов первого списка не содержится во втором, то выбирается первый элемент первого списка.

```
(SEXPR FIRST (LAMBDA (X Y) (COND
  ((NULL Y) (CAR X))
  ((MEMBER (CAR Y) X) (CAR Y))
  (T (FIRST X (CDR Y))) )))
```

Пр и м е р ы:

```
(FIRST (QUOTE (C O N D)))
  (QUOTE (P R O D U C T I O N))) → O
(FIRST (QUOTE (C O N D)))
  (QUOTE (L A N G U A G E))) → N
(FIRST (QUOTE (C O N D)))
  (QUOTE (F I R S T))) → C
```

Функция RANK упорядочивает список, заданный в качестве ее первого аргумента, переставляя его элементы в той последовательности, в какой они встречаются в списке, являющемся значением второго аргумента.

```
а) (SEXPR RANK (LAMBDA (X Y) (COND
  ((NULL X) NIL)
  (T (CONS (FIRST X Y)
    (RANK (REMOVEF
      (FIRST X Y) X) Y) )) )))
б) (SEXPR RANK (LAMBDA (X Y) (PROG (U V W)
  A (COND ((OR (NULL X) (NULL Y)) (GO C)))
    (SETQ U (CAR Y)) (SETQ Y (CDR Y))
    (SETQ V X) (SETQ X NIL)
  B (COND ((EQUAL (CAR V) U) (SETQ W (CONS U W)))
    (T (SETQ X (CONS (CAR V) X))) )
    (SETQ V (CDR V))
    (COND ((NULL V) (GO A))) (GO B)
  C (COND ((NULL W) (RETURN X)))
    (SETQ X (CONS (CAR W) X))
    (SETQ W (CDR W)) (GO C) )))
```

Функция RANK (вариант а)) использует функции REMOVEF, FIRST и ORDER.

Пр и м е р :

```
(RANK (QUOTE (B A B A)) (QUOTE (A B))) →
  → (A A B B)
```

3.5. Поиск

В этом разделе собраны функции, выбирающие из списка элемент или элементы, обладающие заданным свойством. Список задается в виде значения аргумента, соответствующего связанной переменной *L*, свойство характеризуется предикатом, наименование или определяющее выражение которого дано в качестве значения аргумента, соответствующего связанной (функциональной) переменной *P*.

Функция *POSSESSING* образует список из всех элементов данного списка, обладающих заданным свойством.

- а) `(SEXPR POSSESSING (LAMBDA (P L) (COND
((NULL L) NIL)
((P (CAR L)) (CONS (CAR L)
(POSSESSING P (CDR L))))
(T (POSSESSING P (CDR L))))))`
- б) `(SEXPR POSSESSING (LAMBDA (P L) (PROG (U)
A (COND ((NULL L) (RETURN (REVERSE U)))
((P (CAR L)) (SETQ U (CONS (CAR L) U))))
(SETQ L (CDR L)) (GO A))))`

Примеры:

```
(POSSESSING (FUNCTION ATOM)
  (QUOTE (A (A B) B (B . C) C))) → (A B C)
(POSSESSING (FUNCTION (LAMBDA (X) (NOT (ATOM X))))
  (QUOTE (A (A B) B (B . C) C C)))
→ ((A B) (B . C) (C))
```

Функция *SUCHTHAT* выбирает из заданного списка первый элемент, обладающий заданным свойством. Если такого элемента нет, то вырабатывается значение *NIL*.

```
(SEXPR SUCHTHAT (LAMBDA (P L) (COND
((NULL L) NIL)
((P (CAR L)) (CAR L))
(T (SUCHTHAT P (CDR L))))))
```

Примеры:

```
(SUCHTHAT (FUNCTION ATOM)
  (QUOTE ((A . B) (B C) C))) → C
(SUCHTHAT (FUNCTION LISTP)
  (QUOTE ((A . B) (B C) C))) → (B C)
```

Определение предиката *LISTP* см. в разд. 3.3.

Функция *SUCHTHAT1* проверяет, содержится ли в данном списке хотя бы один элемент с заданным свойством. Если да, то в момент обнаружения такого элемента в качестве результата принимается значение четвертого аргумента функции *SUCHTHAT1*. Если нет, то результатом является значение третьего аргумента.

```
(SEXPR SUCHTHAT1 (P L X Y) (COND
  ((NULL L) X) ((P (CAR L)) Y)
  (T (SUCHTHAT1 P (CDR L) X Y)) )))
```

Пр и м е р ы:

```
(SUCHTHAT1 (FUNCTION ATOM)
  (QUOTE ((A . B) B (B . C)))
  (QUOTE NONE) (QUOTE FOUND)) → FOUND
(SUCHTHAT1 (FUNCTION LISTP)
  (QUOTE ((A . B) B (B . C)))
  (QUOTE NONE) (QUOTE FOUND)) → NONE
```

Функция *SUCHTHAT2* проверяет, содержится ли в данном списке хотя бы один элемент с заданным свойством. Если да, то к хвосту заданного списка, начиная с найденного элемента, применяется функция, наименование или определяющее выражение которой задано в качестве значения третьего аргумента (функционального). Если нет, то вырабатывается значение *NIL*.

```
(SEXPR SUCHTHAT2 (LAMBDA (P L F) (COND
  ((NULL L) NIL) ((P (CAR L)) (F L))
  (T (SUCHTHAT2 P (CDR L) F)) )))
```

Пр и м е р ы. Пусть значение константы *C* равно $((A . B) (B C) A)$. Тогда

```
(SUCHTHAT2 (FUNCTION ATOM) C (FUNCTION
  (LAMBDA (X) X))) → (A)
(SUCHTHAT2 (FUNCTION LISTP) C (FUNCTION
  (LAMBDA (X) X))) → ((B C) A)
(SUCHTHAT2 (FUNCTION NUMBERP) C
  (FUNCTION (LAMBDA (X) X))) → NIL
```

3.6. Операции над множествами

В этом разделе списки рассматриваются как множества своих элементов — порядку элементов в списке не придается значения, а два или более одинаковых элемента списка рассматриваются как один элемент множества.

Функция SETOF для каждого повторяющегося элемента исключает из списка все вхождения, кроме одного.

```
(SEXPR SETOF (LAMBDA (X) (COND
  ((NULL X) NIL)
  ((MEMBER (CAR X) (CDR X))
   (SETOF (CDR X)))
  (T (CONS (CAR X) (SETOF (CDR X)))))))
```

Пример:

```
(SETOF (QUOTE (A A X B B A L C B)))
→ (X A L C B)
```

Функция MAKESET делает то же, что SETOF, но описана через PROG. Порядок элементов в результирующем списке оказывается другим.

```
(SEXPR MAKESET (LAMBDA (X) (PROG (Y)
  A (COND ((NULL X) (RETURN Y))
  ((NOT (MEMBER (CAR X) Y))
   (SETQ Y (CONS (CAR X) Y)) ) )
  (SETQ X (CDR X)) (GO A) )))
```

Пример:

```
(MAKESET (QUOTE (A A X B B A L C B)))
→ (C L B X A)
```

Функция DIFFLIST вычисляет разность множеств $X \setminus Y$. Иначе говоря, она исключает из списка, заданного в качестве значения первого аргумента функции, все элементы, встречающиеся в списке, представленном значением второго аргумента.

```
(SEXPR DIFFLIST (LAMBDA (X Y) (COND
  ((NULL Y) X)
  (T (DIFFLIST (REMOVE (CAR Y) X) (CDR Y)))))
```

Функция DIFFLIST обращается к функции REMOVE.

Пример:

```
(DIFFLIST (QUOTE (A A X B B A L C B))
  (QUOTE (A B))) → (X L C)
```

Функция SUBSET вычисляет предикат $X \subseteq Y$. Иначе говоря, она вырабатывает значение T, если каждый элемент списка, заданного в качестве первого аргумента функции, содержится в списке, представленном значением второго аргумента.

- а) (SEXPR SUBSET (LAMBDA (X Y)
 (OR (NULL X)
 (AND (MEMBER (CAR X) Y)
 (SUBSET (CDR X) Y)))))
 б) (SEXPR SUBSET (LAMBDA (X Y)
 (NULL (DIFFLIST X Y))))

Примеры:

(SUBSET (QUOTE (A B))
 (QUOTE (A A X B B A L C B))) → T
 (SUBSET NIL (QUOTE (A B))) → T
 (SUBSET (QUOTE (A B C))
 (QUOTE (A B X Y))) → NIL

Функция UNION вычисляет объединение двух множеств. Значение функции представляет собой список всех выражений, являющихся элементами хотя бы одного из заданных списков. Если каждый из заданных списков не содержал повторяющихся элементов, то в результирующий список каждый элемент войдет лишь один раз (для варианта б) оговоренное условие необязательно).

- а) (SEXPR UNION (LAMBDA (X Y) (COND
 ((NULL X) Y)
 ((MEMBER (CAR X) Y) (UNION (CDR X) Y))
 (T (CONS (CAR X) (UNION (CDR X) Y))))))
 б) (SEXPR UNION (LAMBDA (X Y) (PROG NIL
 (SETQ Y (MAKESET Y))
 A (COND ((NULL X) (RETURN Y))
 ((NOT (MEMBER (CAR X) Y))
 (SETQ Y (CONS (CAR X) Y))))
 (SETQ X (CDR X)) (GO A))))

Пример:

(UNION (QUOTE (A A B)) (QUOTE (B B C)))
 → (A A B B C)

- б) (SEXPR UNION (LAMBDA (X Y) (PROG NIL
 (SETQ Y (MAKESET Y))
 A (COND ((NULL X) (RETURN Y))
 ((NOT (MEMBER (CAR X) Y))
 (SETQ Y (CONS (CAR X) Y))))
 (SETQ X (CDR X)) (GO A))))

Примеры:

(UNION (QUOTE (A A B)) (QUOTE (B B C))) → (A C B)
 (UNION (QUOTE (A B C)) (QUOTE (1 2 3)))
 → (C B A 3 2 1)

Функция LUNION объединяет множества, заданные в качестве элементов списка, являющегося значением аргумента функции.

(SEXPR LUNION (LAMBDA (X) (COND
 ((NULL X) NIL)
 (T (UNION (CAR X) (LUNION (CDR X)))))))

Пр и м е р (в этом примере используется вариант б) функции UNION):

(LUNION (QUOTE ((A (B C) C) ((B C) B)
(C (A B))))) → (A (A B) C (B C) B)

Функция INTERSECTION вычисляет пересечение двух множеств. Значением функции является список всех выражений, входящих элементами в оба заданных списка. Если каждый из заданных списков не содержит повторяющихся элементов (для варианта б) это условие необязательно), то и в результирующем списке элементы не будут повторяться.

а) (SEXPR INTERSECTION (LAMBDA (X Y) (COND
((NULL X) NIL)
((MEMBER (CAR X) Y)
(CONS (CAR X)
(INTERSECTION (CDR X) Y)))
(T (INTERSECTION (CDR X) Y)))))

Пр и м е р:

(INTERSECTION (QUOTE (1 2 3 3 2 1))
(QUOTE (5 5 3 3 1 1))) → (1 3 3 1)

б) (SEXPR INTERSECTION (LAMBDA (X Y) (PROG (U)
(SETQ X (MAKESET X))
A (COND ((NULL X) (RETURN U))
((MEMBER (CAR X) Y)
(SETQ U (CONS (CAR X) U)))
(SETQ X (CDR X)) (GO A))))

Пр и м е р ы:

(INTERSECTION (QUOTE (1 2 3 3 2 1))
(QUOTE (5 5 3 3 1 1))) → (1 3)
(INTERSECTION (QUOTE (1 2 3))
(QUOTE (A B C))) → NIL

Предикат EQUALSET проверяет, равны ли множества, представленные двумя заданными списками.

а) (SEXPR EQUALSET (LAMBDA (X Y)
(AND (SUBSET X Y) (SUBSET Y X))))
б) (SEXPR EQUALSET (LAMBDA (X Y) (PROG (U)
(SETQ U X)
A (COND ((NULL U) (GO B))
((NOT (MEMBER (CAR U) Y))
(RETURN NIL)))

```

    (SETQ U (CDR U)) (GO A)
  В (COND ((NULL Y) (ETURN T))
          ((NOT (MEMBER (CAR Y) X))
           (RETURN NIL)) )
    (SETQ Y (CDR Y)) (GO B) )))

```

Пр и м е р ы:

```

(EQUALSET NIL NIL) → T
(EQUALSET (QUOTE (1 2 3))
           (QUOTE (1 2 3 4))) → NIL
(EQUALSET (QUOTE (1 2 3 3 3))
           (QUOTE (3 2 1 1 1))) → T

```

Функция CART образует декартово (прямое) произведение двух заданных множеств. Точнее говоря, она формирует лексикографически упорядоченный список, элементами которого являются всевозможные списки, содержащие по два элемента каждый, причем первый элемент берется из первого, а второй — из второго заданного списка.

```

(SEXPR CART (LAMBDA (X Y) (PROG (U V W)
  А (COND ((NULL X) (RETURN (REVERSE W))))
    (SETQ U (CAR X))
    (SETQ X (CDR X)) (SETQ V Y)
  В (COND ((NULL V) (GO A)))
    (SETQ W (CONS (LIST U (CAR V)) W))
    (SETQ V (CDR V)) (GO B) )))

```

П р и м е р:

```

(CART (QUOTE (A B C)) (QUOTE (1 2 3 4))) →
→ ((A 1) (A 2) (A 3) (A 4)
   (B 1) (B 2) (B 3) (B 4)
   (C 1) (C 2) (C 3) (C 4))

```

3.7. Ассоциативные списки

Ассоциативным списком или *списком соответствия* называется список вида

$$((x_1 \cdot y_1) (x_2 \cdot y_2) \dots (x_n \cdot y_n)),$$

где x_i и y_i для $i = 1, \dots, n$ — произвольные выражения. Говорят, что такой список ставит в соответствие выражение y_i выражению x_i .

У функций этого раздела аргумент A всегда имеет своим значением некоторый ассоциативный список, возможно, пустой.

Функция PAIR объединяет элементы двух заданных списков в ассоциативный список, вырабатываемый в качестве значения функции. Функция не определена, если второй из заданных списков короче первого.

- а) (SEXPR PAIR (LAMBDA (X Y) (COND
 ((NULL X) NIL)
 (T (CONS (CONS (CAR X) (CAR Y))
 (PAIR (CDR X) (CDR Y)))))))
 б) (SEXPR PAIR (LAMBDA (X Y) (PROG (U)
 A (COND ((NULL X) (RETURN (REVERSE U))))
 (SETQ U (CONS (CONS (CAR X)
 (CAR Y)) U))
 (SETQ X (CDR X)) (SETQ Y (CDR Y))
 (GO A))))

Вариант б) использует функцию *REVERSE*.

Пр и м е р ы:

(PAIR (QUOTE (A B C)) (QUOTE (1 2 3)))
 → ((A . 1) (B . 2) (C . 3))
 (PAIR (QUOTE ((A B) D ((A B) D) C))
 (QUOTE (1 2 3 4 5))) →
 → (((A B) . 1) (D . 2) (((A B) D) . 3) (C . 4))

Функция DPAIR действует аналогично *PAIR*, он значение первого аргумента заменяется значением функции.

(SEXPR DPAIR (LAMBDA (X Y) (PROG (U)
 (SETQ U X)
 A (COND ((NULL U) (RETURN X)))
 (RPLACA U (CONS (CAR U) (CAR Y)))
 (SETQ U (CDR U))
 (SETQ Y (CDR Y)) (GO A))))

Пр и м е р. Пусть значение константы *C* равно *(A (B . C) D)*. Тогда

(DPAIR (CDR C) (QUOTE ((A B) (C . D)))) →
 → (((B . C) A B) (D C . D))

и значение *C* изменится на

(A ((B . C) A B) (D C . D))

Функция ASSOC вырабатывает в качестве значения первую по порядку пару из заданного ассоциативного списка, у которой

первый элемент совпадает с заданным выражением. Если такой пары нет, то вырабатывается значение *NIL*.

- а) (SEXPR ASSOC (LAMBDA (X A) (COND
 ((NULL A)) NIL)
 ((EQUAL X (CAAR A)) (CAR A))
 (T (ASSOC X (CDR A))))))
 б) (SEXPR ASSOC (LAMBDA (X A) (PROG NIL
 A (COND ((NULL A) (RETURN NIL))
 ((EQUAL X (CAAR A))
 (RETURN (CAR A))))
 (SETQ A (CDR A)) (GO A))))

Пр и м е р ы:

(ASSOC (QUOTE B) (QUOTE ((A . D) (B . F)
 (C D C E)))
 → (B . F)
 (ASSOC (QUOTE (C . F))
 (QUOTE (((A B) . D) ((C . F) E G))))
 → ((C . F) E G)
 (ASSOC (QUOTE C) (QUOTE ((A . D) (B . F)
 (C D C E))))
 → (C D C E)
 (ASSOC (QUOTE C)
 (QUOTE (((A B) . D) ((C . F) E G))))
 → NIL

Функция PAIRLIS подобна функции *PAIR* и отличается от нее лишь тем, что она не создает новый ассоциативный список, а добавляет новые пары к существующему списку.

- а) (SEXPR PAIRLIS (LAMBDA (X Y A) (COND
 ((NULL X) A)
 (T (CONS (CONS (CAR X) (CAR Y))
 (PAIRLIS (CDR X) (CDR Y) A))))))
 б) (SEXPR PAIRLIS (LAMBDA (X Y A) (PROG (U)
 B (COND ((NULL X) (GO C)))
 (SETQ U (CONS (CONS (CAR X) (CAR Y)) U))
 (SETQ X (CDR X)) (SETQ Y (CDR Y)) (GO B)
 C (COND ((NULL U) (RETURN A)))
 (SETQ A (CONS (CAR U) A) (SETQ U (CDR U))
 (GO C))))

Примеры:

```
(PAIRLIS (QUOTE (A B D))
          (QUOTE (D ((A B) D) C))
          (QUOTE ((A . D) (B . F) (C D C E))) )
→ ((A . D) (B (A B) D) (D . C) (A . D) (B . F) (C D C E))
(PAIRLIS (QUOTE (D ((A B) D) C)) (QUOTE (A B D))
          (QUOTE (((A B) . D) ((C . F) E G))) )
→ ((D . A) (((A B) D) . B) (C . D) ((A B) . D) ((C . F) E G))
```

Функция SUBST подставляет в заданное выражение z (в значение ее третьего аргумента) выражение x (значение первого аргумента) вместо всех подвыражений, совпадающих со значением y второго аргумента (на какой бы глубине они ни находились). Результат подстановки выдается в качестве значения функции.

```
(SEXPR SUBST (LAMBDA (X Y Z) (COND
  ((EQUAL Y Z) X) ((ATOM Z) Z)
  (T (CONS (SUBST X Y (CAR Z))
            (SUBST X Y (CDR Z)) )) ))
```

Примеры:

```
(SUBST (QUOTE X) (QUOTE B)
        (QUOTE ((A B) D ((A B) D) C)) )
→ ((A X) D ((A X) D) C)
(SUBST (QUOTE (A . B)) (QUOTE (A B))
        (QUOTE ((A B) D ((A B) D) C)) )
→ ((A . B) D ((A . B) D) C)
(SUBST (QUOTE (A B D)) (QUOTE D)
        (QUOTE (A ((B . D) C) . F)) )
→ (A ((B A B D) C) . F)
```

Функция SUBLIS в заданном выражении y (значении ее второго аргумента) заменяет все входящие в него атомы, которым в заданном ассоциативном списке (значении первого аргумента) поставлены в соответствии некоторые выражения, этими выражениями. Преобразованное выражение выдается в качестве значения функции. Функция **SUBLIS** обращается к функции **SUB2**, которая для заданного атома (значения второго аргумента) выдает в качестве значения либо выражение, поставленное ему в соответствие в заданном ассоциативном списке, либо сам этот атом, если такого выражения нет.


```
(SEXPR SUBLIS (LAMBDA (A Y) (COND
  ((NULL Y) NIL) ((ATOM Y) (SUB2 A Y))
  (T (CONS (SUBLIS A (CAR Y))
            (SUBLIS A (CDR Y)) ) ) )))
(SEXPR SUB2 (LAMBDA (A Y) (COND
  ((NULL A) Y)
  ((EQ Y (CAAR A)) (CDAR A))
  (T (SUB2 (CDR A) Y)) )))
```

П р и м е р ы:

```
(SUBLIS (QUOTE ((A . D) (B . F) (C D C E)))
  (QUOTE ((A B) D ((A B) D) C)) )
→ ((D F) D ((D F) D) (D C E))
(SUBLIS (QUOTE (((A B) . D) ((C . F) E G)))
  (QUOTE ((A B) ((C . F) (D (A B)))))) )
→ ((A B) ((C . F) (D (A B))))
```

Функция SASSOC подобна функции ASSOC и отличается от нее тем, что, когда в заданном ассоциативном списке не найдено никакого соответствия для заданного выражения, в качестве значения выдается результат обращения к функции (без аргументов), наименование или определяющее выражение которой задано в качестве третьего аргумента функции SASSOC.

```
а) (SEXPR SASSOC (LAMBDA (X A F) (COND
  ((NULL A) (F))
  ((EQUAL X (CAAR A)) (CAR A))
  (T (SASSOC X (CDR A) F)) )))
б) (SEXPR SASSOC (LAMBDA (X A F) (PROG NIL
  A (COND ((NULL A) (RETURN (F)))
  ((EQUAL X (CAAR A))
    (RETURN (CAR A))) )
  (SETQ A (CDR A)) (GO A) )))
```

Если в первых трех примерах обращения к функции ASSOC изменить наименование функции на SASSOC и добавить произвольный третий аргумент, то результат обращения не изменится, например,

```
(SASSOC (QUOTE (C . F))
  (QUOTE (((A B) . D) ((C . F) E G)))
  (FUNCTION (LAMBDA NIL (QUOTE FAIL)))) )
→ ((C . F) E G)
```

В четвертом примере при такой замене произойдет обращение к заданной функции

```
(SASSOC (QUOTE C)
  (QUOTE (((A B) . D) ((C . F) E G)))
  (FUNCTION (LAMBDA NIL (QUOTE FAIL)))) )
→ FAIL
```

3.8. Функционалы

В этом разделе собраны функции, характерной чертой которых является поочередное применение функции, заданной в качестве функционального аргумента данной функции, к некоторой последовательности выражений.

Функция MAPLIST применяет функцию, заданную в качестве ее второго аргумента (функционального), последовательно ко всему списку, заданному в качестве значения первого аргумента, и ко всем спискам, поочередно получаемым отбрасыванием первого элемента от предыдущего списка. Значением функции **MAPLIST** является список полученных результатов.

```
(SEXPR MAPLIST (LAMBDA (X F) (COND
  ((NULL X) NIL)
  (T (CONS (F X) (MAPLIST (CDR X) F))) )))
```

Пр и м е р ы:

```
(MAPLIST (QUOTE (A B C)) (FUNCTION CDR))
→ ((B C) (C) NIL)
(MAPLIST (QUOTE (1 —2 3))
  (FUNCTION (LAMBDA (X) (MINUS (CAR X)))))
→ (—1 2 —3)
```

Функция MAPCAR подобна функции **MAPLIST** и отличается от нее тем, что заданная функция применяется к первым элементам тех списков, к которым она применялась бы в случае обращения к **MAPLIST**.

- а) (SEXPR MAPCAR (LAMBDA (X F) (COND
 ((NULL X) NIL)
 (T (CONS (F (CAR X)) (MAPCAR (CDR X) F))))))
- б) (SEXPR MAPCAR (LAMBDA (X F)
 (MAPLIST X (FUNCTION (LAMBDA (U)
 (F (CAR U)))))))

Примеры:

(MAPCAR (QUOTE (A B C)) (FUNCTION LIST))

→ ((A) (B) (C))

(MAPCAR (QUOTE (1 2 3 4))

(FUNCTION (LAMBDA (X) (TIMES X X))))

→ (1 4 9 16)

Функция MAP имеет смысл, если функция *F*, заданная в качестве ее второго аргумента, обладает каким-либо побочным эффектом. Эта функция последовательно применяется к тем же аргументам, что и в случае функции *MAPLIST*, но выработанные значения никак не используются и не сохраняются. Значение функции *MAP* равно *NIL* (а если очередной аргумент, подготовленный для обращения к функции *F*, окажется атомом, то этому атому).

(SEXPR MAP (LAMBDA (X F) (PROG NIL

A (COND ((ATOM X) (RETURN X)))

(F X) (SETQ X (CDR X)) (GO A))))

Пример:

(MAP (QUOTE (A B C)) (FUNCTION PRINT)) → NIL

При выполнении этого обращения будут отпечатаны три списка:

(A B C)

(B C)

(C)

3.9. Операторы

Для сокращения записи многих операторов в аппарате *PROG* можно прибегнуть к помощи функций *PUSH*, *POPUP* и *POP*. Если ввести их в программу с помощью тех определений, которые даны ниже, то они будут работать правильно, но выполнение программы замедлится. Целесообразно реализовать их в виде встроенных функций.

Функция PUSH. Обращение к функции *PUSH* вида

(PUSH *e v*)

где *e* — некоторое выражение, а *v* — переменная, в точности эквивалентно выражению

(SETQ *v* (CONS *e v*))

О п р е д е л е н и е:

```
(SFEXPR PUSH (LAMBDA (*1 *2)
  (EVAL (LIST (QUOTE SETQ) *2
    (LIST (QUOTE CONS) *1 *2) )) ))
```

Несколько экзотический выбор обозначений для аргументов связан с тем, что функция, введенная этим определением, будет работать неверно, если обозначения ее связанных переменных совпадут с обозначением переменной *v* или переменных, фигурирующих в выражении *e*.

Функция POPUP. Обращение к функции *POPUP* вида
(*POPUP u v*)

где *u, v* — переменные, в точности эквивалентно последовательному выполнению двух операторов

```
(SETQ u (CAR v)) (SETQ v (CDR v))
```

О п р е д е л е н и е:

```
(SFEXPR POPUP (LAMBDA (*1 *2) (PROG NIL
  (EVAL (LIST (QUOTE SETQ) *1
    (LIST (QUOTE CAR) *2)))
  (EVAL (LIST (QUOTE SETQ) *2
    (LIST (QUOTE CDR) *2) )) )))
```

Функция POP. Обращение к функции *POP* вида
(*POP v*)

где *v* — переменная, в точности эквивалентно выражению

```
(SETQ v (CDR v))
```

О п р е д е л е н и е:

```
(SFEXPR POP (LAMBDA (*1)
  (EVAL (LIST (QUOTE SETQ) *1
    (LIST (QUOTE CDR) *1) )) ))
```

П р и м е р. Перепишем описание функции *FLATTEN* из разд. 3.1, используя операторы *PUSH*, *POPUP* и *POP*, а также форму условного оператора *COND*, описанную в разд. 2.9.

```
(SEXPR FLATTEN (LAMBDA (X) (PROG (U V)
  A (COND ((NULL X))
    ((ATOM X) (PUSH X U))
    (T (PUSH (CAR X) V) (POP X) (GO A)) )
  (COND ((NULL V) (RETURN U)))
  (POPUP X V) (GO A) )))
```

3.10. Списки свойств

В этом разделе будут описаны функции, исследующие и преобразующие списки свойств атомов. В разд. 2.2 были описаны две возможные структуры списка свойств. При второй структуре стандартные свойства (*EXPR*, *FEXPR*, *SUBR*, *FSUBR*, *APVAL*, *FIX*, *BITS*) требуют особой трактовки и многие функции, описанные в настоящем разделе, неприменимы к атомам, наделенным одним из стандартных свойств. Однако функции *DEFINE*, *DEFLIST*, *PUTPROP* можно применять в любом случае (в реализации ЛИСПа на машине БЭСМ-6 функция *DEFINE* с небольшим отличием встроена в систему).

Большинство функций данного раздела используют функции *PROPLIST*, имеющую своим значением список свойств атома, заданного в качестве ее аргумента, и *SPROPL*, заменяющую список свойств данного атома значением второго аргумента функции. Если доступ к списку свойств открывается через *d*-указатель информационной ячейки атома, как это описано в разд. 2.2 (первый вариант), то эти функции могут быть определены выражениями

$$\begin{aligned} & (SEXPR PROPLIST (LAMBDA (A) (CDR A))) \\ & (SEXPR SPROPL (LAMBDA (A X) (RPLACD A X))) \end{aligned}$$

Иначе говоря, функция *PROPLIST* идентична функции *CDR*, а функция *SPROPL* — функции *RPLACD*, но применяются они, когда значение аргумента *A* — атом (если реализация это позволяет). С учетом этого замечания можно упростить определения некоторых функций данного раздела, как это показало ниже на примере функции *REMPROP*.

Функции этого раздела составлены в расчете на то, что индикаторы — это атомы, а свойства — выражения. В крайнем случае допускаются свойства, представленные отдельными атомами, но тогда эти атомы должны быть отличны от любого из индикаторов.

Допускается также включать в списки свойств индикаторы, за которыми не следуют свойства (т. е. не следует ничего или следует другой индикатор). Такие индикаторы называются *флагами*. Для работы с флагами предназначены функции, описанные в конце раздела.

Функция *PUTPROP* помещает в список свойств атома, указанного в качестве ее первого аргумента, свойство, представленное значением третьего аргумента, с индикатором, заданным в виде значения второго аргумента. Особо выделяются случаи стандартных индикаторов *EXPR*, *FEXPR* и *APVAL*, для которых формируются

обращения к соответствующим встроенным функциям.

```
(SEXPR PUTPROP (LAMBDA (A I P) (PROG (U V)
(COND ((EQ I (QUOTE EXPR)) (RETURN
(EVAL (LIST (QUOTE SEXPR) A P)) ))
((EQ I (QUOTE FEXPR)) (RETURN
(EVAL (LIST (QUOTE SFEXPR) A P)) ))
((EQ I (QUOTE APVAL)) (RETURN
(EVAL (LIST (QUOTE CSETQ) A
(LIST (QUOTE
(QUOTE) P))) )))
(SETQ U (CONS NIL (PROPLIST A))) (SETQ V U)
A (COND ((NULL (CDR V)) (RPLACD V (LIST I P)))
((EQ (CAR (SETQ V (CDR V))) I) (COND
((NULL (CDR V))
(RPLACD V (LIST P)))
(T (RPLACA (CDR V) P)) ))
(T (GO A)) )
(SPROPL A (CDR U)) (RETURN A) )))
```

В качестве результата функция *PUTPROP* вырабатывает атом, список свойств которого подвергся изменению. Если в списке свойств этого атома уже было свойство с данным индикатором, то это свойство замещается новым, индикатор не дублируется.

Функция *DEFINE* предназначена для одновременного определения нескольких функций класса *EXPR*. Ее аргументом должен быть список вида

$$((f_1 e_1) \dots (f_n e_n))$$

где f_1, \dots, f_n — атомы, которые должны стать наименованиями функций, а e_1, \dots, e_n — соответствующие определяющие выражения. Одно такое обращение к функции *DEFINE* заменяет n обращений к *SEXPR*:

$$(SEXPR f_1 e_1) \dots (SEXPR f_n e_n)$$

которые формируются и выполняются в результате данного обращения к *DEFINE*. Результатом обращения является список наименований функций $(f_1 \dots f_n)$

```
(SFEXPR DEFINE (LAMBDA (X) (COND
((NULL X) NIL)
(T (CONS (EVAL (LIST (QUOTE SEXPR)
(CAAR X) (CADAR X) ))
(EVAL (LIST (QUOTE DEFINE)
(CDR X) )) )) )))
```

Функция *DEFLIST* похожа на функцию *DEFINE*. Ее первый аргумент имеет такой же вид, как и аргумент *DEFINE*, а вторым аргументом должен быть некоторый атом (сам атом, а не выражение, имеющее этот атом своим значением, так как функция *DEFLIST* — специальная). Этот атом заносится в списки свойств атомов f_i для $i = 1, \dots, n$ вместе с выражением e_i в качестве соответствующего свойства. Одно обращение к *DEFLIST* вида

$$(DEFLIST ((f_1 e_1) \dots (f_n e_n)) ind)$$

заменяет n обращений к *PUTPROP* вида

$$(PUTPROP (QUOTE f_i) (QUOTE ind) (QUOTE e_i))$$

для $i = 1, \dots, n$.

$$\begin{aligned} & (SEXPR DEFLIST (LAMBDA (X I) (COND \\ & \quad ((NULL X) NIL) \\ & \quad (T (CONS (PUTPROP (CAAR X) I (CADAR X)) \\ & \quad \quad (EVAL (LIST (QUOTE DEFLIST) \\ & \quad \quad \quad (CDR X) I))))))) \end{aligned}$$

Функция *GETPROP ** извлекает из списка свойств данного атома (значения первого аргумента) свойство с данным индикатором (значением второго аргумента). Если такого индикатора в списке свойств нет, то вырабатывается значение *NIL*.

$$\begin{aligned} & (SEXP GETPROP (LAMBDA (A I) (PROG (U) \\ & \quad (SETQ U (PROPLIST A)) \\ & \quad A (COND ((NULL U) (RETURN NIL)) \\ & \quad \quad ((EQ (CAR U) I) (GO B))) \\ & \quad (SETQ U (CDR U)) (GO A) \\ & \quad B (RETURN (COND ((NULL (CDR U)) NIL) \\ & \quad \quad (T (CADR U))))))) \end{aligned}$$

Функция *GETPROP* родственна функции *get*, которая использовалась в разд. 2.9. Разница между ними заключается в том, что значением *GETPROP* является свойство с данным индикатором (если оно есть у данного атома), а функция *get* присваивает это свойство переменной, указанной в качестве ее третьего аргумента, значением же функции является константа *T* или *NIL* в зависимости

*) Эта функция не рассчитана на работу со списками свойств, в которых стандартные свойства (см. разд. 2.2) трактуются особым образом.

от наличия или отсутствия у атома свойства с данным индикатором.

Функция *GETPROP* в описанном выше виде обладает одним недостатком. Если данному индикатору сопоставлено свойство со значением *NIL*, то это нельзя установить по значению функции. Если изменить определение функции так, чтобы она имела своим значением список, единственным элементом которого является искомое свойство, то можно распознавать и такое свойство. Для этого следует в последней строчке описания функции заменить (*CADR U*) на (*LIST (CADR U)*).

Не следует обращаться к функции *GETPROP* со вторым аргументом, имеющим значение *SUBR* или *FSUBR*. Значением функции будет при этом адрес машинной подпрограммы, который нельзя ни подвергнуть дальнейшей обработке средствами языка, ни даже отпечатать.

Функция *PROP* отличается от функции *GETPROP* тем, что она выдает в качестве результата не свойство, связанное с данным индикатором (если оно есть в списке), а весь хвост списка свойств, следующий за найденным индикатором. Если же индикатор в списке свойств не обнаруживается, то производится обращение к функции без аргументов, наименование или определяющее выражение которой должно быть задано в качестве третьего аргумента *PROP*. Результат этого обращения выдается в этом случае в качестве результата обращения к *PROP*.

```
(SEXPR PROP (LAMBDA (A I F) (PROG (U)
  (SETQ U (PROPLIST A))
  A (COND ((NULL U) (RETURN (F)))
    ((EQ (CAR U) I) (RETURN (CDR U))) )
  (SETQ U (CDR U)) (GO A) )))
```

Пользоваться этой функцией следует с осторожностью, если среди свойств атома содержится свойство *SUBR* или *FSUBR* (см. замечание к описанию функции *GETPROP*).

Функция *REMPROP* *) удаляет из списка свойств данного атома (значения первого аргумента в обращении к ней) свойство, снабженное данным индикатором (значением второго аргумента), вместе с этим индикатором. Если такого индикатора не было, то

*) См. сноску на стр. 158.

список свойств не меняется.

```
(SEXPR REMPROP (LAMBDA (A I) (PROG (U V)
  (SETQ U (CONS NIL (PROPLIST A)))
  (SETQ V U)
  A (COND ((NULL (CDR V)) (RETURN A))
    ((EQ (CADR V) I) (GO B)) )
  (SETQ V (CDR V)) (GO A)
  B (COND ((NULL (CDDR V)) (RPLACD V NIL))
    (T (RPLACD V (CDDDR V))) )
  (SPROPL A (CDR U)) (RETURN A) )))
```

Значение функции совпадает со значением первого аргумента.

Если, как это описано в разд. 2.2, указатель на список свойств помещается в *d*-указателе информационной ячейки атома, то описание функции *REMPROP* может быть переписано так:

```
(SEXPR REMPROP (LAMBDA (A I) (PROG (V)
  (SETQ V A)
  A (COND ((NULL (CDR V)) (RETURN A))
    ((EQ (CADR V) I) (GO B)) )
  (SETQ V (CDR V)) (GO A)
  B (RPLACD V (COND ((NULL (CDDR V)) NIL)
    (T (CDDDR V))) )
  (RETURN A) )))
```

Аналогичные упрощения могут быть внесены в определения функций *PUTPROP* и *REMFLAG*.

Функция *PUTFLAG* *) помещает в начало списка свойств атома, заданного в качестве значения ее первого аргумента, флаг, представленный вторым аргументом. Если такой флаг уже был в списке свойств, то он дублируется.

```
(SEXPR PUTFLAG (LAMBDA (A I)
  (SPROPL A (CONS I (PROPLIST A))) ))
```

Предикат *FLAGP* *) проверяет, содержится ли в списке свойств данного атома указанный флаг.

```
(SEXPR FLAGP (LAMBDA (A I)
  (MEMBER I (PROPLIST A)) ))
```

Функция *FLAG* *) помещает данный флаг (значение второго аргумента) в списки свойств атомов, перечисленных в списке, являю-

*) См. сноску на стр. 158.

щемся значением первого аргумента обращения к функции.

```
(SEXPR FLAG (LAMBDA (L I) (PROG (U)
  A (COND ((NULL L) (RETURN NIL)))
    (SETQ U (PROPLIST (CAR L)))
    (COND ((NOT (MEMBER I U))
      (SPROPL (CAR L) (CONS I U)) ))
    (SETQ L (CDR L)) (GO A) )))
```

Функция REMFLAG *) удаляет из списков свойств всех атомов, являющихся элементами списка, заданного в виде значения первого аргумента обращения к REMFLAG, все вхождения флага, указанного в качестве значения второго аргумента.

```
(SEXPR REMFLAG (LAMBDA (L I) (PROG (U V)
  A (COND ((NULL L) (RETURN NIL)))
    (SETQ U (CONS NIL (PROPLIST (CAR L))))
    (SETQ V U)
  B (COND ((NULL (CDR V)) (GO D))
    ((EQ (CADR V) I) (GO C)) )
    (SETQ V (CDR V)) (GO B)
  C (RPLACD V (CDDR V)) (GO B)
  D (SPROPL (CAR L) (CDR U))
    (SETQ L (CDR L)) (GO A) )))
```

* См. сноску на стр. 158.

ПРИЛОЖЕНИЕ

Дадим пример программы, несколько более сложный, чем в разд. 1.38. Впрочем, решаемая задача — проверка общезначности формулы исчисления высказываний — достаточно элементарна. Несколько иной вариант программы для ее решения содержится в [15].

Формула — это либо логическая константа (T или NIL), либо переменная (любой атом, отличный от T и NIL), либо список, первый элемент которого — знак логической операции (NOT , AND , OR , IMP или EQU), а следующие один (для NOT) или два элемента — формулы, являющиеся операндами данной операции. Например, формула, которая в обычной записи имеет вид $\neg (X \supset Y) \equiv X \wedge \neg Y$, запишется так:

$$(EQU (NOT (IMP X Y)) (AND X (NOT Y)))$$

По определению формула общезначима, если она истинна при любых значениях переменных. Проверку на общезначимость будем вести, пытаясь построить контрпример — набор значений переменных, при котором данная формула принимает значение «ложь». Для этого будем расчленять формулу на подформулы и заносить их в два списка: список L будет содержать подформулы, которые в искомом контрпримере должны быть истинны, а список R — формулы, которые, подобно исходной формуле, должны быть ложны. Пара, составленная из списков L и R , называется секвенцией.

В некоторых случаях требование, чтобы формула была истинной (или ложной), однозначно определяет необходимые значения ее частей. Но в общем случае возможны варианты. Например, если мы расчленяем формулу списка L текущей секвенции и эта формула имеет вид $(OR F G)$, то следует рассмотреть два случая. В первом случае эта формула заменяется в списке L формулой F , во втором — формулой G . Список R в обоих случаях остается прежним. Одна из полученных секвенций становится текущей, другая заносится в список S нерассмотренных секвенций. Если

по текущей секвенции не удалось построить контрпример, то надо перейти к очередной секвенции из списка S . Если же он пуст, то вырабатывается результат T , означающий, что исходная формула общезначима.

Итак, основная функция *PROVE* нашей программы такова:

```
(SEXPR PROVE (LAMBDA (L R S) (COND
  (L (PROVE1 (CAR L) (CDR L) R S))
  (R (PROVE2 (CAR R) (CDR R) S))
  (S (PROVE (CAAR S) (CDAR S) (CDR S)))
  (T T) )))
```

Функция *PROVE1* анализирует первую формулу F списка L , если он не пуст, функция *PROVE2* — первую формулу списка R :

```
(SEXPR PROVE1 (LAMBDA (F L R S) (COND
  ((ATOM F) (SUBT F L R S))
  (T (PROVE3 (CAR F) (CDR F) L R S)) )))
(SEXPR PROVE2 (LAMBDA (F R S) (COND
  ((ATOM F) (SUBV NIL F R NIL NIL S))
  (T (PROVE4 (CAR F) (CDR F) R S)) )))
```

Функции *SUBT* и *SUBV* упрощают текущую секвенцию в случае, когда обнаружилась переменная, которая в искомом контрпримере должна принимать значение T или NIL соответственно. Их мы опишем ниже.

Функциям *PROVE3* и *PROVE4* передаются для анализа знак операции OP формулы F , список A ее операндов и остальные аргументы в прежнем виде:

```
(SEXPR PROVE3 (LAMBDA (OP A L R S) (COND
  ((EQ OP (QUOTE NOT))
    (PROVE L (CONS (CAR A) R) S))
  (T (PROVE5 OP) (CAR A) (CADR A) L R S)) )))
(SEXPR PROVE4 (LAMBDA (OP A R S) (COND
  ((EQ OP (QUOTE NOT)) (PROVE A R S))
  (T (PROVE6 OP (CAR A) (CADR A) R S)) )))
```

Эти функции выделяют случай одноместной операции *NOT*, перебрасывая ее операнд в другую часть секвенции. Если операция OP — двухместная, то функции *PROVE5* или *PROVE6* передаются ее операнды F и G . Работа этих функций управляется знаком операции OP (случай $OP \rightarrow EQU$ выделяется неявно).

```
(SEXPR PROVE5 (LAMBDA (OP F G L R S)
  (SELECTQ OP (AND (PROVE1 F (CONS G L) R S)
    OR (PROVE1 F L R
```

```

(CONS (CONS (CONS G L) R) S) )
IMP (PROVE1 G L R
(CONS (CONS L (CONS F R)) S) ) )
(PROVE1 F (CONS G L) R
(CONS (CONS L (CONS F (CONS G R))) S) ) )))
(SEXPR PROVE6 (LAMBDA (OP F G R S)
(SELECTQ OP (OR (PROVE2 F (CONS G R) S)
(AND (PROVE2 F R
(CONS (CONS NIL (CONS G R)) S) )
IMP (PROVE1 F NIL (CONS G R) S) )
(PROVE1 F NIL (CONS G R)
(CONS (CONS (LIST G) (CONS F R)) S) ) ) )))

```

Функция *SUBT* подготавливает подстановку значения *T* переменной *X* в текущую секвенцию. Для единообразия последующих действий все члены списка *L* переносятся со знаком *NOT* в список *R*:

```

(SEXPR SUBT (LAMBDA (X L R S) (COND
((NULL L) (SUBV T X R NIL NIL S))
(T (SUBT X (CDR L)
(CONS (LIST (QUOTE NOT)
(CAR L) ) R) S) ) )))

```

Функции *SUBV* и *SUBV1* преобразуют одну за другой формулы списка *R*, присваивая переменной *X* значение *V*. Преобразование одной формулы делает функция *SUB*. Если при этом формула упрощается до константы *NIL*, то она интереса уже не представляет, а если получается значение *T*, то контрпример построить невозможно и работа функции *SUBV1* завершается так же, как работа *PROVE* при пустых *L* и *R*. Если формула упростилась до переменной, то она заносится в список *LX*, а если нет — то в список *LF*. Если после преобразования всех формул список *LF* пуст, то контрпример построен и вычисление завершается со значением *NIL*. Если списки *LF* и *LX* не пусты, то начинается преобразование формул из списка *LF*, соответствующее присваиванию значения *NIL* первой переменной из списка *LX*. Если же *LX* пуст, то возобновляется работа функции *PROVE2* со списком *LF* в качестве *R*.

```

(SEXPR SUBV (LAMBDA (V X R LX LF S) (COND
(R (SUBV1 V X (SUB V X (CAR R))
(CDR R) LX LF S))
((NULL LF) NIL)
(LX (SUBV NIL (CAR LX) LF (CDR LX) NIL S))

```

```

(T (PROVE2 (CAR LF) (CDR LF) S)) )))
(SEXPR SUBV1 (LAMBDA (V X F1 R LX LF S)
(COND ((NULL F) (SUBV V X R LX LF S))
      ((EQ F T) (COND (S (PROVE (CAAR S)
                                (CDAR S) (CDR S) ))
                        (T T) ))
      ((ATOM F) (SUBV V X R (CONS F LX) LF S))
      (T (SUBV V X R LX (CONS F LF) S)) )))

```

Функция *SUB* выделяет случай атомарной формулы *F* и завершает ее преобразование:

```

(SEXPR SUB (LAMBDA (V X F) (COND
  ((EQ X F) V) ((ATOM F) F)
  (T (SUBA V X (CAR F)
    (SUB V X (CADR F)) (CDDR F) )) )))

```

Функция *SUBA* отделяет случай формулы со знаком *NOT* от формул с двухместной операцией.

```

(SEXPR SUBA (LAMBDA (V X OP F A) (COND
  ((EQ OP (QUOTE NOT)) (NEG F))
  (T (SUBB OP F (SUB V X (CAR A)))) )))

```

Функции *NEG* и *SUBB* завершают преобразование формулы, члены которой уже преобразованы:

```

(SEXPR NEG (LAMBDA (F) (SELECTQ F
  (NIL T T NIL) (LIST (QUOTE NOT) F) )))
(SEXPR SUBB (LAMBDA (OP F G) (COND
  ((EQ F T) (COND
    ((OR (EQ G T) (EQ OP (QUOTE OR))) T)
    ((NULL G) NIL) (T G) ))
  ((NULL F) (SELECTQ OP
    (OR G AND NIL IMP T) (NEG G) ))
  ((EQ G T) (SELECTQ OP
    (OR T IMP T) F))
  ((NULL G) (SELECTQ OP
    (OR F AND NIL)
    (LIST (QUOTE NOT) F) ))
  (T (LIST OP F G)) )))

```

Функция *CHECK*, с обращения к которой начинается проверка формулы *F* на общезначимость, может быть описана так:

```

(SEXPR CHECK (LAMBDA (F)
  (PROVE2 F NIL NIL) ))

```

ЛИТЕРАТУРА

1. Алгоритмический язык АЛГОЛ 68. Пер. с англ. Под общ. ред. А. П. Ершова. — Кибернетика, 1969, № 6, с. 17—144; 1970, № 1, с. 13—160.
2. Базисный рефал. Описание языка и основные приемы программирования (Методические рекомендации). — М.: ЦНИПИАСС, 1974. (Фонд алгоритмов и программ для ЭВМ (в отрасли «Строительство»). Вып. V—33).
3. Баррон Д. Рекурсивные методы в программировании. Пер. с англ. — М.: Мир, 1974.
4. Вирт Н. Язык программирования Паскаль (пересмотренное сообщение). Пер. с англ. — В сб.: Алгоритмы и организация решения экономических задач, вып. 9. — М.: Статистика, 1977, с. 52—86.
5. Кнут Д. Искусство программирования для ЭВМ, т. I. Основные алгоритмы. Пер. с англ. — М.: Мир, 1976.
6. Лавров С. С., Гончарова Л. И. Автоматическая обработка данных. Хранение информации в памяти ЭВМ. — М.: Наука, 1971.
7. Лавров С. С., Силагадзе Г. С. Входной язык и интерпретатор системы программирования на базе языка ЛИСП для машины БЭСМ-6. — М.: ИТМ и ВТ АН СССР, 1969.
8. Маурер У. Введение в программирование на языке ЛИСП. Пер. с англ. — М.: Мир, 1976.
9. Фостер Дж. Обработка списков. Пер. с англ. — М.: Мир, 1974.
10. Юфа В. М. — О новых функциях в системе ЛИСП — БЭСМ-6. В сб. «Обработка символьной информации», вып. 4, М.: Вычислительный центр АН СССР, 1978, с. 26—50.
11. Юфа В. М. Развитие системы хранения информации в ЭВМ. — М.: ИТМ и ВТ АН СССР, 1970.
12. Berkeley E. C., Bobrow D. G. The programming language LISP. — Cambridge, Mass.: Information International, 1964.

13. H e w i t t C. Planner: a language for proving theorems in robots, Proc. Intern. Joint Conf. on Artif. Intell. — Bedford, Mass.: Mitre Corp., 1969, p. 295—301.
14. H e w i t t C. Procedural embedding of knowledge in PLANNER, Proc. Second Intern. Joint Conf. on Artif. Intell. — London: 1971, p. 167—182.
15. M c C a r t h y J. et al. LISP 1.5 programmer's manual. — Cambridge, Mass.: M. I. T. Press, 1963.
16. M c C a r t h y J. Recursive functions of symbolic expressions and their computation by machine, p. 1, Comm. ACM 3, 1960, № 4, p. 184—195.
17. S u s s m a n G. J., M c D e r m o t t D. V. From PLANNER to CONNIVER: a genetic approach, Proc. FJCC, 1972, p. 1171—1180.

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

- Адрес возврата 75, 98, 99, 103, 125
Адрес выражения 62
Активная связь переменной, см. Активное значение
Активное значение 34, 75
Алфавит 8
Аргументы обращения к функции 11—13, 22, 24, 25, 51, 87, 113
Арифметическая функция 46—48, 97
Арифметический предикат 48, 49, 98
Ассоциативный список 74, 75, 83—85, 88, 90, 93, 94, 103, 110—113, 121, 123, 148
Атом 8—10, 18, 19, 35, 38, 44, 62, 63, 66—68, 78, 80, 81, 103, 112

Базовый лисп 30
Блок выхода из подпрограммы 99—101
Буква 8
Буфер ввода 74, 80, 81
— внешних наименований 74, 80
— вывода 74, 78, 79

Внешнее представление атома 66, 74, 78
Внешняя переменная 112
Внутреннее представление выражений 62, 133
Внутренняя переменная 112
Восьмеричное число 44, 45, 50, 51
Встроенная константа 9, 11, 16, 18, 67
— функция 9, 23, 67, 87, 115
Выражение 11, 14, 20, 30, 37, 39, 41, 62

Десятичное число, см. Целое число

Значение 11, 13, 14, 22, 25, 29, 30, 34, 35, 39—44, 51, 52, 73—75, 83, 84, 97

Индикатор 66, 86, 89, 156—161
Интерпретатор 85
Интерпретация 85—98, 121—124
Информационная ячейка атома 62—68, 75, 97, 103, 106
Исполнительная часть рабочей подпрограммы 113—115, 121

Класс функции, см. Свойство функции
Компилятор 75, 100, 110—127
Компиляция 33, 85, 107, 110, 121—124
Композиции *CAR* и *CDR* 15
Константа 9, 11, 16, 18, 29, 44, 67, 103, 122, 123
Константа-магазин 123

Лисповская ячейка, см. Списочная ячейка
Литера 8, 66, 78, 81
Логическое выражение 18, 20, 23, 32, 41
— значение 18, 23

Магазин 74, 75, 88, 99, 100, 103, 125
Массив вынесенных указателей 103, 107, 115, 120
Масштабный множитель 45
Метка 31, 32, 90, 91, 126
Мусорщик 72, 84, 102

Наименование функции 11, 22, 28, 51, 52, 93, 113, 157

Область машинных программ 74, 75
— полных слов 74, 102, 107—110
— списочной памяти 64, 72, 74, 75, 102, 103, 107—109
Обращение к функции 11, 12, 22, 25, 32, 33, 86, 113
Обычная функция 24, 67, 110
Оглавление списка объектов 76, 106
Ограничитель 8, 78, 80
Оператор 31, 90, 91
— выхода 32, 91
— перехода 31, 91
— присваивания 33
Определяемая функция 23, 67, 103
Определяющее выражение функции 11, 21—25, 33, 51, 52, 87, 93, 110, 113, 115, 157
Особая литера 81
Относительный адрес 111
Очередь 134

Пара 42, 62, 63
Параметр подпрограммы 70
Паспорт участка 108, 126
Пассивная связь переменной, см.

Пассивное значение
Пассивное значение 34, 75
Переменная 9, 11, 22, 33—35, 51, 52, 75, 83—85, 97, 110—112
Подготовительная часть рабочей подпрограммы 113—115, 121
Подпрограмма выражения 112
— реализации 70, 74, 85, 98
n-поле 108

Предикат 18, 41, 45
Преемник оператора 32
Признак атома, см. Свойство атома
Пробел (литера) 8, 10, 12, 43, 44, 79, 80

Программа 30, 38, 85
Программная переменная 31, 33—35, 90, 111
Прочие литеры 8
Пустой список 10, 16, 83

Рабочая подпрограмма 110—121, 126
Разметка списочной структуры 102—107

b-разряд 105
m-разряд 105
Рекурсивная подпрограмма 98
— функция 27

Сборка мусора 102, 107—110
Свободная переменная 52

Свойство атома 23, 65, 67, 77, 86, 113, 156
— функции 23, 65, 67, 87, 113—115, 122
Связанная переменная 22, 29, 34, 51, 110

Символ **function** 69

Скобки 10, 12, 43, 44, 79
Специальная функция 24, 67, 111
Список 10—16, 43, 64, 87
— объектов 64, 74, 75, 81, 102, 106
— свойств атома 65—68, 86, 87, 89, 103, 106, 156—161

Списочная структура 65, 82, 83, 102—106

— ячейка 62—65, 105

Стандартная функция, см. Встроенная функция

Строка битов, см. Восьмеричное число

Тело определяющего выражения 22, 33, 110, 111, 116

Точечная запись выражения 42—44, 64

Уборка мусора 65, 72, 75, 102—110, 125

a-указатель 62, 63, 70—73, 104, 105

d-указатель 62, 63, 70—73, 104, 105

Уплотнение занятой памяти 103

Управляющая литера 81

— программа 74

Условное выражение 20, 28, 91

Условный оператор 32, 91

Флаг 156

Функционал 51, 52, 84, 93, 94, 97

Функциональная переменная 51, 52, 94, 120—122

Функциональный аргумент 51, 93, 97, 120, 122, 125

Функция 9, 11, 12, 21

— расстановки 76—78

Целое число 44—51

Цифра 8

Число 9, 44, 68, 77, 78, 80, 97

Элемент списка 10, 12, 13, 15

Элементарный лисп, см. Базовый лисп

УКАЗАТЕЛЬ ЛИСПОВСКИХ ФУНКЦИЙ И КОНСТАНТ

Ниже перечислены все упоминаемые в этой книге атомы с указанием их свойств (классов). Для более полной классификации используется признак * *FSUBR* для атома, встречающегося в позиции наименования функции, но не являющегося им в действительности (например, *LAMBDA*). Признаком *IND* помечены атомы, используемые как индикаторы (наименования свойств).

Функции, описанные в главе 3, условно отнесены к классам *EXPR* и *FEXPR*. В действительности в ряде реализаций они могут быть встроенными. В то же время не все функции, отнесенные к классам *SUBR* или *FSUBR*, могут быть реализованы. Для функций указано также число их аргументов, причем буква *A* означает, что это число может быть произвольным.

Атом	Класс	Число аргументов	Страницы
<i>ADDIFNONE</i>	<i>EXPR</i>	2	131
<i>ADD1</i>	<i>SUBR</i>	1	48
<i>AND</i>	<i>FSUBR</i>	<i>A</i>	40, 89, 118
<i>APPEND</i>	<i>EXPR</i>	2	129
<i>APVAL</i>	<i>IND</i>	—	67, 86, 122, 156
<i>ASSOC</i>	<i>EXPR</i>	2	85, 149
<i>ATOM</i>	<i>SUBR</i>	1	18, 71, 116
<i>ATOMLIST</i>	<i>EXPR</i>	1	138
<i>ATTACH</i>	<i>EXPR</i>	2	133
<i>BITS</i>	<i>IND</i>	—	68, 77, 86
<i>BITSP</i>	<i>SUBR</i>	1	46
<i>BLANK</i>	<i>APVAL</i>	—	79
<i>CAR</i>	<i>SUBR</i>	1	13, 17, 42, 70, 71, 116
<i>CART</i>	<i>EXPR</i>	2	148
<i>CDR</i>	<i>SUBR</i>	1	14, 17, 41, 42, 71, 116
<i>COLLECT</i>	<i>EXPR</i>	1	131
<i>COMMA</i>	<i>APVAL</i>	—	80
<i>COMP</i>	<i>IND</i>	—	113—115
<i>COND</i>	<i>FSUBR</i>	<i>A</i>	20, 28, 32, 89, 91, 116, 155
<i>CONS</i>	<i>SUBR</i>	2	16, 41
<i>COPY</i>	<i>EXPR</i>	1	123
<i>CSETQ</i>	<i>FSUBR</i>	2	29

Атом	Класс	Число аргументов	Страницы
<i>CYCLEP</i>	<i>EXPR</i>	1	137
<i>CYCLE1</i>	<i>EXPR</i>	3	137
<i>DEFINE</i>	<i>FEXPR</i>	1	156, 157
<i>DEFLIST</i>	<i>FEXPR</i>	2	156, 158
<i>DIFFERENCE</i>	<i>SUBR</i>	2	47
<i>DIFFLIST</i>	<i>EXPR</i>	2	145
<i>DPAIR</i>	<i>EXPR</i>	2	149
<i>DREMOVE</i>	<i>EXPR</i>	1	136
<i>DREVERSE</i>	<i>EXPR</i>	1	134
<i>EFFACE</i>	<i>EXPR</i>	2	135
<i>EQ</i>	<i>SUBR</i>	2	19, 28, 44, 49, 72, 116
<i>EQSIGN</i>	<i>APVAL</i>	—	80
<i>EQUAL</i>	<i>SUBR</i>	2	28
<i>EQUALSET</i>	<i>EXPR</i>	2	147
<i>EVAL</i>	<i>SUBR</i>	1	39, 85
<i>EXPR</i>	<i>IND</i>	—	24, 25, 30, 67, 87, 113, 156
<i>EXPT</i>	<i>SUBR</i>	2	47
<i>FCOMP</i>	<i>IND</i>	—	113, 114
<i>FEXPR</i>	<i>IND</i>	—	24, 25, 30, 54, 67, 87, 97, 113, 156
<i>FFARG</i>	<i>*FSUBR</i>	2	97
<i>FIRST</i>	<i>EXPR</i>	2	141
<i>FIX</i>	<i>IND</i>	—	67, 77, 86
<i>FIXP</i>	<i>SUBR</i>	1	46
<i>FLAG</i>	<i>EXPR</i>	2	160
<i>FLAGP</i>	<i>EXPR</i>	2	160
<i>FLATTEN</i>	<i>EXPR</i>	1	132, 155
<i>FORALL</i>	<i>EXPR</i>	2	133
<i>FORODD</i>	<i>EXPR</i>	2	138
<i>FORSOME</i>	<i>EXPR</i>	2	138
<i>FSUBR</i>	<i>IND</i>	—	23, 67, 87, 116, 121
<i>FUNARG</i>	<i>*FSUBR</i>	2	93, 97
<i>FUNCTION</i>	<i>FSUBR</i>	1	51, 54, 93, 97, 120
<i>GENSYM</i>	<i>SUBR</i>	0	38
<i>GETPROP</i>	<i>EXPR</i>	2	158
<i>GO</i>	<i>FSUBR</i>	1	31, 91, 92, 119, 120, 123
<i>GREATERP</i>	<i>SUBR</i>	2	48
<i>GREQP</i>	<i>SUBR</i>	2	49
<i>INTERSECTION</i>	<i>EXPR</i>	2	147
<i>LABEL</i>	<i>FSUBR</i>	2	31
<i>LAMBDA</i>	<i>*FSUBR</i>	2	22, 25, 29, 87
<i>LAST</i>	<i>EXPR</i>	1	130
<i>LCYCLEP</i>	<i>EXPR</i>	1	136
<i>LCYCLE1</i>	<i>EXPR</i>	2	136
<i>LEFTSHIFT</i>	<i>SUBR</i>	2	51
<i>LENGTH</i>	<i>EXPR</i>	1	131
<i>LESSP</i>	<i>SUBR</i>	2	48
<i>LEXORDER</i>	<i>EXPR</i>	3	140
<i>LEXORDER1</i>	<i>EXPR</i>	3	141
<i>LIST</i>	<i>FSUBR</i>	A	40, 87

Атом	Класс	Число аргументов	Страницы
<i>LISTP</i>	<i>EXPR</i>	1	139
<i>LOGAND</i>	<i>FSUBR</i>	A	50
<i>LOGOR</i>	<i>FSUBR</i>	A	50
<i>LOGXOR</i>	<i>FSUBR</i>	A	50
<i>LPAR</i>	<i>APVAL</i>	—	79
<i>LUNION</i>	<i>EXPR</i>	1	146
<i>MAKESET</i>	<i>EXPR</i>	1	145
<i>MAP</i>	<i>EXPR</i>	2	154
<i>MAPCAR</i>	<i>EXPR</i>	2	53, 153
<i>MAPLIST</i>	<i>EXPR</i>	2	53, 153
<i>MAX</i>	<i>FSUBR</i>	A	48
<i>MEMB</i>	<i>EXPR</i>	2	27, 33
<i>MEMBER</i>	<i>SUBR</i>	2	28, 41
<i>MIN</i>	<i>FSUBR</i>	A	48
<i>MINUS</i>	<i>SUBR</i>	1	47
<i>MINUSP</i>	<i>SUBR</i>	1	49
<i>NCONC</i>	<i>EXPR</i>	2	134
<i>NIL</i>	<i>APVAL</i>	—	16, 20, 32, 35, 43, 64, 89
<i>NOT</i>	<i>SUBR</i>	1	23, 41, 118
<i>NULL</i>	<i>SUBR</i>	1	23, 72, 116, 118
<i>NUMBERP</i>	<i>SUBR</i>	1	45
<i>ONEP</i>	<i>SUBR</i>	1	49
<i>OR</i>	<i>FSUBR</i>	A	40, 90, 118
<i>ORDER</i>	<i>EXPR</i>	3	139
<i>ORDER1</i>	<i>EXPR</i>	3	143
<i>PAIR</i>	<i>EXPR</i>	2	148
<i>PAIRLIS</i>	<i>EXPR</i>	3	85, 150
<i>PERIOD</i>	<i>APVAL</i>	—	79
<i>PLUS</i>	<i>FSUBR</i>	A	46, 97
<i>PNAME</i>	<i>IND</i>	—	66
<i>POP</i>	<i>FEXPR</i>	1	120, 155
<i>POPUP</i>	<i>FEXPR</i>	2	120, 155
<i>POSSESSING</i>	<i>EXPR</i>	2	143
<i>PRINT</i>	<i>SUBR</i>	1	37, 78, 79
<i>PRIN1</i>	<i>SUBR</i>	1	79
<i>PROG</i>	<i>FSUBR</i>	A	31, 90—93, 111, 119, 123 154
<i>PROP</i>	<i>EXPR</i>	3	159
<i>PROPLIST</i>	<i>SUBR</i>	1	156
<i>PUSH</i>	<i>FEXPR</i>	2	120, 154
<i>PUTFLAG</i>	<i>EXPR</i>	2	160
<i>PUTPROP</i>	<i>EXPR</i>	3	156, 158, 160
<i>QUOTE</i>	<i>APVAL</i>	1	13, 16—18, 24, 39, 44, 52 68, 88, 107, 116
<i>QUOTIENT</i>	<i>SUBR</i>	2	47
<i>RANK</i>	<i>EXPR</i>	2	142
<i>READ</i>	<i>SUBR</i>	0	37, 80, 81
<i>REMAINDER</i>	<i>SUBR</i>	2	47
<i>REMFLAG</i>	<i>EXPR</i>	2	160, 161
<i>REMOVE</i>	<i>EXPR</i>	2	130

Атом	Класс	Число аргументов	Страницы
<i>REMOVEF</i>	<i>EXPR</i>	2	130
<i>REMPROP</i>	<i>EXPR</i>	2	159
<i>RETURN</i>	<i>SUBR</i>	1	32, 91, 92, 119, 120
<i>REVERSE</i>	<i>EXPR</i>	1	129
<i>REVERSE1</i>	<i>EXPR</i>	2	129
<i>RPAR</i>	<i>APVAL</i>	—	79
<i>RPLACA</i>	<i>SUBR</i>	2	73, 74, 116, 132
<i>RPLACD</i>	<i>SUBR</i>	2	73, 74, 116, 132
<i>SADD1</i>	<i>FSUBR</i>	1	98, 131
<i>SASSOC</i>	<i>EXPR</i>	3	152
<i>SELECTQ</i>	<i>FSUBR</i>	3	54
<i>SET</i>	<i>SUBR</i>	2	92
<i>SETOF</i>	<i>EXPR</i>	1	145
<i>SETQ</i>	<i>FSUBR</i>	2	29, 33—35, 92, 120, 154, 155
<i>SEXPR</i>	<i>FSUBR</i>	2	25, 89, 157
<i>SFEXPR</i>	<i>FSUBR</i>	2	25, 27, 39, 89
<i>SPROPL</i>	<i>SUBR</i>	2	156
<i>SSUB1</i>	<i>FSUBR</i>	1	98
<i>SUBLIS</i>	<i>EXPR</i>	2	151
<i>SUBR</i>	<i>IND</i>	—	23, 67, 87, 115, 121
<i>SUBSET</i>	<i>EXPR</i>	2	145
<i>SUBST</i>	<i>EXPR</i>	3	151
<i>SUB1</i>	<i>SUBR</i>	1	48
<i>SUB2</i>	<i>EXPR</i>	2	152
<i>SUCHTHAT</i>	<i>EXPR</i>	2	143
<i>SUCHTHAT1</i>	<i>EXPR</i>	4	143
<i>SUCHTHAT2</i>	<i>EXPR</i>	3	144
<i>T</i>	<i>APVAL</i>	—	18, 20
<i>TCONC</i>	<i>EXPR</i>	2	134
<i>TERPRI</i>	<i>SUBR</i>	0	79
<i>TIMES</i>	<i>FSUBR</i>	A	46
<i>UNION</i>	<i>EXPR</i>	2	146
<i>ZEROP</i>	<i>SUBR</i>	1	49, 98

УКАЗАТЕЛЬ ОБОЗНАЧЕНИЙ, ИСПОЛЬЗОВАННЫХ В ОПИСАНИИ РЕАЛИЗАЦИИ

Здесь приведены почти все идентификаторы, встречающиеся в главе 2. Они снабжены следующими признаками: *f* — функция, *p* — процедура, *v* — переменная, *a* — массив, *c* — константа, *l* — метка. Звездочкой помечены функции, процедуры и константы, имеющие соответствующее обозначение в языке.

Обозначение	Признак	Страницы
<i>abeg</i>	<i>c</i>	84
<i>aend</i>	<i>c</i>	84
<i>and</i>	<i>*f</i>	89
<i>apply</i>	<i>f</i>	85, 86, 94, 97
<i>aptr</i>	<i>v</i>	84, 88, 94, 111, 114, 121
<i>args</i>	<i>v</i>	115, 116
<i>assoc</i>	<i>f</i>	85, 94
<i>atom</i>	<i>*f</i>	72
<i>a1</i>	<i>v</i>	69, 70, 87, 100, 102
<i>a2</i>	<i>v</i>	69, 70, 87, 100
<i>base</i>	<i>v</i>	100, 111, 114, 115, 121
<i>blank</i>	<i>*c</i>	79
<i>bridge</i>	<i>p</i>	94
<i>calculate addresses</i>	<i>p</i>	108
<i>call</i>	<i>p</i>	88, 114, 115, 121, 125
<i>car</i>	<i>*f</i>	70
<i>cb</i>	<i>f</i>	105
<i>cdr</i>	<i>*f</i>	71
<i>cm</i>	<i>f</i>	105
<i>cn</i>	<i>f</i>	108
<i>collect</i>	<i>p</i>	102, 108
<i>comp</i>	<i>f</i>	77
<i>cond</i>	<i>*f</i>	89, 91, 92
<i>cons</i>	<i>*f</i>	72
<i>correct</i>	<i>p</i>	108—110
<i>eq</i>	<i>*f</i>	72
<i>eval</i>	<i>*f</i>	85, 121
<i>evatom</i>	<i>f</i>	85, 86, 112
<i>evcomp</i>	<i>p</i>	114, 121

Обозначение	Признак	Страницы
<i>evexpr</i>	<i>p</i>	114, 121
<i>evfarg</i>	<i>f</i>	86, 94, 97
<i>evfal</i>	<i>p</i>	121
<i>evffarg</i>	<i>f</i>	97
<i>evfun</i>	<i>f</i>	86, 87, 94
<i>evlam</i>	<i>f</i>	86—88
<i>evlis</i>	<i>f</i>	87
<i>false</i>	<i>l</i>	101
<i>funarg</i>	<i>*c</i>	93
<i>function</i>	<i>*f</i>	93, 97
<i>fwpt</i>	<i>v</i>	108
<i>get</i>	<i>f</i>	86, 158
<i>go</i>	<i>*f</i>	92
<i>h</i>	<i>c</i>	76
<i>hash</i>	<i>f</i>	76, 77
<i>hashtable</i>	<i>a</i>	76
<i>intern</i>	<i>f</i>	76, 77, 80, 81, 97
<i>jump</i>	<i>p</i>	100
<i>lablist</i>	<i>v</i>	90, 91
<i>lassoc</i>	<i>f</i>	91
<i>list</i>	<i>*f</i>	87
<i>lpar</i>	<i>*c</i>	79
<i>lptr</i>	<i>v</i>	109
<i>mark</i>	<i>p</i>	102
<i>markcell</i>	<i>p</i>	103
<i>marked</i>	<i>f</i>	103
<i>marklist</i>	<i>p</i>	103, 105
<i>markoblist</i>	<i>p</i>	106
<i>mbeg</i>	<i>c</i>	108
<i>memory</i>	<i>a</i>	109
<i>mend</i>	<i>c</i>	109
<i>move</i>	<i>p</i>	110
<i>newatom</i>	<i>t</i>	77
<i>nil</i>	<i>*c</i>	70, 94
<i>nillist</i>	<i>c</i>	90
<i>null</i>	<i>*f</i>	72
<i>pairlis</i>	<i>p</i>	84, 90, 94
<i>pairlist</i>	<i>p</i>	114
<i>pairlis2</i>	<i>p</i>	115
<i>pairlis3</i>	<i>p</i>	119
<i>period</i>	<i>*c</i>	79
<i>plus</i>	<i>*f</i>	97
<i>pnbuf</i>	<i>a</i>	74, 78, 80, 81
<i>print</i>	<i>*f</i>	79
<i>prin0</i>	<i>f</i>	79
<i>print</i>	<i>*f</i>	78, 79
<i>prog</i>	<i>*f</i>	90, 91
<i>put</i>	<i>p</i>	89
<i>quote</i>	<i>*f</i>	89
<i>r</i>	<i>v</i>	70, 100, 101, 112
<i>raddr</i>	<i>v</i>	100

Обозначение	Признак	Страниц
<i>rb</i>	<i>p</i>	105
<i>read</i>	<i>*f</i>	81
<i>reada</i>	<i>f</i>	80
<i>read0</i>	<i>f</i>	82
<i>read1</i>	<i>f</i>	81
<i>reallocate</i>	<i>p</i>	109
<i>reclaim</i>	<i>p</i>	72, 84, 102
<i>restore</i>	<i>p</i>	88, 99
<i>result</i>	<i>v</i>	88
<i>return</i>	<i>*f</i>	92
<i>rm</i>	<i>p</i>	105
<i>rpar</i>	<i>*c</i>	79
<i>rplaca</i>	<i>*f</i>	72
<i>rplacd</i>	<i>*f</i>	73
<i>r1, r2, ...</i>	<i>v</i>	70, 100
<i>sadd1</i>	<i>*f</i>	98
<i>save</i>	<i>p</i>	88, 99
<i>sbase</i>	<i>v</i>	125
<i>segment</i>	<i>f</i>	90
<i>set</i>	<i>*f</i>	93
<i>setq</i>	<i>*f</i>	92
<i>sexpr</i>	<i>*f</i>	89
<i>spread</i>	<i>p</i>	87, 88
<i>stack</i>	<i>a</i>	99
<i>stlist</i>	<i>v</i>	90
<i>switch</i>	<i>v</i>	91
<i>t</i>	<i>*c</i>	71
<i>terpri</i>	<i>*f</i>	78, 79
<i>true</i>	<i>l</i>	101
<i>val</i>	<i>f</i>	98
<i>vars</i>	<i>v</i>	115, 120
<i>zero</i>	<i>c</i>	98
<i>zerop</i>	<i>*f</i>	98