МОСКОВСКИЙ ОРДЕНА ЛЕНИНА, ОРДЕНА ОКТЯБРЬСКОЙ РЕВОЛЮЦИИ
И ОРДЕНА ТРУДОВОГО КРАСНОГО ЗНАМЕНИ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ имени М.В.ЛОМОНОСОВА

М.Ю.Семенов

ЯЗЫК ЛИСП ДЛЯ ПЕРСОНАЛЬНЫХ ЭВМ

Учебное пособие

ИЗДАТЕЛЬСТВО МОСКОВСКОГО УНИВЕРСИТЕТА

1989

ББК 22.14
С 30 УДК
519.68

Рецензенты:
кандидат физ.-мат. наук, доцент В.Н.Пильщиков
кандидат физ.-мат. наук В.М. Юфа

М.Ю.Семенов. С 30 Язык лисп для персональных ЭВМ. — М.: Изд-во Моск. ун-та, 1989. — 74 с.

В учебном пособии изложены основы программирования на языке лисп, используемого в символьных вычислениях и при решении задач искусственного интеллекта. Большое внимание уделяется построению рекурсивных функций.

Рассматриваются основные конструкции диалектов COMMON LISP, MULISP-85 и STANDARD LISP, реализованных на персональных компьютерах и миниЭВМ.

Книга ориентирована на студентов и аспирантов, занимающихся символьными вычислениями, компьютерной алгеброй, а также решением задач искусственного интеллекта.

077(02) - 89 - заказное **ББК 22.14**

ISBN 5-211-01773-0

ВВЕДЕНИЕ

Язык программирования лисп был создан в начале 60-х годов профессором Джоном Маккарти и его студентами из Массачусетского технологического института.

Основными объектами, с которыми оперирует лисп, являются списки. Отсюда происходит и название языка: словосочетание list processing означает "обработка списков".

В 60-е годы сформировался базовый диалект языка — LISP 1.5. Следует отметить, что, в отличие от других языков программирования (например, фортрана, паскаля), не существует стандарта языка лисп. Более того, различные диалекты лиспа, хотя во многом и похожи друг на друга, имеют незначительные синтаксические различия, которые не позволяют выделить подмножество, общее для всех диалектов. Вместе с тем в языке лисп имеются средства, позволяющие в рамках одного диалекта реализовать конструкции другого диалекта.

С развитием языка лисп появлялись новые диалекты. Так во второй половине 60-х годов был создан диалект STANDARD LISP. В 70-е годы появились достаточно развитые диалекты языка, такие, как MACLISP, INTERLISP. В начале 80-х годов был разработан диалект COMMON LISP.

Разнообразие диалектов не обошло стороной и персональные ЭВМ. Так, на персональных компьютерах IBM PC/XT и AT существуют следующие основные диалекты языка лисп: UO-LISP и AMI-LISP (близкие К STANDARD LISP), GOLDEN COMMON LISP (близкий к COMMON LISP) и MULISP-85. Кроме этого, имеются также реализации языка лисп в ряде специализированных систем. Рассмотрим основные достоинства языка лисп. Во-первых, лисп ориентирован на решение задач символьной обработки. Это удобно при выполнении формульных преобразований. Так, UO-LISP использовался для создания системы преобразования математических формул REDUCE.

В языке лисп текст программы и данные представляются в единой форме — в виде списков. Более того, в процессе вычислений можно формировать фрагменты программы и выполнять их. Это позволяет легко организовывать преобразование программ и их хранение, что удобно при организации баз знаний, необходимых в системах искусственного интеллекта.

Лисп предоставляет богатые средства для хранения данных об объектах различной природы, в частности геометрических. Это достоинство языка используется в системе автоматизированного проектирования AUTOCAD, реализованной на персональных ЭВМ.

Другая особенность лиспа — это удобство работы с динамическими структурами данных. Программист может вводить новые структуры, "забывая" старые, не заботясь о том, чтобы очищать память от ненужных структур, как это приходится делать, например, в языке паскаль, обращаясь к процедуре DISPOSE. В лисп-системе есть специальное средство "сборщик мусора", которое автоматически освобождает память от неиспользуемых структур.

Обычно транслятор с языка лисп (подобно транслятору с языка бейсик) представляет собой интерпретатор, что предоставляет готовые средства для создания различных диалоговых систем.

Как известно, вычисления в интерпретаторе выполняются медленнее, чем в скомпилированной программе. Поэтому было бы бессмысленно использовать лисп при решении больших вычислительных задач, тем более что запись арифметических выражений в лиспе менее наглядна, чем в языке паскаль.

Вместе с тем есть диалекты лиспа (COMMON LISP, MULISP-85), позволяющие выполнять точные вычисления в рациональных числах, практически любой величины. Такие диалекты можно использовать для нахождения точных значений различных коэффициентов, представляемых рациональными числами.

В данном учебном пособии рассматриваются общие сведения о языке лисп на примере конструкций из диалектов MULISP-85 и COMMON LISP [1]. Кроме этого кратко излагаются основные особенности STANDARD LISP [2]. Другие диалекты рассмотрены в [3,4]. В [3] рассматриваются также способы реализации языка лисп.

1. Атомы и списки как основные объекты языка лисп

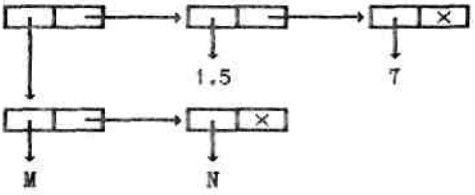
Основными объектами языка лисп являются атомы и списки. Они представляют собой выражения языка лисп и служат как для представления данных, так и для представления программы на языке лисп. Атомы подразделяются на (целые и вещественные) числа и идентификаторы. Идентификаторы и числа записываются подобно тому, как это осуществляется в языке паскаль.

Список представляет собой заключенную в круглые скобки последовательность выражений языка лисп, между которыми могут стоять пробелы (последнее из выражений может отделяться точкой, но это особый случай, который мы рассмотрим ниже). Если в списке рядом стоят два атома, то между ними следует поместить хотя бы один пробел. Несколько пробелов эквивалентны одному. Следующие выражения являются списками:

```
()
(APPLE BOOK CAT)
((M1 N) 1.5 A -9)
((AN APPLE) (Q (P R)))
```

Как видно, элементами списка могут быть списки. Выражение () представляет собой пустой список, не содержащий ни одного элемента. Он имеет также и другое представление — NIL. Указанные два обозначения пустых списков полностью эквивалентны. Таким образом, пустой список является одновременно и атомом, и списком.

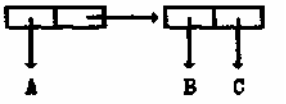
Мы рассмотрели внешнее представление списков. В лисп-системе списки имеют внутреннее представление, которое можно изобразить в виде звеньев, соединенных ссылками; каждое звено состоит из пары ссылок. Так, для списка ((M N) 1.5 7) можно изобразить следующее внутреннее представление:



Значками x мы обозначаем пустые ссылки, которые являются пустыми списками.

В приведенном внутреннем представлении списка все правые ссылки в звеньях указывают на списки. Естественно возникает вопрос: "Какая конструкция получится, если некоторые из правых ссылок будут указывать на атомы, не являющиеся списками?" Такая конструкция будет списком, но с особым внешним представлением.

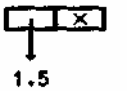
Рассмотрим, например, такое внутреннее представление:



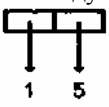
Внешнее представление такого списка будет следующим:

```
(A B . C)
```

В записях таких конструкций используется точка, слева и справа от которой обычно ставятся пробелы. Обратите внимание на разницу конструкций **(1.5)** и **(1 . 5)**. Первая из них имеет такое внутреннее представление:



А вторая — следующее внутреннее представление:



2. Программа на языке лисп. Вычислимые выражения. Основные функции

Программа на языке лисп представляет собой последовательность вычислимых выражений, т.е. выражений, имеющих значения. Значения вычислимых выражений могут быть получены в процессе вычислений. Выполнение программы заключается в последовательном вычислении входящих в нее выражений.

Рассмотрим теперь вычислимые выражения и правила получения их значений.

Число представляет собой вычислимое выражение, значением которого является само это число. Для обозначения того, какое значение имеет выражение, будем справа от выражения помещать знак **==>**, а следом за ним вырабатываемое

значение. Таким образом:

```
   6.0       ==> 6.0
     7       ==> T
-8.32        ==> -8.32
```

Атомы **T** и **NIL** имеют значения, совпадающие с ними самими. Так:

```
T ==> T
() ==> NIL
```

Напомним, что записи **()** и **NIL** эквивалентны.

Другие идентификаторы имеют значения только тогда, когда являются именами переменных. В таком случае значением идентификатора будет значение соответствующей переменной. Например, если значением переменной **X** является список **(A B C)**, то

```
X ==> (A B C)
```

Список вида **(f** a_1 a_2 **...** a_n**)** или **(g)** имеет значение, если f или g, соответственно, является обозначением функции, при этом a_1, a_2, ..., a_n являются аргументами (фактическими параметрами) функции **f**, а функция **g** не имеет аргументов. Обозначение функции представляет собой либо имя функции, либо λ-выражение, которое мы рассмотрим в дальнейшем. Рассмотренный список, таким образом, представляет собой обращение к функции.

При выполнении большинства функций вначале вычисляются фактические параметры, а затем выполняются действия, предписанные данной функции. Такие функции будем называть *обычными*. Однако есть функции, у хоторых вычисляются лишь некоторые фактические параметры, либо параметры вовсе не вычисляются. Такие функции назовем *особыми*.

Аргументом обычной функции будем называть значение фактического параметра, а *аргументом особой функции* — сам фактический параметр.

Одной из особых функций является **QUOTE**, Эта функция имеет один фактический параметр, который и выдается в качестве значения, не будучи вычисленным. Эта функция нужна для того, чтобы явно задавать необходимые значения. Таким образом:

```
(QUOTE A) ==> A
(QUOTE (A B C)) ==> (A B C)
```

Очевидно, что значение **(QUOTE 7)** совпадает со значением выражения **7**, а значение **(QUOTE NIL)** со значением **()**.

Во многих диалектах языка лисп наряду с функцией **QUOTE** можно использовать маркер **'** (апостроф), помещаемый непосредственно перед выражением, которое не следует вычислять. Например:

```
'A ==> A
'(A B C) ==> (A B C)
'8.2 ==> 8.2
```

Рассмотрим теперь простейшие функции над списками. Будем при описании функции после ее имени указывать общий вид обращения к ней.

Функция CAR

(CAR x). Это обычная функция, ее аргумент — непустой список. Значением функции является первый элемент этого списка. Например:

```
(CAR '(A B C)) ==> A
(CAR '(1 . 2)) ==> 1
(CAR '((A B) C)) ==> (A B)
```

Функция CDR:

(CDR x). Это — обычная функция, аргумент **x** — непустой список. Значением функции является выражение, на которое указывает правая ссылка первого звена, т.е. *хвост списка*. Например:

```
(CDR '(A B C)) ==> (B C)
(CDR '(A)) ==> NIL
(CDR '(1 . 2)) ==> 2
(CAR (CDR '(A B C))) ==> B
```

Отметим, что **CAR** и **CDR** — очень быстрые функции.

Фактически, действие **CAR** сводится к выдаче левой ссылки первого звена списка, а CDR — правой ссылки того же звена.

Функция CONS:

(CONS x y). Это — обычная функция, а **x** и **y** — произвольные выражения. Функция создает звено, левая ссылка которого указывает на **x**, а правая — на **y**. Например:

```
(CONS 'A NIL) ==> (A)
(CONS 'A 'B) ==> (A . B)
(CONS 'A '(1 2)) ==> (A 1 2)
(CAR (CONS 1  '(B C))) ==> 1
```

```
(CDR (CONS 1  '(B C))) ==> (B C)
(CONS '(A B) '(C D)) ==> ((A B) C D)
(CONS (CAR '(A B)) (CDR '(A B))) ==> (A B)
(CONS '(CAR '(A)) '(CDR '(A))) ==>
==> ((CAR (QUOTE (A))) CDR (QUOTE (A)))
```

Предикаты

Рассмотрим теперь основные предикаты — функции, проверяющие наличие некоторых свойств. Отметим, что в лиспе в качестве логического значения "ложь" используется атом **NIL**, а в качестве значения "истина" — любое другое значение. Так, следующие значения считаются истинными:

```
5
(A B)
T
```

Для определенности в качестве истинного значения предикатов используется атом **T** (значением **T** является тоже **T**).

Предикаты являются обычными функциями. Существуют следующие основные одноместные предикаты (справа от имени предиката указываем условие, при котором значением предиката будет **T**):

ATOM — аргумент является атомом

NULL — аргумент есть пустой список

NUMBERP — аргумент является числом

ZEROP — аргумент равен нулю

Примеры:

```
(ATOM (CAR '(A B C))) ==> T
(ATOM ()) ==> T
(NULL (CDR '(A))) ==> T
(NULL ()) ==> T
(ZEROP 0) ==> T
(ZEROP  '(A B C)) ==> NIL
(NUMBERP 5) ==> T
```

Рассмотрим теперь операции отношения.

Функция EQ:

(EQ e_1, e_2). Это — обычная функция. Если известно, что один из аргументов — идентификатор, то значение функции будет истинным при равенстве ее аргументов и ложным в противном случае. Если аргументы не идентификаторы, то значение функции может быть и истинным, и ложным. Подробнее мы рассмотрим эту функцию в разделе 9.

Примеры:

```
(EQ 'A 'B) ==> NIL
(EQ 'A  '(A B)) ==> NIL
(EQ 'M (CAR '(M N P))) ==> T
```

Функция EQUAL:

(EQUAL e_1 e_2). Это — обычная функция. Она принимает истинное значение, если у нее одинаковые аргументы. Функция **EQUAL** позволяет сравнивать на равенство любые значения (идентификаторы, числа, списки), В большинстве лисп-систем результатом сравнения вещественного и целого числа будет всегда значение "ложь", даже если эти значения арифметически равны друг другу (например, **5.0** и **5**), Функция **EQUAL** выполняется медленнее функции **EQ**, поэтому использовать функцию **EQUAL** следует в случае особой необходимости.

Примеры:

```
(EQUAL 5 (CAR '(5 A))) ==> T
(EQUAL 0 -0) ==> T
(EQUAL '(A B) (CDR '(1 A B))) ==> T
(EQUAL (CONS 'A 'B) '(A . B)) ==> T
```

Операции отношения

Перейдем теперь к арифметическим операциям отношения. В разных диалектах лиспа эти функции имеют разные названия, мы рассмотрим названия, которые встречаются в COMMON LISP, GOLDEN COMMON LISP и MULISP-85. В новых диалектах языка лисп преимущественно встречаются именно такие названия. Имена функций, определяющих операции отношения, следующие:

= (равно)
/= (не равно)
< (меньше)
> (больше)
<= (не больше)
>= (не меньше)

Примеры:

```
(=  3.0 3.0) ==> T
(<  3 5.5) ==> T
(>= 7.2 7) ==> T
```

Арифметические операции

Перейдем к арифметическим операциям. Они реализуются с помощью обычных функций языка лисп, обозначения которых различны в разных диалектах. Рассмотрим обозначения, используемые в COMMON LISP, GOLDEN COMMON LISP И MULISP-85:

+ (сложение)

- (вычитание)

***** (умножение)

/ (деление)

TRUNCATE (деление нацело)

Примеры:

```
(+ 5 6 7.0) ==> 18.0
(+ 5 6 7) ==> 18
(/ 5 2) ==> 2.5
(TRUNCATE 5 2) ==> 2
(- 7 3.0) ==> 4.0
(* 2 3 4) ==> 24
```

А теперь расскажем о логических функциях.

Функция NOT:

(NOT e). Это — обычная функция. Она определяет операцию логического отрицания. Значение функции всегда совпадает со значением функции **NULL**. Обычно для наглядности в качестве предиката применяется функция **NULL**, а в качестве логического отрицания — **NOT**. Однако это не мешает отдавать предпочтение только одной из этих функций.

Примеры:

```
(NOT NIL) ==> T
(NOT  '(A B C) ==> NIL
```

Функция AND:

(AND e_1 e_2 ... e_n). Это — особая функция. Значением является конъюнкция аргументов. Эта функция последовательно вычисляет аргументы (слева направо) до тех пор, пока не появится значение, равное **NIL**; в этом случае функция прекращает вычисление аргументов и заканчивает свое выполнение со значением **NIL**. Если же значения всех аргументов отличны от **NIL**, то значением функции будет значение последнего аргумента. Таким образом, функция **AND** может использоваться как следующее условное выражение: "если e_1, и e_2,..., и e_{n-1}, то выполнить е ".

Примеры:

```
(AND 'A () (5 B C)) ==> NIL
(AND (ATOM 5) (NUMBERP 5)) ==> T
(AND (+ 5 6) 3 7) ==> T
```

Функция OR:

(OR e_1 e_2 ... e_n). Это — особая функция. Значением ее является дизъюнкция аргументов.

Эта функция последовательно вычисляет аргументы до тех пор, пока не появится значение, отличное от **NIL**; в этом случае функция прекращает вычисление аргументов и выдает последнее вычисленное значение. Если же значения всех аргументов равны **NIL**, то и значением функции будет **NIL**. Таким образом, функция может использоваться для задания такого условного выражения: "если не e_1, не e_2, ..., не e_{n-1} то выполнить e_n"

Примеры:

```
(OR 'A 'B 'C) ==> A
(OR 'NIL 'B 'C) ==> B
(OR (ATOM '(A)) (NUMBERP 'A)) ==> NIL
```

Рассмотрим теперь функцию **COND**, которая служит для задания условных выражений произвольного вида.

Функция COND:

(COND (p_1 e_{11}...e_{1n1})
(p_2 e_{21}...e_{2n2})
(p_k e_{k1}...e_{1nk}))

Эта функция является особой. Как видно, и сами аргументы имеют особый вид. Функция последовательно вычисляет значения p_i(i=1,k) до тех пор, пока не встретится p_j, значение которого отлично от **NIL**; в этом случае функция вычисляет последовательно все $e_{j1},...,e_{jnj}$, а ее значением становится значение ejnj . Если значения всех p_1, p_2,..., p_k равны **NIL**, то функция **COND** заканчивает свое выполнение со значением **NIL**.

Таким образом, выражение **(COND (a b) (T c))** представляет собой условное выражение вида: "если a, то b, иначе c".

Примеры:

```
(COND ((ATOM '(A B C)) A) (T 'B)) ==> B
(COND (T 'A) (Q P R S T)) ==> A
(COND ((< 5 3) 5) (T 3)) ==> 3
```

Рассмотрим теперь некоторые производные функции над списками, действия которых реализуются через простейшие функции **CAR**, **CDR** и **CONS**.

Функция LIST:

(LIST e_1 e_2...e_n). Это — обычная функция. Значением ее является список из аргументов. Таким образом, значение функции совпадает со значением выражения **(CONS e_1, (CONS e_2 ... (CONS e_n NIL) ...))**.

Примеры:

```
(LIST (+ 5 6) (CAR '(A B))) ==> (11 A)
(LIST 'A) ==> (A)
(LIST NIL) ==> (NIL)
(EQUAL (LIST NIL) '(())) ==> T
```

Суперпозиции CAR и CDR

Существуют обычные функции от одного аргумента, действия которых эквивалентны выполнению некоторой суперпозиции **CAR** и **CDR**. Например, выражение **(CADAR x)** имеет то же значение, что и **(CAR (CDR (CAR x)))**. Последовательность **A** и **D** в имени функции отвечает порядку следования **CAR** и **CDR** в эквивалентной суперпозиции.

Так, функция **CDDR** выдает список без двух первых элементов, а **CDDDR** — без трех первых элементов. Функция **CADR** выбирает из списка второй элемент, а **CADDR** — третий. Во многих диалектах языка лисп таких функций 28:

CAAR	**CAADR**	**CDDAR**	**CAADDR**	**CDAAAR**	**CDDAD**
CADR	**CADAR**	**CDDDR**	**CADAAR**	**CDAADR**	**CDDDA**
CDAR	**CADDR**	**CAAAAR**	**CADADR**	**CDADAR**	**CDDDD**
CDDR	**CDAAR**	**CAAADR**	**CADDAR**	**CDADDR**	
CAAA	**CDADR**	**CAADAR**	**CADDDR**	**CDDAAR**	

Примеры:

```
(CADR '(A B C D)) ==> B
(CDADR '((1 Q 3) (M N P) (ONE TWO))) ==> (N P)
(CDDAR '((1 2 3) A)) ==> (3)
```

3. Лямбда-выражения и определение новых функций

В языке лисп есть возможность в качестве обозначения функции использовать наряду с идентификатором функции специальное выражение, называемое лямбда-выражением (λ-выражением). Оно используется, в частности, для явного задания определения функции в обращении к функции. Лямбда-выражение имеет такой вид:

(LAMBDA a e_1 e_2 ... e_n)

где **a** — это список идентификаторов формальных параметров (он может быть и пустым), а **e_1**, **e_2**, ..., **e_n** — вычислимые выражения.

Выполнение обращения к λ-выражению вида **((LAMBDA (a_1 a_2 ... a_k) e_1 e_2 ... e_n) p_1 p_2 ... p_k)**, заключается в следующих трех этапах:

1) последовательно вычисляются фактические параметры p_1, p_2,..., p_k
2) значение каждого p_i(i=1,k) присваивается соответствующему формальному параметру, который внутри λ-выражения служит локальной переменной;
3) последовательно вычисляются e1, e2,..., en, а значением обращения к λ-выражению становится значение en.

Вычисление обращения к λ-выражению вида **(LAMBDA NIL e_1 e_2 ... e_n)** сводится к последовательному вычислению выражений e_1, e_2,...,e_n и к выдаче значения e_n.

Приведем примеры:

```
((LAMBDA () (+ 5 6)) ==> 11
((LAMBDA (X Y) (+ (* X X) (* Y Y))) 3 4) ==> 25
((LAMBDA (X Y)
    (COND ((> X Y) X) (T Y)))
          (* 3 7 8 9) (* 10 11 14)) ==> 1540
```

В последнем примере вычисляется максимальное из значений фактических параметров.

Лямбда-выражение позволяет сократить число вычислений и вводить локальные переменные для дальнейшего использования. Так, без λ-выражения конструкцию из последнего примера можно было бы представить следующим образом:

```
(COND ((> (* 3 7 8 9) (* 10 11 14)) (* 3 7 8 9))
      (T (* 10 11 14)))
```

В этом случае, как видно, больше вычислений.

Определение функций

Однако если нам часто приходится вычислять максимум, то неудобно каждый раз записывать одно и то же λ-выражение. Хотелось бы поставить ему в соответствие имя и указывать это имя всякий раз, когда нужно вычислить максимум. А имя вместе

с поставленным ему λ-выражением есть не что иное, как функция. Тем самым мы приходим к определению обычной функции. В диалектах MULISP-85 и COMMON LISP оно выглядит так:

(DEFUN f a e_1 e_2 ... e_n), где **f** — имя определяемой функции, **a** — список идентификаторов формальных параметров, а e_1, e_2, ..., e_n — выражения вычисляемые при выполнении функции и составляющие тело функции. Указанная конструкция ставит в соответствие имени **f** λ-выражение вида **(LAMBDA a e_1 e_2 ... e_n)**. Теперь при выполнении любой конструкции вида **(f p_1 p_2 ... p_k)**, в том числе и входящей в одно из e_1, e_2,...,e_k будет вычисляться выражение **((LAMBDA a e_1 e_2 ... e_n) p_1 p_2 ... p_k)**, значение которого и станет значением **f**.

Рассмотрим, к примеру, определение функции **EQL**, которая осуществляет сравнение любых атомов:

Пример:

```
(DEFUN EQL (X Y)
 (COND ((NUMBERP X) (AND (NUMBERP Y) (= X Y)))
       (T (EQ X Y))))
```

В некоторых диалектах языка лисп (в частности, в MULISP-85 и COMMON LISP), где функция **EQ** не позволяет сравнивать числа, есть функция **EQL**.

4. Рекурсивные функции

В языке лисп допускается определение рекурсивных функций. Функция называется *рекурсивной*, если во время выполнения ее тела может встретиться обращение к этой функции.

В частности, рекурсивной является функция, в теле которой находится обращение к ней самой. Такая рекурсия называется *прямой*.

Если же в теле некоторой функции f есть обращение к функции f_1, а в теле f_1 — обращение к f_2, ..., а в теле f_{n-1}— обращение к f_n, а в теле f_n — обращение к f, то такая рекурсия называется *косвенной*.

Рассмотрим случаи прямой рекурсии.

Для разъяснения механизма выполнения рекурсивной функции воспользуемся схемой представления выражений и определений функций. Так, запись **R(Q_1(x), Q_2(x))** означает что у схемы **R** параметрами являются две схемы, зависящие от **x**.

Схеме **R** может соответствовать выражение такого вида **(CONS (CAAR X) ((CDR X))**.

Большими буквам будем обозначать имена схем, соответствующие выражениям общего вида, а малыми — только имена схем обращений к функциям, Например, **f(x)** есть схема обращения к функции.

Обычно схему обращения к функции будем изображать с одним параметром. Очевидно, что преобразовать функцию от нескольких аргументов в функцию от одного аргумента можно, объединив аргументы в список.

Рассмотрим теперь схемы для определения функций. Схему определения нерекурсивной функции можно записать так:

f(x)=D(x)

Будем говорить, что некоторая схема **D(a,**b_1**,**b_2**,...,**b_k**)** при заданном значении параметра a *существенно не зависит* от параметра b_i (1<=**i**<=**k**), если при вычислении выражения, соответствующего схеме, параметр b_i не используется.

Пусть схеме **D(a, g(a))** соответствует выражение **(COND ((ATOM X) X) (T (CDR X)))**,

где параметру **a** соответствует **X**, а схеме **g(a)** — обращение к функции **(CDR X)**. Тогда если значением **a** является атом, то схема существенно не зависит от **g(a)**.

Простейший пример схемы рекурсивного определения можно записать так:

f(a)=D(a,f(Q(a))) (1)

причем сами **D** и **C** не содержат обращений к **f**, кроме указанных в параметрах.

Для схемы определения рекурсивной функции существенным является *условие рекурсивного завершения*, которое определяет, при каких ограничениях на аргумент будет получен результат рекурсивной функции. При вычислении выражения **f(y)** получается следующее: вычисляется **D(y,f(Q(y)))**, что приводит к вычислению **D(y,D(Q(y),f(Q(Q(y)))))**, затем вычисляется **D(y,D(Q(y),D(Q(Q(y)),f(Q(Q(Q(y)))))))** и т.д. Чтобы вычисление функции завершилось, необходимо, чтобы на некотором этапе вычислений второй параметр схемы **D** не стал бы влиять на результат, т.е. схема **D** стала бы существенно не зависеть от этого параметра. А в этом и заключается суть

условия рекурсивного завершения.

Так, для приведенной выше схемы условие завершения будет таким: для произвольного допустимого аргумента y либо схема **D(y,f(Q(y)))** существенно не зависит от второго параметра, либо существует **z=(Q(Q(... (Q(y))...))** (для некоторого числа суперпозиций **Q**) — такое, что **D(z,f(Q(z)))** существенно не зависит от **z**.

Если же для некоторого аргумента условие завершения не выполняется, то при выполнении функции это приведет к бесконечной рекурсии, и результат не будет получен.

Для иллюстрации схемы (1) приведем определение функции **ADD**, находящей сумму элементов списка. Считаем, что список содержит только числа, причем последняя правая ссылка равна **NIL**:

```
(DEFUN ADD(X)
       (COND ((NULL X) 0)
             (T (+ (CAR X) (ADD (CDR X))))))
```

Очевидно, что для любого допустимого аргумента выполняется следующее: либо **X** равен **NIL**, либо значение некоторой суперпозиции **(CDR (CDR ... (CDR X) ...))** равно **NIL**. Таким образом, условие завершения выполняется для любого предполагаемого аргумента,

Большинство прямых рекурсивных определений можно записать в виде следующей более сложной схемы:

$$f(a)=D(a, f(Q_1(a)),...,f(Q_m(a))) \qquad (2)$$

где **D** и Q_i (i=1,m) не зависят от **f**. При этом условие завершения имеет следующий вид: для любого допустимого аргумента **y** либо $D(y, f(Q_1(y)), ...,f(Q_m(y)))$ существенно не зависит ни от одного аргумента, кроме первого, либо существует такое **k** (зависящее от **y**), что для любой последовательности j_1, j_2,...,j_k, где $1<=j_i<=m$ (i=1,k), существует **n**, не превосходящее **k** и такое, что для $z=Q_{jn}(Q_{jn-1}...(Q_{j2}\ Q_{j1}(y)))...)$ выражение $D(z,f(Q_1(z)),...f(Q_m(z)))$ существенно не зависит ни от одного аргумента, кроме первого.

Минимальное значение **k** для заданного аргумента, при котором выполняется условие завершения, будем называть *глубиной рекурсии*.

Рассмотрим теперь функцию **SUM**, которая суммирует все числа, содержащиеся в некотором произвольном выражении. (Мы рассматриваем выражения, состоящие только из атомов и

списков. В некоторых диалектах языка лисп есть и другие выражения.) Определение функции можно записать так:

```
(DEFUN SUM (X)
       (COND ((NUMBERP X) X)
             ((ATOM X) 0)
                  (T (+ (SUM (CAR X)) (SUM (CDR X))))))
```

Рассмотрим поэтапное вычисление следующего обращения к функции: **(SUM '(A (3 5) . 7))**. В теле функции будет вычисляться следующее выражение:

```
1). (+ (SUM (CAR '(A (3 5) . 7)))
       (SUM (CDR '(A (3 5) . 7)))
```

В результате вычисления **CAR** получаем следующее:

```
2). (+ (SUM A) (SUM (CDR '(A (3 5) . 7)))
```

Значением первого слагаемого становится 0. **После этого вычисляется** CDR, **в результате чего получаем следующее:**

```
3). (+ 0 (SUM '((3 5) . 7)))
```

Теперь рассматриваем список **((3 5) . 7)**:

```
4). (+ 0
       (+ (SUM '(3 5)) (SUM (CDR '((3 5) . 7)))))
```

Далее порядок вычислений будет следующим:

```
5). (+ 0
            (+      (+ (SUM '3) (SUM (CDR '(3 5))))
                    (SUM (CDR '((3 5) . 7)))))
6). (+ 0
       (+   (+ 3 (SUM '(5)))
               (SUM (CDR '((3 5) . 7)))))
7). (+ 0
       (+   (+ 3  (+ 5 (SUM 'NIL)))
               (SUM (CDR '((3 5) . 7)))))
8). (+ 0
       (+   (+3 (+ 5 0))
               (SUM (CDR '((3 5) . 7)))))
9). (+ 0
       (+   8
               (SUM '7)))
10). (+ 0
            (+ 8 7))
```

После суммирования получается результат — **15**. Для рассмотренного примера глубина рекурсии равна 4.

5. Вспомогательные функции над списками

В этом разделе мы рассмотрим ряд часто встречающих вспомогательных функций. Большинство из этих функций обычно являются встроенными. Если же в какой-то реализации лиспа отсутствует необходимая функция, то можно воспользоваться приводимым ниже определением.

Функция APPEND:

(APPEND x y). Это — обычная функция. Значением функции является список, получающийся в результате конкатенации (соединения) списков **x** и **y**. Определить функцию можно так:

```
(DEFUN APPEND (X Y)
        (COND ((ATOM X) Y)
          (T (CONS (CAR X) (APPEND (CDR X) Y)))))
```

Примеры:

```
(APPEND '(A B) '(C (1) D)) ==> (A B C (1) D)
(APPEND '(1 2 . 3) '(7 8)) ==> (1 2 7 8)
```

Функция REVAPPEND:

(REVAPPEND x y). Это — обычная функция. Она выполняет конкатенацию перевернутого списка x со списком y.

Ее определение может быть таким:

```
(DEFUN REVAPPEND (X Y)
    (COND ((ATOM X) Y)
          (T (REVAPPEND (CDR X) (CONS (CAR X) Y)))))
```

Примеры:

```
(REVAPPEND '(A (B C) D) '(1 2)) ==>   (D (B C) A 1 2)
(REVAPPEND '(A B . C) '(2)) ==> (B A 2)
(REVAPPEND '(ONE TWO) ()) ==> (TWO ONE)
```

Функция REVERSE:

(REVERSE x). Это — обычная функция. Значением функции является список, полученный из элементов списка x, расположенных в обратном порядке. Для данной функции можно написать следующее определение, в котором используется функция **REVAPPEND**:

```
(DEFUN REVERSE (X)
    (REVAPPEND X NIL))
```

Пример:

```
(REVERSE '(10 (A B) 20)) ==> (20 (A B) 10)
```

Функция MEMBER:

(MEMBER x y). Это — обычная функция. Значением функции является подсписок списка y, начинающийся с элемента, совпадающего с атомом x. Если элемента, совпадающего с x, в списке y нет, то значением функции будет NIL. Поскольку в лиспе любое значение, отличное от NIL, является истинным, функция MEMBER может использоваться как предикат, проверяющий, принадлежит ли x списку y. Ее можно определить так:

```
(DEFUN MEMBER(X Y)
    (COND ((ATOM Y) NIL)
        ((EQL X (CAR Y)) Y)
        (T (MEMBER X (CDR Y)))))
```

Примеры:

```
(MEMBER NIL '(A () B)) ==> (NIL B)
(MEMBER 'TWO '(1 TWO 3 TWO 1)) ==> (TWO 3 TWO 1)
(MEMBER 'A '(B (A) C . A)) ==> NIL
```

Функция NTH:

(NTH i x). Это — обычная функция. Значением ее является **i**-й элемент списка **x**. Элементы нумеруются начиная с нуля. Ее можно определить так:

```
(DEFUN NTH (I L)
    (COND ((ATOM L) NIL)
        ((ZEROP I) (CAR L))
        (T (NTH (- I 1) (CDR L)))))
```

Примеры:

```
(NTH 1 '(1 2 3)) ==> 2
(NTH 3 '(A B C (D E))) ==> (D E)
(NTH 0 '(A B C)) ==> A
```

Функция NTHCDR:

(NTHCDR i x). Эта функция обычная. Значением ее является список x без i начальных элементов. Определение функции может быть таким:

```
(DEFUN NTHCDR (I L)
    (COND   ((NULL L) NIL)
            ((ZEROP I) L)
            (T (NTHCDR (- I 1) (CDR L)))))
```

Примеры:

```
(NTHCDR 0 '(1 2 3)) ==> (1 2 3)
```

```
(NTHCDR 2 '(1 2 3)) ==> (3)
(NTHCDR 2 '(A (B C))) ==> NIL
```

Функция LENGTH:

(LENGTH x). Это — обычная функция. Ее значением является число элементов в списке x, т.е. длина этого списка. Функцию можно определить так:

```
(DEFUN LENGTH (X)
    (COND   ((ATOM X) 0)
            (T (+ 1 (LENGTH (CDR X))))))
```

Примеры:

```
(LENGTH NIL) ==> 0
(LENGTH '((A B) C)) ==> 2
```

Функция BUTLAST:

(BUTLAST x i). Это — обычная функция. Она формирует список, содержащий элементы списка x, кроме i последних. Приведем определение этой функции.

```
(DEFUN MCAR (X N)
    (COND   ((<= N 0) NIL)
            (T (CONS (CAR X) (MCAR (CDR X) (- N 1))))))
(DEFUN BUTLAST (L I)
    (MCAR L (- (LENGTH L) I)))
```

Здесь мы воспользовались определением вспомогательной функции **MCAR**, которая формирует список из **N** первых элементов списка **X**. Сама по себе функция **MCAR** является полезной, но она редко встречается в существующих диалектах, поэтому мы не рассматриваем ее отдельно.

Приведем примеры:

```
(BUTLAST '(A B C) 1) ==> (A B)
(BUTLAST '(JOHN MARY JANE) 2) ==> (JOHN)
```

Функция SUBSTITUTE:

(SUBSTITUTE z y z). Эта функция обычная. Она формирует новый список, получающийся из списка z путем подстановки выражения x вместо всех элементов, совпадающих с атомом y. Определение можно дать такое:

```
(DEFUN SUBSTITUTE (X Y L)
    (COND   ((ATOM L) L)
            ((EQL Y (CAR L))
                    (CONS X (SUBSTITUTE X Y (CDR L))))
            (T (CONS (CAR L) (SUBSTITUTE X Y (CDR L))))))
```

Пример:

```
(SUBSTITUTE 1 'ONE '(ONE TWO (ONE TWO) ONE)) ==>
==> (1 TWO (ONE TWO) 1)
```

Функция SUBST:

(SUBST x y z). Это — обычная функция. Она формирует новый список, получающийся из списка z путем подстановки выражения x вместо всех выражений на всех уровнях списка z, совпадающих с атомом y. Если же z — атом, совпадающий с y, то значением функции будет x. Если z — атом, не совпадающий с y, то значением функции будет z. Можно дать следующее определение:

```
(DEFUN SUBST (X Y Z)
    (COND   ((EQL Y Z) X)
            ((ATOM Z) Z)
            (T (CONS (SUBST X Y (CAR Z))
                     (SUBST X Y  (CDR Z))))))
```

Примеры:

```
(SUBST 1  'ONE '(ONE TWO (ONE TWO) ONE)) ==>
==> (1 TWO (1 TWO) 1)
(SUBST 'B 'A '(A C (C A) . A)) ==> (B C (C B) . B)
```

Функция POSITION:

(POSITION x y). Обычная функция. Ее значением является номер первого элемента списка y, совпадающего с атомом x. Если такого элемента в списке нет, то значением функции будет **NIL**. Ее определение может быть таким:

```
(DEFUN POSITION (X L)
    (COND   ((ATOM L) NIL)
            ((EQL X (CAR L)) 0)
            (T ((LAMBDA (N) (COND (N (+ N 1))))
                    (POSITION X (CDR L))) )))
```

Подобно **MEMBER**, эта функция может использоваться как предикат.

Примеры:

```
(POSITION 'A '(C (A B) A)) ==> 2
(POSITION  5 '(1 2 3 4 5)) ==> 4
```

Функция LAST:

(LAST x). Обычная функция. Ее значением является последнее звено списка **x**, т.е. список из последнего элемента списка x. Если список x пустой, то значением функции будет

NIL. Ее можно опередилить так:

```
(DEFUN LAST (L)
    (COND   ((ATOM L) L)
            ((ATOM (CDR L)) L)
            (T (LAST (CDR L)))))
```

Примеры:

```
(LAST '(A1 B2 C3)) ==> (C3)
(LAST '(A1 B2 . C3)) ==> (B2 . C3)
```

Функция REMOVE:

(REMOVE x y). Это — обычная функция. Значением ее является список, состоящий из элементов списка y, за исключением совпадающих с атомом x. Эту функцию можно определить так:

```
(DEFUN REMOVE (X L)
    (COND ((ATOM L) L)
        ((EQL X (CAR D) (REMOVE X (CDR L)))
        (T (CONS (CAR L) (REMOVE X (CDR L))))))
```

Пример:

```
(REMOVE 'A '(A C B A 6 A)) ==> (C B 6)
```

6. Глобальные переменные. Изменение значений переменных

Рассмотрим теперь, как определяется доступ к глобальным переменным.

В языке лисп (в отличие, например, от паскаля) глобальные переменные определяются в том месте, где находится обращение к функции, а не там, где стоит определение.

Пусть имеются следующие определения функций:

```
(DEFUN Q (N) (F 5))
(DEFUN Z (N) (DEFUN F (X) (* N X))
```

Выполним выражение **(Z 10)**. В результате этого будет определена функция **F**, в которой глобальной является переменная с именем **N**. Однако эта переменная не имеет никакого отношения к формальному параметру функции **Z**.

Теперь выполним **(Q 2)**. В теле **Q** стоит обращение к **F**. При выполнении **F** будет использоваться значение **N**, которое выбирается из ближайшей объемлющей функции, где есть переменная с таким именем. В данном случае это — функция **Q**. Таким образом, значением **(F 5)** будет 10. Значит, **(Q 2)** будет

также иметь значение 10. Вычислим теперь следующее выражение:

((LAMBDA (N) (F 5)) 10). В данном случае глобальная переменная **N** в теле функции **F** является параметром λ-выражения. Значит, значением всей указанной конструкции будет 50.

Упомянутое выше правило является частным случаем более общего правила:

a) связь между именем переменной и самой переменной осуществляется в момент доступа к переменной, т.е. при изменении или выборке значения переменной;
b) в момент доступа используется переменная из ближайшей объемлющей выполняемой функции.

Теперь рассмотрим функцию, которая позволяет изменять значения переменных.

Функция SETQ:

(SETQ x v). Особая функция. Первый аргумент не вычисляется, он должен быть идентификатором. Второй аргумент должен быть произвольным вычислимым выражением. В результате выполнения функции переменной, имя которой задано первым аргументом, присваивается значение второго аргумента.

Если к моменту выполнения функции не существовало переменной с именем **x**, то такая переменная заводится в самом широком, глобальном контексте и, таким образом, становится глобальной по отношению ко всем используемым функциям. Такое, автоматическое определение переменной справедливо для большинства современных диалектов языка лисп. В ранних диалектах этого не было.

Значением функции **SETQ** становится значение второго аргумента.

Примеры:

```
(LIST (SETQ M (CAR '(1 2 3))) M) ==> (1 1)
((LAMBDA (N) (LIST N (SETQ N 5) N)) 10) ==> (10 5 5)
```

Можно дать другое определение функции **POSITION**, рассмотренной в предыдущем разделе:

```
(DEFUN POSITION (X L)
    (COND   ((ATOM L) NIL)
            ((EQL X (CAR D) 0)
```

```
((SETQ L (POSITION X (CDR L))) (+ L 1))))
```

7. Диалоговый режим работы. Функции ввода-вывода

Как отмечалось выше, лисп-программа представляет собой последовательность вычислимых выражений. При работе в режиме диалога пользователь вводит необходимое выражение с терминала. Оно обрабатывается лисп-системой. Если это выражение синтаксически правильное, то оно вычисляется. Если во время вычисления не было обнаружено ошибок, то значение выражения выдается на экран дисплея. В случае ошибки на экран выдается соответствующая диагностика.

Функция и определенные в глобальном контексте переменные доступны в течение всего сеанса работы (если не было выполнено специальных действий по их уничтожению).

Порядок определения функций может быть произвольным, поскольку связь между именем функции и определяющим ее выражением осуществляется в момент ее выполнения. Конечно, к началу выполнения функция должна быть определена.

Иногда требуется производить выдачу на экран дисплея промежуточных результатов, полученных во время вычисления некоторого выражения. Для этого служит следующая функция.

Функция PRINT:

`(PRINT e)`. Это — обычная функция. Она выдает на экран дисплея с новой строки значение выражения e. Значением функции становится выдаваемое на экран выражение.

Например, если ввести выражение **`(CONS 'A (PRINT '(1 2 3)))`**, то на экран будет выдано следующее:

```
(1 2 3)
(A 1 2 3)
```

Первая строчка выдается функцией PRINT, а вторая — лисп-системой.

Если необходимо вычислить одно и то же выражение с разными начальными данными, то удобно осуществлять ввод данных с терминала непосредственно при вычислении выражения. Такой ввод можно осуществить с помощью следующей функции.

Функция READ:

`(READ)`. При выполнении функции на экран дисплея выдается символ приглашения на ввод. При этом необходимо

ввести выражение, которое и станет значением функции **READ**.

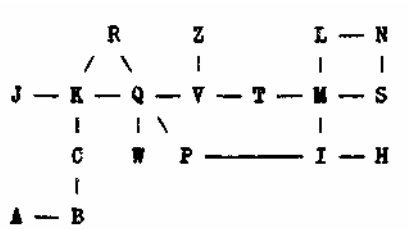
Так, если ввести выражение **(CONS (READ) (READ))** , то на экране дисплея появится символ приглашения на ввод. Если после этого ввести выражение **(+ 2 3)**, то вновь появится символ приглашения. Если теперь ввести идентификатор A, то на экран будет выдано выражение **((+ 2 3) . A)**, которое и является значением выражения, содержащего обращение к READ.

8. Пример программы на языке лисп: поиск пути в лабиринте

Рассмотрим задачу поиска пути в лабиринте. Представим лабиринт в виде отдельных комнат, соединенных проходами. Комнаты обозначим идентификаторами.

Составим список, в котором после каждого имени комнаты укажем список из имен комнат, с которыми данная комната непосредственно соединена проходами. Полученный список присвоим переменной **LABYRINTH**. Так будет задан лабиринт в программе.

Пусть имеется следующий лабиринт:



Он будет представлен в программе так:

```
(SETQ LABYRINTH '(A (B) B (C A) C (B K) H (I) I (P M H) J
(K) K (J C Q R) L (M N) M (T I L S) N (L S) P (Q I) Q (V P
R K W) R (Q K) S (M N) T (M V) V (Q T Z) W (Q) Z (V)))
```

Задача состоит в том, чтобы найти все пути без циклов (где нет комнат, проходимых более одного раза) из одной заданной комнаты в другую. Путь будем задавать списком из пройденных комнат.

Определим функцию WAY, которая выполняет поиск путей:

```
(DEFUN WAY (A B) (PRLISTS (PATH A ())))
```

Параметр **A** определяет исходную комнату, а параметр **B** — конечную. Функция **PATH** выдает список,

содержащий все пути из комнаты, определяемой первым аргументом, в комнату **B** (**B** — глобальная переменная в теле **PATH**). Второй аргумент задает пройденный путь. В начальный момент — это пустой список.

Функция **PRLISTS** служит для выдачи на экран найденных ею путей. Ее можно определить так:

```
(DEFUN PRLISTS (L)
    (COND   ((NULL L) 0)
            (T (PRINT (CAR L)) (+ 1 (PRLISTS (CDR L))))))
```

Значением **PRLISTS** является число найденных путей. Функцию **PATH** определим так:

```
(DEFUN PATH (THIS PATH)
  (COND   ((EQ THIS B) (LIST (REVERSE (CONS B PATH))))
          ((MEMBER THIS PATH) ())
          (T (NEXT (CADR (MEMBER THIS LABYRINTH))
                   (CONS THIS PATH)))))
```

В функции **PATH** вначале проверяется, совпадает ли имя текущей комнаты **THIS** с именем конечной комнаты **B**. Если это так, то выдается список, содержащий найденный путь. Поскольку в процессе вычислений заносить элементы удобно в начало списка, а не в конец, то имена комнат будут перечислены в обратном порядке. Для получения правильного порядка комнат применяется функция **REVERSE**.

Во втором условии функции **COND** проверяется, не была ли ранее пройдена эта комната. Если была, то значением **PATH** будет **NIL**, что означает: путь не найден, текущая вершина найдена неверно.

Третий случай предполагает продвижение в соседние комнаты. Это осуществляется функцией **NEXT**, которая, исходя из текущего пути **PATH**, формирует список путей, получаемых при переходе в соседние комнаты. Приведем определение функции **NEXT**:

```
(DEFUN NEXT (NEXT PATH)
    (COND   ((NULL NEXT) NIL)
            (T (APPEND (PATH (CAR NEXT) PATH)
                       (NEXT (CDR NEXT) PATH)))))
```

Если теперь лисп-системе дать на ввод выражение **(WAY 'A 'Z)**, то на экран будет выдано следующее:

```
(A B C K Q V Z)
(A B C K Q P I M T V Z)
```

```
(A B C K R Q V Z)
(A B C K R Q P I M T V Z)
4
```

Таким образом, будут получены четыре пути без циклов из комнаты **A** в комнату **Z** в заданном лабиринте **LABYRINTH**.

Задавая другие параметры в обращении к функции **WAY**, можно получить пути, соединяющие другие комнаты. Можно исследовать другой лабиринт, изменив содержимое переменной **LABYRINTH**. При этом не требуется изменять или вводить заново используемые в программе функции.

9. Разрушающие функции

До сих пор мы рассматривали такой способ обработки списков, при котором формируются списки, но не изменяется структура уже существующих.

Однако если нам приходится часто изменять отдельные элементы списка, тем более если он при этом большой длины, то частое формирование новых списков приведет к долгому выполнению программы.

Хотелось бы иметь средство, позволяющее изменять структуру списков подобно тому, как мы изменяем, например, элементы массивов в других языках программирования.

Такой способ в лиспе есть. Он, в противовес формированию, называется *разрушением*, а функции, которые изменяют структуру списков, называются *разрушающими*.

Следующие две разрушающие функции являются основными.

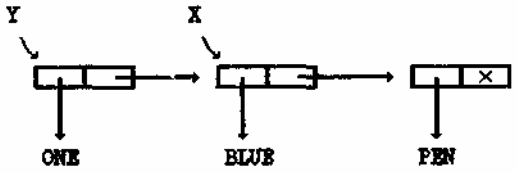
Функция RPLACA:

(RPLACA x e). Обычная функция. Она заменяет левую ссылку в первом звене непустого списка **x** ссылкой на выражение **e**. Значением функции является измененный список.

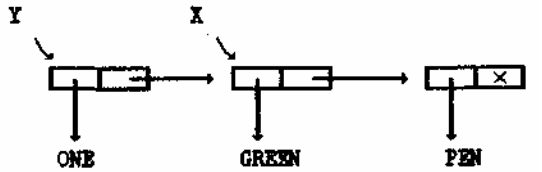
Например, если значением переменной **A** является список **(ONE TWO)**, то после выполнения **(RPLACA A 'ONLY)** значением переменной **A** станет список **(ONLY TWO)**.

Приведем другой пример. Пусть значением переменной **X** является список **(BLUE PEN)**. В результате выполнения **(SETQ Y (CONS 'ONE X))** значением **Y** становится список **(ONE BLUE FEN)**.

Представим схематично, что получилось:



Выполним теперь **(RPLACA X 'GREEN)**. Рассмотрим схему полученной структуры:

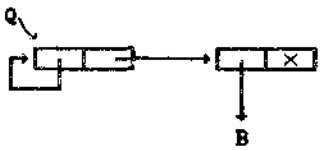


Таким образом, изменилось значение и у переменной Y:

```
Y ==> (ONE GREEN PEN)
```

Отметим, что точно такой же эффект получился бы, если бы мы выполнили вместо **(RPLACA X 'GREEN)** выражение **(RPLACA (CDR Y) 'GREEN)**, при этом значение **X** также изменилось бы.

Рассмотрим третий пример. Пусть значением переменной **Q** является список **(A B)**. Выполнив функцию **(RPLACA Q Q)**, мы получим структуру:



Значение **(CAR Q)** совпадает со значением **Q**, а значение **(CADR Q)** есть атом **B**. При обработке такой структуры может возникнуть бесконечная рекурсия, а при попытке выдать ее на экран непрекращающаяся последовательность из открывающихся скобок появляется на дисплее.

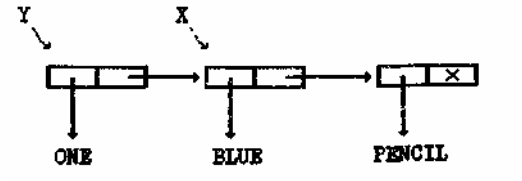
Это — пример так называемой *циклической структуры*. Такие структуры бывают иногда полезны, например, для организации графов. Однако обработку таких структур следует осуществлять особым образом (отлично от обычных списков), чтобы не возникли эффекты, упомянутые выше.

Функция RPLACD:

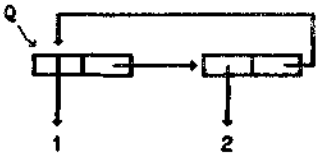
(RPLACD x e). Обычная функция. Она заменяет правую ссылку в первом звене непустого списка **x** на выражение **e**. Значением функции является измененный список **x**.

Например, если значением переменной **Z** является список **(A B)**, то после выполнения **(RPLACD Z 2)** значением **Z** станет список **(A . 2)**.

Рассмотрим другой пример. Пусть значения **X** и **Y** такие же и так же структурно связаны, как во втором примере с функцией **RPLACA**. Тогда результатом выполнения **(RPLACD X '(PENCIL))** будет структура:



Приведем еще один пример. Пусть **Q** имеет значение **(1 2)**. Выполним **(RPLACD (CDR Q) Q)**. Получаем кольцевой список:



Это - частный случай циклической структуры.

Рассмотрим теперь ряд полезных производных разрушающих функций, которые являются встроенными во многих современных диалектах.

Функция NCONC:

(NCONC x y). Обычная функция. Она заменяет ссылку в последнем звене списка **x** ссылкой на **y**. Значением функции при этом становится измененный список **x**. Если исходный список **x** пустой, то значением функции будет **y**.

Ее можно определить так:

```
(DEFUN NCONC (X Y)
    (COND   ((ATOM X) Y)
            (T (RPLACD (LAST X) Y) X)))
```

Для многих разрушающих функций существуют *неразрушающие двойники* — функции, которые формируют значения, во многих случаях такие же, как и соответствующие разрушающие, но при этом не изменяют структуру аргументов. Вместе с тем разрушающие функции являются *разрушающими двойниками* по отношению к своим неразрушающим двойникам. Так, функция **NCONC** является разрушающим двойником для функции **APPEND**.

Пусть, например, значением переменной **x** является список **(A B C)**, а значением **Y** — список **(1 2)**, Тогда в результате выполнения **(NCONC X Y)** значением функции и переменной **X** станет список **(A B C 1 2)**.

Функция NREVERSE:

(NREVERSE x). Обычная функция. Значением ее является перевернутый список **x**. При этом функция разрушает исходную структуру аргумента, причем новым значением аргумента вовсе не является перевернутый список,

Определить функцию можно так:

```
(DEFUN NREVAPPEND (A B)
    (COND   ((ATOM A) B)
            (T (NREVAPPEND (CDR A) (RPLACD A B)))))
(DEFUN NREVERSE (X) (NREVAPPEND X NIL))
```

Мы определили вспомогательную функцию **NREVAPPEND**, которая приписывает к перевернутому списку **A** список **B**, причем исходный список **A** при этом разрушается.

Для функции **NREVERSE** неразрушающим двойником является функция **REVERSE**.

Если, например, значением переменной **A** является список и необходимо заменить ее значение на перевернутый список, то следует выполнить **(SETQ A (NREVERSE A))**, поскольку разрушенное значение аргумента не совпадает со значением функции **NREVERSE**.

Функция DELETE:

(DELETE x y). Обычная функция. Ее значением является список **y**, из которого удалены все атомы **x**. Функция разрушает прежнее значение аргумента **y**, причем новое значение этого аргумента может не совпадать со значением функции.

Определить функцию можно так:

```
(DEFUN DELETE (X L)
    (COND   ((ATOM L) L)
            ((BOX (CAR L) X) (DELETE X (CDR L)))
            ((SQL (CADR L) X)
                    (RPLACD L (CDDR D) (DELETE X L))
            (T (DELETE X (CDR D) L)))
```

Для функции **DELETE** неразрушающим двойником является функция **REMOVE**.

Как было показано выше, при выполнении разрушающих функций могут возникать циклические структуры. Для работы с такими структурами необходимо проверять совпадение ссылок на структуры. Для этого служит рассмотренная ранее функция **EQ**.

Дело в том, что совпадающие по написанию идентификаторы присутствуют в лисп-системе в единственном экземпляре, поэтому при их сравнении достаточно сравнивать ссылки. Однако внешне одинаковые числа и списки могут присутствовать в системе в нескольких экземплярах. Поэтому для их сравнения совпадения ссылок недостаточно, приходится использовать другие функции сравнения (**=** и **EQUAL**). Однако для отыскания циклов в списочных структурах необходимо сравнивать именно ссылки.

Пусть, например, значением переменной **X** является список **(A B C)**. Тогда:

```
(LIST (SETQ Y (CDR X)) (SETQ Z (CDR X)) (EQ Y Z)) ==>
==> ((B C) (B C) T)
```

С другой стороны:

```
(LIST (SETQ Y (CDR X)) (SETQ Z (CONS 'B (CDDR X)))
   (EQ Y Z)) ==> ((B C) (B C) NIL)
```

Приведем более "безопасный" вариант функции **LENGTH** — функцию **LIST-LENGTH**, которая, во-первых, может быть применена к циклическому списку, а во-вторых, позволяет определить, есть ли в списке циклы на верхнем уровне. В диалекте COMMON LISP эта функция является встроенной. В MULISP-85 точно такие же действия выполняет функция **LENGTH**.

Функция LIST-LENGTH:

(LIST-LENGTH x). Обычная функция. Если в списке **x** нет циклов на верхнем уровне, то значением функции будет

длина списка. Если же цикл на верхнем уровне есть, то значением функции будет **NIL**. Определение может быть таким:

```
(DEFUN LL1 (Y1 Y2)
    (COND   ((ATOM Y2) 0)
            ((ATOM (CDR Y2)) 1)
            ((EQ (CDR Y1) (CDDR Y2)) NIL)
            (T (SETQ Y1 (LL1 (CDR Y1) (CDDR Y2)))
                    (AND Y1 (+ Y1 2)))))
(DEFUN LIST-LENGTH (L) (LL1 L D)
```

Примеры:

```
(AND (SETQ X '(A B C)
    (LIST-LENGTH (CONS 'B (RPLACD (CDR X) X)))) ==> NIL
(LIST-LENGTH '(A B C)) ==> 3
```

10. Функционалы

В языке лисп есть возможность выполнять сформированные выражения. Для этого служит функция **EVAL**.

Функция EVAL:

(EVAL e). Обычная функция. Ее значением является вычисленное значение аргумента (значение значения фактического параметра).

Пример:

```
(EVAL (LIST 'CAR '(QUOTE (A B)))) ==> A
(EVAL '(+ 1 2 3)) ==> 6
(QUOTE (EVAL (+ 1 2 3))) ==> (EVAL (+1 2 3))
```

Последние два примера показывают взаимосвязь **EVAL** и **QUOTE**.

С помощью функции EVAL можно определять *функционалы* — функции, аргументами которых являются имена других функций, применяемых функционалами для обработки данных.

Пример функционала — функция, вычисляющая интеграл. Приведем ряд функционалов, которые в большинстве современных диалектов являются встроенными функциями.

Функция MAPCAR:

(MAPCAR f x). Обычная функция. Она применяет обычную функцию **f**(от одного аргумента) к каждому элементу списка **x**, причем элементы списка при этом не вычисляются. Значением **MAPCAR** является список из полученных значений.

Этот функционал можно определить так:

```
(DEFUN MAPCAR (F77 L77)
    (COND   ((ATOM L77) NIL)
            (T (CONS
                    (EVAL (LIST F77 (LIST 'QUOTE
                                          (CAR L77))))
                    (MAPCAR F77 (CDR L77))))))
```

"Хитрые" имена параметров выбраны для того, чтобы они не совпадали с именами переменных в аргументах функционала. Если такое совпадение будет иметь место, то вместо значений указанных перемешших при выполнении функции **EVAL** будут взяты значения параметров, и будет получен неверный результат (см. раздел 6).

Примеры:

```
(MAPCAR 'REVERSE '((1 2) (A B C))) ==> ((2 1) (C B A))
(MAPCAR '(LAMBDA (X) (+ X 10)) '(7 8 9)) ==>
==>(17 18 19)
```

Функция MAPLIST:

(MAPLIST f x). Обычная функция. Она применяет функцию **f** (одного аргумента) вначале ко всему списку **x**, затем к списку **x** без начального элемента, потом к списку **x** без двух первых элементов и т.д., в конце концов к последнему звену списка **x**. Функция **MAPLIST** выдает в качестве результата список из значений, выработанных функцией **f**. Определить ее можно так:

```
(DEFUN MAPLIST (F77 L77)
    (COND   ((ATOM L77) NIL)
            (T (CONS
                    (EVAL
                            (LIST F77 (LIST 'QUOTE L77)))
                    (MAPLIST F77 (CDR L77))))))
```

Примеры:

```
(MAPLIST
    '(LAMBDA (X)
            (COND ((CDR X) (< (CAR X) (CADR X)))))
'(7 2 1 5 3 8)) ==> (NIL NIL T NIL T NIL)
```

Пусть значением переменной X является список **(5 7 -3 9 1)**.

В результате выполнения

```
(MAPLIST
    '(LAMBDA (A) (RPLACA A (- 0 (CAR A))))
    X)
```

значением переменной **X** станет список **(-5 -7 3 -9 -1)**.

Функция MAPCAN:

(MAPCAN f x). Обычная функция. Она подобна **MAPCAR**, только соединяет результаты выполнения **f**, используя **NCONC**.

Для нее можно написать следующее определение:

```
(DEFUN MAPCAN (F77 L77)
    (COND ((ATOM L77) NIL)
        (T (NCONC
            (EVAL
                (LIST F77 (LIST 'QUOTE (CAR L77))))
                    (MAPCAN F77 (CDR L77))))))
```

Пример:

```
(MAPCAN     '(LAMBDA (X) (AND (> X 0) (LIST X)))
            '(-1 2-7 3 -9 9 5)) ==> (2 3 8 5)
```

Функция MAPCON:

(MAPCON f x). Обычная функция. Она подобна **MAPLIST**, только для соединения результатов выполнения f применяет **NCONC**. функцию **MAPCON** можно определить так:

```
(DEFUN MAPCON (F77 L77)
    (COND ((ATOM L77) NIL)
            (T (NCONC
                    (EVAL (LIST F77 (LIST 'QUOTE L77)))
                    (MAPCON F77 (CDR L77))) )))
```

Пример:

```
(MAPCON '(LAMBDA (X)
            (COND   ((MEMBER (CAR X) (CDR X) NIL)
                    (T (LIST (CAR X)))))
    '(A B A C C D))==> (B A C D)
```

В данном примере из списка удаляются повторяющиеся элементы.

Функция MEMBER-IF:

(MEMBER-IF f x). Обычная функция. Она ищет элемент, для которого одноместный предикат **f** принимает

истинное значение, и выдает часть списка **x** начиная с этого элемента. При применении предиката элементы списка не вычисляются. Если для всех элементов списка значение **f** будет **NIL**, то значением функции станет также **NIL**. Определить функцию можно так:

```
(DEFUN MEMBER-IT (F77 L77)
  (COND ((ATOM L77) NIL)
    ((EVAL (LIST F77 (LIST 'QUOTE (CAR L77)))) L77)
    (T (MEMBER-IF F77 (CDR L77)) )))
```

Пример:

```
(MEMBER-IF 'ATOM '((+ 2 3) 7 (A B))) ==> (7 (A B))
```

Функция REMOVE-IF:

(REMOVE-IF f x). Обычная функция. Она формирует новый список, который получается из x удалением всех элементов, для которых значение предиката f является истинным. При применении предиката элементы списка не вычисляются. Можно дать следующее определение этой функции:

```
(DEFUN REMOVE-IF (F77 L77) (COND ((ATOM L77) L77)
            ((EVAL (LIST F77 (LIST 'QUOTE (CAR L77))))
   (REMOVE-IF F77 (CDR L77))) (T (CONS (CAR L77) (REMOVE-IF
F77 (CDR L77))))))
```

Пример:

```
(REMOVE-IF '(LAMBDA (X) (> X 5)) '(10 7 3 8 2 1)) ==>
==> (3 2 1)
```

Функция SUBSTITUTE-IF:

(SUBSTITUTE-IF x f y). Обычная функция. Она формирует новый список, получающийся из списка **y** путем подстановки выражения **x** вместо каждого элемента, для которого истинен предикат **f**. Элементы списка при выполнении предиката не вычисляются.

Определить функцию можно так:

```
(DEFUN SUBSTITUTE-IF (X77 F77 L77)
    (COND ((ATOM L77) L77)
      ((EVAL (LIST F77 (LIST 'QUOTE (CAR L77))))
        (CONS X77 (SUBSTITUTE-IF X77 F77 (CDR L77))))
      (T (CONS (CAR L77)
           (SUBSTITUTE-IF X77 F77 (CDR L77))) )))
```

Пример:

```
(SUBSTITUTE-IF 9 'ZEROP '(0 5 7 0 2)) ==> (9 5 7 9 2)
```

Функция SUBST-IF:

(SUBST-IF x f y). Обычная функция. Она формирует новое выражение из **y** путем подстановки выражения **x** вместо тех выражений на всех уровнях **y**, для которых одноместный предикат **f** принимает истинное значение. Выражения, к которым применяется предикат, не вычисляются. Определить функцию можно так:

```
(DEFUN SUBST-IF (X77 F77 Y77)
    (COND   ((EVAL (LIST F77 (LIST 'QUOTE Y77))) X77)
            ((ATOM Y77) Y77)
            (T (CONS (SUBST-IF X77 F77 (CAR Y77))
                     (SUBST-IF X77 F77 (CDR Y77))) )))
```

Пример:

```
(SUBST-IF 'A  '(LAMBDA (X) (NOT (ATOM X)))
        '(M (1 2) 3)) ==> A
(SUBST-IF 'Q 'NUMBERP '(M (1 2) 3)) ==> (M (Q Q) Q)
```

11. Циклы и блочные функции

До сих пор при определении функций, обрабатывающих списки, мы часто использовали рекурсию. Это удобно: запись определения получается краткой и наглядной.

Однако при выполнении рекурсивной функции требуется дополнительная память, объем которой пропорционален глубине рекурсии и во многих реализациях вычисляется по формуле:

(M*P+K)*L байтов, где **K** и **P** — некоторые фиксированные величины, **M** — число локальных переменных и формальных параметров функции, а **L** — глубина рекурсии. Из формулы вытекает, что при обработке длинных списков рекурсивными функциями может не хватить памяти.

Вместо рекурсии можно воспользоваться циклами. Объем памяти, используемый в цикле, без учета памяти, занимаемой значениями переменных, не зависит от числа итераций. Кроме того, циклические алгоритмы, как правило, выполняются быстрее рекурсивных.

В языке COMMON LISP и некоторых других современных диалектах для организации циклов используется следующая функция.

Функция DO:

(DO x (p s_1 s_2 ... s_n) e_1 e_2 ... e_k). Особая функция. Список **x** пустой либо следующего вида:

(v_1 v_2 ... v_t), где v_i (1<=i<=t) — либо идентификатор, либо список вида: (идентификатор начальное-значение [шаг]), где начальное-значение и шаг — это некоторые вычислимые выражения (квадратные скобки указывают на то, что шаг может отсутствовать).

Выполнение функции заключается в следующем. Вначале объявляются локальные для **DO** переменные с именами, определяемыми идентификаторами из списка x, если, конечно, x не пустой. При этом переменным присваиваются значения выражений, задающих начальные значения. Если начальное значение не указано, то переменной присваивается значение **NIL**. После этого проверяется значение выражения p. Если оно истинно, то вычисляются значения s_1, s_2,...,s_n, а значение s_n становится значением функции **DO**. На этом выполнение функции **DO** завершается.

Если значение p ложно (равно **NIL**), то вычисляются последовательно e_1, e_2, ..., e_k. После этого вычисляются значения шагов и вновь присваиваются локальным переменным. Так продолжается до тех пор, пока значение p не станет истинным.

Поскольку значения шагав присваиваются переменным одновременно, то при отсутствии побочных эффектов (что весьма желательно) порядок v_i в списке x не влияет на результат выполнения **DO**.

Отметим еще одну особенность выполнения **DO**: при отсутствии s_1,s_2,...,s_n значением **DO** будет **NIL**.

Пример 1

Определение функции **REVERSE** можно записать следующим образом:

```
(DEFUN REVERSE (L)
    (DO ((X L (CDR X))
            (Y NIL (CONS (CAR X) Y)))
       ((NULL X) Y) ))
```

Пример 2.

Приведем определение функции (EXP1 X), которая вычисляет значение e^x для простейшего случая, когда $abs(x) < 1$, по ряду $1 + \frac{x}{1!} + \frac{x^2}{2!} + ... + \frac{x^n}{n!} + ...$ с точностью 10^{-6}.

```
(DEFUN EXP1 (X)
    (DO     ((I 1 (+ I 1))
             (U 1 (/ (* U X) I))
             (Z 0 (+ Z U)))
        ((< (ABS U) 0.000001) Z)))
```

Отметим, что для MULISP-85 есть специальный пакет **COMMON.LSP**, в котором реализована эта функция.

В COMMON LISP, а также во многих более ранних диалектах языка лисп есть **PROG**-выражение, позволяющее вводить в употребление блок с метками. Это выражение реализуется с помощью следующей функции.

Функция PROG:

(PROG (i_1 i_2 ... i_n) s_1 s_2 ... s_k). Это особая функция. Здесь $i_1,i_2,...,i_n$ (n>=0)—идентификаторы, а $s_1,s_2,...,s_k$ (k>=1) — идентификаторы или вычислимые выражения. При выполнении функции **PROG** вначале объявляются локальные переменные с именами $i_1,i_2,...,i_n$ и начальными значениями **NIL**. После этого последовательно вычисляются $s_1,s_2,...,s_k$, не являющиеся идентификаторами. Выражения s_j, представляющие собой идентификаторы, называются метками. Во время выполнения **PROG**-выражения может встретиться обращение к функции **GO**, которая осуществляет переход по метке, или обращение к **RETURN**, осуществляющей выход из функции **PROG** с заданным значением. Если при выполнении **PROG**-выражения обращения к функции **RETURN** не было, то значением функции **PROG** становится **NIL**.

Пример:

```
(PROG (X Y) (SETQ X 'A)
  (PRINT (CONS X Y))) ==> NIL
```

При этом на печать будет выдано **(A)**.

Функция RETURN:

(RETURN e). Это — обычная функция. Она осуществляет выход из ближайшего объемлющего PROG-выражения, причем значением PROG-выражения становится значение выражения e.

Функция GO:

(GO x). Особая функция. Аргумент должен быть идентификатором. Он не вычисляется. Функция осуществляет переход по метке x в ближайшем объемлющем **PROG**-выражении. Таким образом, после выполнения функции **GO** будет вычисляться первое из выражений s_j (1<=**j**<=**k**), которое

не является идентификатором и следует за sj0, совпадающим с **x**.

Пример 1.

Рассмотрим реализацию функции **REVERSE**.

```
(DEFUN REVERSE (X)
    (PROG (Y)
      L (COND ((ATOM X) (RETURN Y))
              (T    (SETQ Y (CONS (CAR X) Y))
                    (SETQ X (CDR X))
                    (GO L)))))
```

При использовании **PROG**-выражений часто встречаются конструкции вида **(SETQ x (CONS e x))** и **(SETQ y (CDR y))**. Для упрощения записи таких конструкций используются функции **(PUSH e x)** и **(POP y)**, которые осуществляют те же действия, что и рассмотренные конструкции.

Пример 2.

Рассмотрим определение функции **REVERSE** с использованием функций **POP** и **PUSH**.

```
(DEFUN REVERSE (X)
       (PROG (Y)
          L (COND ((ATOM X) (RETURN Y))
                  (T (PUSH (CAR X) Y) (POP X) (GO L)))))
```

В языке COMMON LISP есть функция **LET**, позволяющая вводить локальные переменные.

Функция LET:

(LET ((i₁ v₁) (i₂ v₂) ... **(iₖ vₖ)) e₁ e₂**... **eₙ)**. Это особая функция. Она вводит в употребление локальные переменные $i_1,i_2,...,i_k$, присваивая им соответственно значения выражений $v_1,v_2,\ldots,v_k$. После этого последовательно вычисляются значения выражений $e_1,e_2,...,$ e_n. Значение e_n становится значением **LET**.

Пример 1.

```
(LET ((A (+ 2 3)) (B '(6 7 8))) (CONS A B)) ==> (5 6 7 8)
```

Пример 2.

Рассмотрим определение функции **MERGESORT**, которая сортирует список располагая его элементы по неубыванию. Исходная структура списка при этом разрушается. Вводятся вспомогательные функции **MERGE** и **MSORT**. Функция **MERGE** формирует упорядоченный список из элементов двух упорядоченных списков.

```
(DEFUN MERGE (L1 L2)
    (COND   ((ATOM L1) L2)
            ((ATOM L2) L1)
            (T (LET ((L NIL))
                (COND ((< (CAR L1) (CAR L2))
                    (SETQ L L1) (POP L1))
                    (T (SETQ L L2) (POP L2)))
                (DO ((P L (CDR P)))
                    ((OR (ATOM L1) (ATOM L2))
                    (COND ((ATOM L1) (RPLACD P L2))
                        (T (RPLACD P L1)))
                    L)
                    (COND ((< (CAR L1) (CAR L2))
                        (RPLACD P L1) (POP L1))
                    (T (RPLACD P L2) (POP 12)))) ))))
(DEFUN MSORT (X1 N)
    (COND ((<= N 1) X1)
            (T (DO (X2
                    (N2 (TRUNCATE N 2))
                    (K 1 (+ K 1))
                    (P X1 (CDR P)))
            ((= K N2)
                    (SETQ X2 (CDR P))
                    (RPLACD P NIL)
                    (SETQ X1 (MSORT X1 K))
                    (SETQ X2 (MSORT X2 (- N K)))
                    (MERGE X1 X2))) )))
(DEFUN MERGESORT (L) (MSORT L (LENGTH L)))
```

12. Макросредства и маркер "обратная кавычка"

Во многих диалектах лиспа есть возможность задавать макроопределения. В MULISP-85 и COMMON LISP это осуществляется функцией **DEFMACRO**:

Функция DEFMACRO:

(DEFMACRO m p e_1 e_2 ... e_n). Особая функция. Здесь **m** — имя определяемого макроса, **p** — список параметров, а e_1, e_2,...,e_n — вычислимые выражения.

Обработка макрокоманд осуществляется в зависимости от реализации либо при вводе в лисп-систему выражений, содержащих макрокоманды, либо непосредственно перед вычислением таких выражений. При этом, естественно, невыполняемые выражения внутри **QUOTE** не обрабатываются.

При обработке макрокоманды вида **(m s_1 s_2...s_k)** вначале устанавливается соответствие между формальными и фактическими параметрами, причем, в отличие от функций, фактические параметры макрокоманды не вычисляются, а присваиваются в явном виде формальным параметрам. После этого вычисляются выражения e_1, e_2,...,e_n а в результате значение e_n подставляется вместо макрокоманды. Значением e_n, в свою очередь, может быть некоторая макрокоманда, которая также будет обработана.

Функция **DEFMACRO** позволяет определять особые функции. Приведем пример определения функции **(IF c e_1 e_2)** (она является встроенной в COMMON LISP и MULISP-85), которая вычисляет e_1, если значение выражения **c** истинно, в противном случае вычисляет значение e_2:

```
(DEFMACRO IF (C E1 E2)
    (LIST 'COND (LIST C E1) (LIST T E2)))
```

На примере определения даже сравнительно простой особой функции **IF** видно, что формировать выражения бывает непросто. Для облегчения формирования выражений введен специальный маркер `(обратная кавычка). При появлении его перед вводимым выражением оно вместе с маркером заменяется конструкцией, значением которой является исходное выражение со следующими подстановками:

a) вместо подвыражений с маркером **,**(запятая) их значений;
b) вместо подвыражений с маркером **,@**(запятая и "коммерческое эт") последовательности из элементов их списочных значений.

Пусть, например, значением переменной **X** является атом **A**, а значением **Y** — список **(1 2 3)**, тогда в результате ввода выражения **`(,X (,Y X) ,@Y)** и его вычисления будет получено следующее:

```
(A ((1 2 3) X) 1 2 3)
```

Маркер "обратная кавычка" есть в COMMON LISP, а также в пакете COMMON.LSP для MULISP-85.

Используя маркер "обратная кавычка" можно переписать определение **IF** так:

```
(DEFMACRO IF (C 11 E2)
 `(COND (,C ,E1) (T ,E2)))
```

Отметим, что если список p в макроопределении имеет вид **(x . y)**, где **x** и **y** — идентификаторы, то значением формального параметра **x** становится выражение s_1, из макрокоманды, а значением **y** — список (s_2 ... s_k). В качестве примера приведем определение функции **(WHEN e z_1 z_2 ... z_k)** (она является встроенной в COMMON LISP и есть в пакете COMMON.LSP для MULISP-85), которая последовательно вычисляет значения z_1, z_2,...,z_k при условии, что значение **e** истинно:

```
(DEFMACRO WHEN (E . S) `(COND (,E ,@S)))
```

13. Обработка текстовых данных

Во многих диалектах языка лисп, в частности MULISP-85, в качестве текстовых строк рассматриваются идентификаторы. Рассмотрим подробнее представление идентификаторов.

У идентифкаторов различают *внешнее представление* и *внутреннее представление*. Внутреннее представление указывает, из каких литер состоит идентификатор, и используется в текстовых операциях (где формируются новые идентификаторы и анализируется, какие литеры входят в идентификатор). Обычно внутреннее представление может содержать произвольную последовательность литер, даже пустую. Внешнее представление необходимо для записи идентификатора в программе, где необходимо указать, какие литеры относятся к идентификатору, а какие к другим конструкциям (числам, спискам и пр.).

Внешнее представление строится по особым правилам на основе внутреннего представления. Обычно каждому внутреннему представлению может соответствовать несколько внешних представлений, но каждому внешнему представлению всегда соответствует только одно внутреннее.

Обычно для записи внешнего представления следует взять внутреннее представление и поместить перед каждой особой литерой специальную литеру — *эскейп*. Особыми литерами обычно являются: круглые скобки, пробел, апостроф, малые буквы, сам эскейп, а также стоящая в начале идетификатора цифра, точка или литера "плюс" или "минус". В идентификаторах, состоящих из одного "плюса" или "минуса", эскейп можно не записывать. Постановка эскейпа перед другими литерами не изменяет внутреннего представления

идентификатора. В некоторых диалектах могут быть и другие особые литеры. Так, в MULISP-85 и COMMON LISP особыми являются, кроме того, и следующие литеры:

```
; " , ' ` | #
```

Литера "обратный слеш" является в этих диалектах эскейпом.

Малые буквы являются особыми, поскольку в обычной записи идентификатора — без эскейпов — они переводятся лисп-системой в соответствующие заглавные буквы.

Приведем примеры внешнего и внутреннего представления идентификаторов. В левой колонке укажем внутреннее представление идентификатора, а в правой одно из возможных внутренних представлений:

внутреннее представление	внешнее представление
abc	**\a\b\c**
3AB	**\3AB**
AbcD	**A\b\cD**
A\B.LSP	**A\\B.LSP**
123"NKL	**\123\"NKL**
****	****
++++	**\++++**

Заметим попутно, что идентификатор, состоящий из цифр, и число без знака — разные объекты. Так:

```
(EQUAL '345 '\345) ==> NIL
(EQUAL '\3\4\5 '\345) ==> T
(NUMBERP '345) ==> T
(NUMBERP '\345) ==> NIL
```

В некоторых других диалектах в качестве эскейпа используются литеры **$** или **!**.

В MULISP-85 и COMMON LISP для облегчения записи идентификатора можно вместо постановки литеры эскейп, заключить внутреннее представление в вертикальные черточки; при этом, однако, требуется поместить эскейп перед следующими литерами: **\ |**

Таким образом, можно записывать внешнее представление так:

внутреннее представлен	внешнее представлен
345abc	|345abc|
• • • •	|....|
a\b"c	|a\\b"c|

Рассмотрим теперь функции над идентификаторами. используемые в диалекте MULISP-85:

Функция UNPACK:

(UNPACK x). Обычная функция. Она создает новый список, состоящий из однолитерных идентификаторов. внутренним представлениеы которых являются литеры идентификатора или числа **x**.

Примеры:

```
(UNPACK 'AB1C2) ==> (A B \1 C \2)
(NUMBERP (CADR (UNPACK 'A123))) ==> NIL
(UNPACK 4567) ==> (\4 \5 \6 \7)
```

Функция PACK:

(PACK y). Обычная функция. Она создает идентификатор, получающийся в результате конкатенации внутренних представлений идентификаторов или чисел, входящих в список **y**.

Примеры:

```
(PACK '(A B 2 CD) ==> AB2CD
(PACK '(ABC |abc|)) ==> |ABCabc|
(PACK '(WONDER FUL)) ==> WONDERFUL
(PACK '(123)) ==> |123|
```

Функция SUBSTRING:

(SUBSTRING x n [m]). Обычная функция. Она формирует идентификатор, внутреннее представление которого состоит из литер внутреннего представления идентификатора **x** с индексами от **n** до **m** включительно. Если m отсутсвует, то считается, что **m** на единицу меньше числа литер в идентификаторе. Литеры нумеруются начиная с нуля.

Примеры:

```
(SUBSTRING '|01234567| 2 5) ==> |2345|
(SUBSTRING 'ABCDEFG 2 4) ==> CDE
(SUBSTRING 'ABCDEF 2) ==> CDEF
```

Функция FINDSTRING:

(FINDSTRING x y [n]). Обычная функция. Она проверяет, входит ли внутреннее представление идентификатора **x** во внутреннее представление идентификатора **y** как подстрока в строку. Если параметр **n** присутствует, то при поиске вхождения просматривается часть литер идентификатора **y** начиная с литеры с номером **n**. Если параметр отсутствует, то просматриваются все литеры идентификатора **y** (начиная с литеры с номером 0).

Примеры:

```
(FINDSTRING 'ABC 'F567ABC*ABC) ==> 4
(FINDSTRING 'ABC 'F567ABC*ABC 5) ==> 8
```

С помощью рассмотренной ранее функции **LENGTH** можно определить число литер во внутреннем представлении идентификатора. Например:

```
(LENGTH 'ABCDEF) ==> 6
(LENGTH 'a\\b) ==> 3
(LENGTH '|"| ) ==> 1
```

Для проверки, является ли значение идентификатором, в MULISP-85 и COMMON LISP используется одноместный предикат **SYMBOLP**. Приведем примеры:

```
(SYMBOLP 'ABC) ==> T
(SYMBOLP '9) ==> NIL
(SYMBOLP '\9) ==> T
```

Определим теперь функцию READ-ATOMS, которая вводит атомы до точки и формирует из них список:

```
(DEFUN READ-ATOMS ()
    (DO     ((X (READ) (READ))
            (Y NIL (CONS X Y)))
    ((AND   (SYMBOLP X)
            (EQ (SUBSTRING X (- (LENGTH X) 1)) '\.))
            (COND ((EQ X '\.) (NREVERSE Y))
                  (T (NREVERSE
                      (CONS
                       (SUBSTRING X 0 (- (LENGTH X) 2))
                      Y)))))))
```

Если с терминала ввести конструкцию **(READ-ATOMS)**, а затем следующее предложение:

```
THERE ARE 3 BOOKS ON THE TABLE.,
```

то на экран будет выдан следующий список:

```
(THERE ARE 3 BOOKS ON THE TABLE)
```

Если же в конце вводимой последовательности находится число, то следует поместить после числа пробел, а затем обратный слеш и точку. Так, для ввода списка из трех чисел **(2 3 4)** с помощью **(READ-ATOMS)** следует ввести с клавиатуры следующую последовательность: **2 3 4 \.**

Если необходимо вводить произвольные последовательности литер, то в диалекте MULISP-85 можно воспользоваться следующей функцией.

Функция READ-CHAR:

(READ-CHAR). Функция осуществляет ввод литеры. Значением функции становится идентификатор, внутренним представлением которого является введенная литера.

Пример.

Определим функцию **READ-CHARS**, которая вводит литеры до точки, формируя из них список (список из однолитерных идентификаторов).

```
(DEFUN READ-CHARS()
    (DO     ((X (READ-CHAR) (READ-CHAR))
             (Y NIL (CONS X Y)))
             ((EQ X '\.) (NREVERSE Y))))
```

При выполнении этой функции пробелы вводятся как отдельные литеры.

Если необходимо определить код литеры или, наоборот, выдать литеру по заданному коду, то следует воспользоваться следующей функцией MULISP-85.

Функция ASCII:

(ASCII x). Обычная функция. Если аргумент **x** — неотрицательное целое число в диапазоне от 0 до 255, то значением функции будет идентификатор, внутренним представлением которого является литера с кодом **x**. Если **x** — однолитерный идентификатор, то значением функции является код литеры из внутреннего представления этого идентификатора.

Пример.

Если необходимо определить такой ввод литер, при котором воспринимались бы только литеры, имеющие графическое представление, то следует игнорировать литеры с кодом, меньшим 32. Определим функцию **READ-GRAPHIC-CHARS**, которая вводит до точки только литеры, имеющие

графическое представление.

```
(DEFUN READ-GRAPHIC-CHARS()
    (DO ((X (READ-CHAR) (READ-CHAR))
           (Y NIL  (COND   ((< (ASCII X) 32) Y)
                           (T (CONS XI)))))
       ((EQ X '\.) (NREVERSE Y))))
```

Если необходимо вывести на экран или напечатать произвольное выражение, не осуществляя перевод строки, то в диалектах MULISP-85 и COMMON LISP следует воспользоваться одной из следующих функций.

Функции PRIN1 и PRINC

(PRIN1 x)

(PRINC x)

Обычные функции. Они выводят на печать аргумент **x** без перевода строчки. Значением каждой из функций является сам аргумент. Первая функция выводит внешнее представление идентификаторов, а вторая внутреннее.

Примеры.

обращение к функциям	вывод на печать
(PRIN1 '|abc|)	|abc|
(PRINC '|abc|)	abc
(PRIN1 'abc)	ABC
(PRINC 'abc)	ABC

Последние два примера показывают, что при отсутствии эскейпов лисп-система переводит малые буквы в заглавные.

Отметим, что функция **PRINT** обычно выводит внешнее представление идентификаторов.

Следующее выражение выводит на печать элементы списка **Y** в строчку:

```
(DO ((X Y (CDR X)))
    ((ATOM X) NIL)
       (PRINC (CAR X)))
```

В MULISP-85 и COMMON LISP для перевода строчки можно воспользоваться следующей функцией.

Функция TERPRI:

(TERPRI). Функция осуществляет перевод строчки. Тем самым, следующий вывод будет выполняться с новой строчки.

Если в диалекте MULISP-85 необходимо расположить идентификаторы в лексикографическом (алфавитном) порядке, то для сравнения идентификаторов можно воспользоваться функцией **STRING<**.

Функция STRING<:

(STRING< x y). Обычная функция. Ее значение будет истинным, если идентификатор **x** предшествует идентификатору **y**, если их расположить в лексикографическом порядке (с учетом литер из их внутренних представлений). В противном случае значением функции будет **NIL**.

В качестве истинного значения функции выступает целое неотрицательное число, которое указывает на номер начальной позиции, начиная с которой литеры идентификаторов различаются.

Примеры.

```
(STRING< 'A *B) ==> 0
(STRING< 'WONDER 'WONDERFUL) ==> 6
(STRING< 'Q 'K) ==> NIL
```

Рассмотрим теперь обработку текстовой информации в COMMON LISP. Для такой цели в этом диалекте есть специальные данные — *строки*. Строка — это массив из литер. Так же, как у идентификаторов, у строк есть внутреннее и внешнее представление.

Строки, в отличие от идентификаторов, не хранятся в лисп-системе в уникальном виде: одновременно может быть несколько одинаковых строк, в то время как идентификатор с данным внутренним представлением существует в единственном экземпляре. При формировании новой строки лисп-система не должна просматривать строки, определяя, существует ли строка с таким внутренним представлением. В связи с этим функция **EQ** не позволяет сравнивать строки. Для сравнения строк используется функция **EQUAL**.

Внешнее представление строки — это последовательность литер из внутреннего представления, заключенная в двойные кавычки. При формировании внешнего представления следует, кроме того, перед каждой литерой "двойная кавычка" и "обратный слеш" из внутреннего представления поставить обратный слеш.

Приведем примеры:

внутреннее представление	внешнее представление
`Abc`	`"Abc"`
`\`	`"\\"`
`"`	`"\""`
`a\b.c`	`"a\\b.c"`

Отметим, что запись "" — является внешним представлением *пустой строки*, т.е. строки, внутреннее представление которой не содержит ни одной литеры.

Значением строки всегда является сама эта строка. Таким образом:

```
"abc" ==> "abc"
"" ==> ""
```

Для определения того, является ли некоторое значение строкой, служит предикат **STRINGP**. Приведем примеры:

```
(STRINGP 'ABC) ==> MIL
(STRINGP "abc") ==> T
(STRINGP '|abc|) ==> NIL
(STRINGP '"w") ==> T
```

Рассмотрим основные операции над строками.

Функция STRING<:

(STRING< s_1 s_2). Обычная функция. Значение ее будет истинным, если строка s_1 предшествует строке s_2 если их расположить в лексикографическом порядке. В противном случае значением функции будет **NIL**. Истинное значение представляет собой число, определяющее номер позиции, начиная с которой строки различны.

Примеры:

```
(STRING< "a" "b") ==> 0
(STRING< "abc" "abcd") ==> 3
```

Функция SUBSEQ:

(SUBSEQ s n [m]). Обычная функция. Она формирует новую строку из литер строки **s** начиная с литеры с индексом **n** и до литеры с индексом (**m**-1) включительно, а если **m** отсутствует, то до последней литеры. Литеры нумеруются начиная с нуля.

Примеры.

```
(SUBSEQ "abc" 0 1) ==> "a"
(SUBSEQ "012345" 2 4) ==> "23"
(SUBSEQ "abcdef" 1) ==> "bcdef"
```

Функция CONCATENATE:

(CONCATENATE 'STRING s_1 s_2...s_n). Обычная функция. Первый аргумент должен быть идентификатором STRING. Аргументы s_1, s_2,...,s_n — строки. Значением функции является конкатенация этих строк. (Вообще говоря, функция **CONCATENATE** может выполнять и другие действия.)

Примеры:

```
(CONCATENATE 'STRING "WONDER" "FUL") ==> "WONDERFUL"
(CONCATENATE 'STRING "1" "2" "" "3") ==> "123"
(CONCATENATE 'STRING "HOPE"
    (SUBSEQ "BEAUTIFUL" 6 9)) ==> "HOPEFUL"
```

Функция SEARCH:

(SEARCH x y [:START2 n]). Обычная функция. Параметр **n** является ключевым, поэтому перед ним необходимо поместить идентификатор **:START2**. Функция определяет, начиная с какой позиции строка **x** входит как подстрока в строку **y**. Если параметр **n** отсутствует, то поиск вхождения осуществляется с **n**-ой позиции строки **y**, в противном случае — с нулевой позиции. Если найдено вхождение, то значением функции становится номер соответствующей позиции, иначе значением функции будет **NIL**. Позиции нумеруются начиная с нуля.

Примеры:

```
(SEARCH "ab" "mabmab") ==> 1
(SEARCH "ab" "mabmab" 2) ==> 4
```

Функция INTERN:

(INTERN s). Обычная функция, аргументом которой должна быть строка. Значением функции становится идентификатор, внутреннее представление которого совпадает с внутренним представлением строки.

Пример:

```
(INTERN "a\\b") ==> |a\\b|
```

Функция PRINC-TO-STRING:

(PRINC-TO-STRING x). Обычная функция. Значением ее является строка, внутреннее представление

которой содержит литеры, которые представляют собой запись аргумента **x**.

Примеры:

```
(PRINC-TO-STRING '|a\\b|) ==> "a\\b"
(PRINC-TO-STRING 729) ==> "729"
(PRINC-TO-STRING '(A B C)) ==> "(A B C)"
```

Для того чтобы напечатать литеры строки, следует воспользоваться функцией **PRINC**, которая при печати строки выводит литеры ее внутреннего представления.

Примеры.

обращения к функции	вывод на печать
`(PRINC "a\\b")`	`a\b`
`(PRINC "\"123\" \"789\"")`	`"123" "789"`

Функция READ-LINE:

(READ-LINE). Функция вводит строчку литер (с экрана или из файла). Значением функции является строка, внутренним представлением которой являются литеры из введенной строчки без литер конца строчки. Так, если при выполнении функции **READ-LINE** набрать с клавиатуры строчку **Input line**

и нажать на клавишу ввода, то значением выражения **(READ-LINE)** станет строка **"Input line"**, содержащая 10 литер.

Для ввода одной литеры в диалекте COMMON LISP можно воспользоваться выражением **(STRING (READ-CHAR))**, значением которого является строка, состоящая из введенной литеры. Функция **READ-CHAR** похожа на одноименнную функцию диалекта MULISP-85, но вырабатывает другое значение, которое функцией **STRING** преобразуется в строковое значение.

14. Работа с файлами

В MULISP-85 для работы с файлами служат следующие функции.

Функция RDS:

(RDS [f]). Обычная функция. Аргумент должен быть идентификатором, внутреннее представление которого совпадает с именем текстового файла. Функция открывает файл для ввода и делает его активным. Таким образом, после выполнения **RDS** функции **READ** и **READ-CHAR** будут осуществлять чтение из данного файла.

При попытке продолжить чтение из файла, когда все его содержимое уже прочитано, приводит к выдаче сообщения об ошибке. Поэтому необходимо вовремя установить ввод с терминала. Для этого в конце файла следует указать признак, при появлении которого следует прекратить ввод. Это может быть, например, **NIL** — при вводе выражений или некоторая редко встречающаяся литера — при вводе литер.

После окончания ввода из файла следует установить ввод с терминала. Это выполняется функцией **RDS** без параметров.

Если текстовый файл содержит программу, то удобно поместить в конце файла выражение **(RDS)**. Для выполнения программы следует выполнить функцию **(RDS 'p)**, где **p** — имя программного файла.

Функция WRS:

(WRS [f]). Обычная функция. Аргумент должен быть идентификатором. Функция определяет файл вывода для функций **PRINT**, **PRINC**, **PRIN1**, **TERPRI**. Имя файла определяется внутренним представлением аргумента **f**. При выполнении **WRS** создается новый пустой файл, куда функции будут заносить информацию.

После окончания вывода следует закрыть файл и переключить вывод на терминал. Для этого следует обратиться к функции **WRS** без параметров.

Пример.

Рассмотрим определение функции **COPYEXPR**, которая осуществляет перепись выражений из одного файла **A** в другой **B**. Признаком конца файла служит появление выражения **(RDS)**.

```
(DEFUN COPYEXPR (A B)
    (RDS A) (WRS B)
    (DO ((X (READ) (READ)))
            ((EQUAL X '(RDS)) (PRINT X) (RDS) (WRS))
                    (PRINT X)))
```

Однако при копировании произвольного текстового файла необходимо определять конец файла. Для этого служит функция **LISTEN**.

Функция LISTEN:

(LISTEN). Значение функции является истинным, если есть еще литеры для ввода (на буфере ввода — при вводе с терминала, в файле — при чтении файла). Значение функции

будет ложным, если литер для ввода нет.

Пример 1.

Определим функцию **COPY**, которая копирует содержимое текстового файла **A** в файл **B**:

```
(DEFUN COPY (A B)
    (RDS A) (WRS B)
    (DO     ((X (READ-CHAR) (READ-CHAR)))
            ((NOT(LISTEN)) (RDS) (WRS))
            (PRINC X)))
```

Пример 2.

С помощью следующего выражения можно осуществлять очистку буфера ввода при вводе с терминала:

```
(DO () ((NOT(LISTEN)) ()) (READ-CHAR))
```

В диалекте MULISP-85 есть удобные средства для сохранения *рабочего поля* лисп-системы, т.е. всех функций и глобальных переменных, созданных пользователем во время работы в системе. В дальнейшем можно загрузить в лисп-систему сохраненное рабочее поле. Для таких целей служат следущие функции.

Функция SAVE:

(SAVE f). Обычная функция. Аргумент должен быть идентификатором. Функция сохраняет рабочее ноле в файле, имя которого определяется аргументом.

Функция LOAD:

(LOAD f). Обычная функция. Аргумент должен быть идентификатором, определяющим имя файла, в котором хранится содержимое рабочего поля. Функция загружает содержимое файла в лисп-систему.

Загрузка рабочего поля осущесвляется гораздо быстрее, чем ввод выражений из текстовых файлов. Поэтому рекомендуется отлаженные программы заносить в файл функцией **SAVE** для последующей быстрой загрузки их функцией **LOAD**.

Рассмотрим теперь работу с файлами в диалекте COMMON LISP.

Функция OPEN:

(OPEN f [:DIRECTION :OUTPUT]). Обычная функция. Аргумент **f** должен быть строкой. Строка определяет имя файла. Если присутствует только первый параметр, то функция выдает в качестве значения локальный файл (значение

используемое функциями ввода-вывода) ввода, связанный с внешним файлом **f**. Если присутствуют оба параметра (второй ключевой — **:DIRECTION :OUTPUT**), то функция выдает локальный файл вывода, связанный файлом **f**. При этом функция открывает внешний файл с именем **f**.

Локальные файлы можно присвоить переменным. Имена переменных, значениями которых являются локальные файлы ввода, можно указывать в качестве параметров в обращениях к функциям **READ**, **READ-CHAR** и **READ-LINE**. Локальные файлы ввода могут быть дополнительными аргументами функций **PRINT**, **PRINC**, **PRIN1** и **TERPRI**. В таких случаях в качестве файла ввода или вывода будет использоваться внешний файл, связанный с данным локальным файлом. Если в указанных функциях вместо локального файла указан идентификатор **T**, то соответствующая функция ввода или вывода будет работать с терминалом. Отметим, что при отсутствии такого параметра вывод будет осуществляться на стандартное устройство вывода, которое не обязательно является терминалом.

Для того чтобы при выполнении функции ввода определить, все ли содержимое файла прочитано, следует указать еще два параметра: **NIL** и некоторое значение, которое будет выдано функцией ввода после того, как все содержимое файла будет прочитано.

Пример.

Выполнение следующего выражения приводит к выдаче выражений из файла с именем **Q.LSP** на терминал.

```
(LET ((Y (OPEN "Q.LSP")))
    (DO     ((X (READ Y NIL '=EOF=) (READ Y NIL '=EOF=)))
            ((EQ X '=EOF=) NIL)
            (PRINT X T))
```

Функция CLOSE:

(CLOSE p). Обычная функция. Аргумент должен быть локальным файлом. Функция закрывает внешний файл, соответствующий данному локальному файлу. После выполнения этой функции возобновить работу с внешним файлом можно только после того, как он будет открыт функцией **OPEN**. Рекомендуется после завершения работы с файлами закрывать их.

Пример 1.

Рассмотрим определение функции **EXEC**, которая выполняет программу из файла, имя которого определяется строкой, являющейся аргументом функции. Используемые переменные имеют сложные имена, чтобы они не совпадали с именами глобальных переменных программы. Считается, что среди выражений файла нет пустых списков.

```
(DEFUN EXEC (F123F)
    (LET ((Y123Y (OPEN F123F)))
            (DO ((X123X     (READ Y123Y NIL NIL)
                            (READ Y123Y NIL NIL)))
            ((NULL X123X) (CLOSE Y123Y))
            (EVAL X123X))))
```

Пример 2.

Определим функцию **(COPY A B)**, которая копирует содержимое текстового файла **A** в файл **B**.

```
(DEFUN COPY (A B)
    (SETQ A (OPEN A))
    (SETQ B (OPEN B :DIRECTION :OUTPUT))
    (DO ((X (READ-LINE A NIL NIL) (READ-LINE A NIL NIL)))
            ((NULL X) (CLOSE A) (CLOSE B))
            (PRINC X B) (TERPRI B)))
```

15. Структурные метки

В некоторых случаях возникает необходимость осуществлять выход из вложенных циклов, из сложных выражений на верхний уровень, Во многих языках это осуществляется с помощью оператора перехода. В лиспе, как известно, действия оператора перехода ограничены ближайшим объемлющим **PROG**-выражением. Есть диалекты лиспа, где вообще нет операторов перехода (например, MULISP-85). В некоторых современных диалектах такие выходы осуществляются с помощью структурных меток. Структурная

метка помечает некоторую конструкцию, из которой в процессе вычислений можно осуществить выход. В MULISP-85 и COMMON LISP для определения структурной метки служит следующая функция.

Функция CATCH:

(CATCH m e_1 e_2 ... e_n). Особая функция, хотя все аргументы вычисляются. Вначале вычисляется первый аргумент, значением которого должен быть идентификатор. Объявляется структурная метка, именем которой становится указанный идентификатор. После этого последовательно вычисляются выражения e_1, e_2, ..., e_n. При вычислении выражений может осуществиться выход из функции **CATCH**, тогда значением функции становится выражение указываемое при выходе. Если же выход не произошел, то значением функции становится значение e_n.

Для выхода по стуктурной метке служит следующая функция.

Функция THROW:

(THROW m e). Обычная функция. Функция осуществляет выход из блищайшей функции **CATCH** со структурной меткой **m**. Значением функции **CATCH** при этом становятся аргумент **e**. Если такой структурной метки в объемлющем выражении нет, то осуществляется выход из программы на верхний уровень лисп-системы. Аргумент **m** может быть пустым списком. В этом случае всегда осуществляется выход из программы.

Пример.

Определим функцию **FINDLIST**, которая определяет, является ли список **A** подсписком списка **B**. Если это так, то значением функции будет часть списка **B**, начало которой совпадает со списком **A**. В противном случае значением функции будет **NIL**.

```
(DEFUN FINDLIST (A B)
(CATCH 'L
  (DO ((B B (CDR B)))
    ((OR NIL))
    (DO ((X A (CDR X)) (Y B (CDR Y)))
            ((AND X Y (NOT (EQUAL (CAR X) (CAR Y)))) NIL)
            (AND (NOLL X) (THROW 'L Y))
            (AND (NULL Y) (THROW 'L NIL))))))
```

Приведем примеры выполнения этой функции:

```
(FINDLIST '() '(A B C D)) ==> (A B C D)
(FINDLIST '(BC) '(A B C B)) ==> (B C D)
```

16. Еще одна программа на лиспе: определение тождественной истинности формул алгебры логики

Рассмотрим следующие обозначения функций алгебры логики:

(NOT x) — логическое отрицание,
(AND z y) — конъюнкция (логическое "и"),
(OR x y) — дизъюнкция (логическое "или"),
(=> x y) — импликация (логическое следование),
(EQ x y) — эквивалентность.

Определим лисп-функцию:

```
(DEFUN => (X Y)
    (COND (X Y) (T T)))
```

Если составить выражения, содержащие такие функции и переменные со значениями **T** или **NIL**, то, вычислив такое выражение в лисп-системе, мы получим значение **T** или **NIL**, которое соответствует значению "истина" или "ложь" логического выражения, построенного по той же формуле. Поэтому нам не требуется создавать специальный механизм вычисления логических выражений, записываемых в такой форме. Вычислять их будет сама лисп-система.

Логическое значение "ложь" будем обозначать буквой **F**.

Составим теперь программу, которая определяет, является ли данная формула алгебры логики тождественно истинной, т.е. принимает ли она истинные значения при любых входящих в нее значениях переменных.

Основная идея решения состоит в том, чтобы выбрать из формулы все имена переменных, а затем, присваивая им различные значения, каждый раз вычислять выражение, соответствующее формуле. Если при вычислении встретилось значение **NIL**, то формула не является тождественно истинной. Если при всех возможных значениях переменных значение выражения является истинным, то формула тождественно истинная.

Составим функцию **VARS**, которая выбирает из лисп-выражения все имена переменных (мы считаем, что они не совпадают с именами логических функций) и заносит их в список, который является значением глобальной переменной **Z**.

```
(DEFUN VARS (X)
    (COND ((NULL X) NIL)
          ((LISTP X) (VARS (CAR X)) (VARS (CDR X)))
          ((MEMBER X Z) NIL)
          ((MEMBER X '(NOT OR AND => EQ T F>) NIL)
          (T (PUSH X Z))))
```

Определим теперь функцию **TRUE-FALSE**, которая присваивает переменным из заданного списка (список — первый аргумент функции) разные наборы значений **T** или **NIL** и каждый раз вычисляет свой второй аргумент, который соответствует логической формуле. При этом мы используем особые имена формальных параметров, чтобы они не совпадали с именами переменных, встречающихся в формуле:

```
(DEFUN TRUE-FALSE (X123X Y123Y)
       (COND ((NULL X123X) (EVAL Y123Y))
             ((OR    (EVAL (LIST 'SETQ (CAR X123X) NIL))
                     (TRUE-FALSE (CDR X123X) Y123Y))
             (EVAL (LIST 'SETQ (CAR X123X) T))
             (TRUE-FALSE (CDR X123X) Y123Y))))
```

Во втором условии функции **COND** проверяется, будет ли значение выражения **Y123Y** истинным, если значение очередной переменной из списка **X123X** примет значение **NIL**. Если такое условие будет выполнено, то рассматривается случай, когда такая переменная принимает значение **T**. Если же условие не выполнено, то нет смысла рассматривать другой случай, значением функции становится **NIL**.

В функции **TRUE-FALSE** не указано, где локализованы рассматриваемые переменные. Конечно, они могут быть и глобальными по отношению ко всей программе; в этом случае их не надо локализовывать. Однако если мы рассматриваем данную задачу как часть более крупной задачи, то идентификаторы других, глобальных переменных не должны совпадать с идентификаторами переменных, встречающихся в логической формуле. Выполнить такие требования бывает трудно. Поэтому лучше локализовать переменные из логической формулы.

Определим теперь одноместный предикат TEST, который проверяет, является ли его аргумент тождественно истинным логическим выражением или нет.

```
(DEFUN TEST (Y)
(LET ((Z NIL) (F NIL)) (VARS Y)
    (EVAL '(LET ,(MAPCAR '(LAMBDA (X) (LIST X NIL)) Z)
```

```
(TRUE-FALSE (QUOTE ,Z) (QUOTE ,Y))))))
```

При выполнении функции формируется обращение к функции **LET**, в котором локализуются необходимые переменные.

Для того чтобы теперь проверить, является ли заданная формула тождественно истинной, следует обратиться к функции TEST, задав соответсвующее лисп-выражение в качестве аргумента. Приведем примеры выполнения функции **TEST**.

```
(TEST '(EQ (OR (NOT M) N) (=> M N))) ==> T
(TEST '(EQ (OR (NOT A) (NOT B)) (NOT (AND A B)))) ==> T
(TEST '(OR A B)) ==> NIL
(TEST '(EQ T (NOT F))) ==> T
(TEST '(EQ A (NOT (NOT A)))) ==> T
(TEST '(NOT F)) ==> NIL
(TEST '(OR A T)) ==> T
(TEST '(NOT (AND A B C F))) ==> NIL
```

Конечно, можно писать логические формулы в таком виде, но это не очень удобно. Дополним программу функциями ввода и вывода, позволяющими вводить формулы в обычной записи. Будем обозначать операции в формуле следующим образом:

~ — отрицание,

& — конъюнкция,

V — дизъюнкция,

=> — импликация,

= — эквивалентность.

Переменные будем обозначать малыми латинскими буквами, а константы "истина" и "ложь" соответственно буквами T и F. Будем записывать формулы с учетом старшинства операций. Мы перечислили операции в порядке убывания их старшинства. Если необходимо изменить порядок операций, то следует поставить скобки.

Теперь необходимо ввести формулу в такой записи и перевести ее в соответствующее лисп-выражение. Так, если на вводе находится формула **~mVn=m=>n** , то она должна быть переведена в следующее лисп-выражение:

```
(EQ (OR (NOT \m) \n) (=> \m \n)).
```

Будем требовать, чтобы после введенной формулы находилась точка. Формулу всегда будем записывать в одну строчку.

Напишем требуемые функции на диалекте MULISP-85 (с использованием пакета COMMON).

Вначале определим функцию ввода строчки литер **READLN**. Известно, что у литеры конца строчки код меньше 32. Значением функции является список из введенных литер.

```
(DEFUN READLN()
    (PRINC '\#)
    (DO ((Y (READ-CHAR) (READ-CHAR))
           (X NIL (CONS Y X)))
           ((< (ASCII Y) 32) (NREYERSE X))))
```

Литера **#** используется в качестве символа приглашения. Определим вспомогательный предикат **LETTERP**, который принимает истинное значение, если его аргумент является допустимым именем переменной или константы.

```
(DEFUN LETTERP(X)
    (OR (EQ X 'F) (EG X 'T)
    (NOT (OR (STRING< X '\a) (STRING< '\z X)))))
```

Введем вспомогательные переменные:

```
(SETQ SET1 '(V & = > |(| ~))
(SETQ SET2 '(V & = |)| ))
```

Определим функцию для выдачи ошибок:

```
(DEFUN ERROR (C I)
    (PRINC '=ERROR=\ ) (PRINC C) (TERPRI)
    (DO ((X LINE (CDR X)))
           ((NULL X) (TERPRI))
           (PRINC (CAR X)))
    (DO ((Z 0 (+ Z 1)))
           ((EQ Z I) (PRINC '^) (TERPRI))
           (PRINC '| |))
    (THROW 'UPPER ()))
```

Эта функция выводит слово **=ERROR=**, за которым указывается аргумент **C**, а в следующей строчке последовательность литер (содержащаяся в переменной **LINE**), под которой в позиции, определяемой переменной **I**, печатается литера **^**(сиркумфлекс).

Определим теперь функцию **EXPR1**, которая будет переводить в списки выражения в скобках. У этой функции первым аргументом является введенный список из литер, вторым — часть сформированного списка (для рекурсивного вызова),

а третьим — признак конца обработки (на верхнем уровне — это точка, а при рекурсивном вызове — закрывающая скобка). При рекурсивном вызове четвертый параметр содержит предыдущую, уже проанализированную литеру, а пятый — номер позиции очередной анализируемой литеры. Четвертый и пятый параметры используются при диагностике ошибок.

При вызове функции на верхнем уровне обращение к ней будет иметь вид:

```
(EXPR1 X () '\. '= 0)
```

где значением переменной **X** является список введенный функцией **READLN**. Отметим, что в качестве четвертого параметра можно указать любую двуместную операцию либо закрывающую скобку. Определение функции **EXPR1** может быть следующим:

```
(DEFUN EXPR1 (X Y END B I)
(COND ((NULL X) (ERROR END I))
    (T (LET ((Z (CAR X)) (P NIL))
    (COND ((EQ Z '| |)
           (EXPR1 (CDR X) Y END B (+ I 1)))
           ((EQ Z END) (CONS I (CONS (CDR X) Y)))
           ((EQ Z '|.|) (ERROR END I))
           ((COND ((EQ Z 'i)|) T)
                  ((MEMBER B SET1)
                  (NOT (OR (EQ Z '|(|)
                           (LETTERP Z) (EQ Z '~))))
                  ((OR (LETTERP B) (EQ B '|)| ))
                  (NOT (MEMBER Z SET2))))
           (ERROR  '|  | I))
           ((EQ Z '|(|  )
                  (SETQ P (EXPR1 (CDR X) NIL '|)|
                         Z   (+ I 1) ))
                  (EXPR1 (CADR P)
                      (APPEND Y (LIST (CDDR P)))
                      END '|)| (+ (CAR P) 1)))
           ((EQ Z '=)
                  (COND ((EQ (CADR X) '>)
                         (EXPR1 (CDDR X)
                             (APPEND Y (LIST '=>))
                             END Z (+ I 1)))
                      (T (EXPR1 (CDR X)
                          (APPEND Y (LIST '=))
```

```
                    END Z (+ I 1)))))
        (T (EXPR1 (CDR X)
                (APPEND Y (LIST Z))
                END Z (+ I 1))))) )))
```

Функция выполняет также удаление пробелов, которые могут встретиться в любом месте.

Обратите внимание, что условие, указывающее на наличие ошибки, записано в виде обращения к функции **COND**.

Если при выполнении функции **EXPR1** была обнаружена ошибка в записи формулы, связанная с отсутствием литеры (точки или закрывающей скобки), то недостающая литера выдается справа от слова **=ERROR=.** Если же ошибка была вызвана появлением недопустимой литеры, то справа от слова **=ERROR=** ничего не указывается (печатается пробел, который невиден). В любом случае позиция ошибки в формуле указывается литерой "сиркумфлекс".

Теперь следует определить функцию, которая будет формировать требуемые лисп-выражение. Перед этим определим ряд глобальных переменных:

```
(SETQ & 'AND)
(SETQ V '0R)
(SETQ = 'EQ)
(SETQ => '=>)
(SETQ OPLIST '(= => V & ~))
```

Введем также функцию **SPLIT**, первый аргумент у которой список, а второй — атом. Значением функции является список, первый элемент которого есть подсписок исходного списка, содержащий элементы до атома, а оставшиеся элементы составляют часть исходного списка после данного атома. Если же атом не входит в заданный список, то значением функции будет **NIL**. Приведем примеры:

```
(SPLIT '(A B C 1 2 3 C B) 'C) ==> ((A B) (1 2 3 C B))
(SPLIT '(1 2 3 4 5) 'A) ==> NIL
```

Определение функции может быть следующим:

```
(DEFUN SPLIT (L X)
    (COND   ((ATOM L) NIL)
            ((EQ (CAR L) X) (CONS NIL (CDR L)))
            ((SETQ X (SPLIT (CDR L) X))
                    (CONS   (CONS (CAR L) (CAR X))
                            (CDR X)))))
```

Определим теперь функцию **EXPR2**, которая строит лисп-выражение по заданной формуле. Первый аргумент функции — это хвост списка, полученный в результате выполнения функции **EXPR1**. Второй аргумент — это список из названий операций, перечисленных в порядке возрастания их приоритета. При рекурсивном вызове первый элемент этого списка всегда указывает название очередной рассматриваемой операции.

Функция находит операции в исходной формуле и подставляет вместо них функции лиспа. Определение функции **EXPR2** может быть следующим:

```
(DEFUN EXPR2 (L OP)
(COND ((ATOM L) L)
       ((AND (NULL OP) L)
              (DO ((X L (CDR X))
                   (Y NIL (CONS (EXPR2 (CAR X)
                  OPLIST)Y)))
                  ((NULL X) (NREVERSE Y))))
       ((NULL (CDR D) (EXPR2 (CAR L) OPLIST))
       (T (COND ((EQ (CAR OP) '~)
           (IF (EQ (CAR L) '~)
                   (LIST 'NOT (EXPR2 (CDR L) OP ))
                   (EXPR2 A ()))))
           (T (LET ((A (SPLIT L (CAR OP)))
                   (Z (CAR OP)))
              (IF A
                   (LIST (EVAL Z)
                          (EXPR2 (CAR A) (CDR OP) )
                          (EXPR2 (CDR A) OP))
                   (EXPR2 L (CDR OP) )))) ))))
```

Определим теперь функцию **PROCESS**, которая вводит формулы и выдает значение **T** или **F** в зависимости от того, является ли формула тождественно истинной или нет. Для того чтобы функция прекратила свое выполнение и не делала бы запрос на ввод, необходимо ввести строку с точкой вначале. Определение функции **PROCESS** может быть следующим:

```
(DEFUN PROCESS () (RDS)
    (DO () ((NOT (LISTEN)) ()) (READ-CHAR))
    (DO ((LINE (READLN) (READLN)) (ERROR NIL) (A NIL))
          ((AND X (EQ (CAR LINE) '\.)) NIL)
          (COND(X
```

```
(CATCH 'UPPER
       (SETQ A (EXPR1 LINE NIL '\. '= 0))
       (SETQ A (EXPR2 (CDDR A) OPLIST ))
       (PRINT (IF (TEST A) T 'F)))))))
```

В начале функции находится обращение к **RDS**, которое устанавливает ввод с терминала. Это необходимо, если обращение к функции **PROCESS** будет введено из файла. Далее идет очистка буфера ввода, чтобы литеры, набранные с клавиатуры при выполнения функции **PROCESS**, не воспринимались бы при выполнении этой функции.

Все относящиеся к программе функции следует поместить в одном файле. Последним выражением должно быть **(PROCESS)**. Для выполнения программы следует ввести этот файл, вычислив выражение **(RDS 'f)**, где **f** — имя данного файла. На экране появится символ приглашения **#**, и можно вводить логические формулы.

Приведем теперь пример выполнения функции **PROCESS**. В строчке с **#** указывается выражение, набираемое пользователем. При этом ответ системы указывается в следующей строчке:

```
#ab.
=ERROR=
ab.
 ^
#~nVm=n=>m.
T
#a
=ERROR=
a
 ^
#a.
F
#(a=b)=(b=a.
=ERROR= )
(a=b)=(b=a.
         ^
#(a=b)=(b=a)).
=ERROR=
(a=b)=(b=a)).
           ^
#    (a=b)=(((b=a)))
```

```
T
#~*V*.
=ERROR=
~*V*.
 ^
#~aVa.
T
#(a=>b)&(b=>a)=(a=b).
T
#~(a&b&c&d&F).
T
# a&T=a.
T
#~(aVb)=~a&~b.
T
#.
NIL
```

Последний ответ системы указывает на то, что произошел выход в лисп: теперь можно набирать выражения лиспа.

17. Особенности диалекта STANDARD LISP

В этом разделе мы рассмотрим основные особенности диалекта STANDARD LISP. Приводимый материал относится также к диалектам UO-LISP и AMI-LISP.

В диалекте есть атомы, списки и строки (есть еще и векторы, которые мы не будем рассматривать). Эскейпом в записи идентификаторов служит литера **!** (восклицательный знак). Строки записываются в двойных кавычках (так же, как в COMMON LISP).

Имеются следующие функции, которые были описаны ранее: **AND**, **APPEND**, **ATOM**, **CAR**, **CDR**, **COND**, **CONS**, **EQ**, **EQUAL**, **EVAL**, **GO**, **LIST**, **MEMBER**, **NCONC**, **NULL**, **NUMBERP**, **PRINT**, **PRIN1**, **PROG**, **QUOTE**, **READ**, **RETURN**, **REVERSE**, **RPLACA**, **RPLACD**, **SETQ**, **STRINGP**, **TERPRI**, а также **CAAR**, **CADR**, ..., **CDDDDR**.

Некоторые функции имеют другие названия:

имя функции в COMMON LISP	имя функции в STANDARD LISP
=	EQN
<	LESSP
>	GREATERP
+	PLUS
-	DIFFERENCE
*	TIMES
SYMBOLP	IDP
PRINC	PRIN2
DEFUN	DE

Функция **LENGTH** используется только для определения длины списка.

Для проверки, является ли некоторое значение непустым списком, служит предикат **PAIRP**, Например:

```
(PAIRP '()) ==> NIL
(PAIRP '(A)) ==> T
(PAIRP '(A B C)) ==> T
(PAIRP 7) ==> NIL
```

Для проверки, является ли число целым, служит предикат **FIXP**, а является ли вещественным — предикат **FLOATP**.

Есть вещественные и целые числа. Деление чисел выполняется следующей функцией.

Функция QUOTIENT:

(QUOTIENT x y). Обычная функция. Если оба аргумента являются целыми числами, то выполняется деление нацело, в результате чего результат будет целым числом. Если хотя бы один из аргументов является вещественным числом, то результат будет вещественным.

Примеры.

```
(QUOTIENT 5.0 3) ==> 1.666666667
(QUOTIENT 5 3) ==> 1
```

Для того чтобы произвести обычное деление целых чисел, требуется преобразовать одно из них в вещественное. Такое преобразование выполняется следующей функцией.

Функция FLOAT:

(FLOAT x). Это — обычная функция. Ее значением является вещественное число, соотвестсвующее целочисленному аргументу.

Пример.

```
(QUOTIENT (FLOAT 5) 3) ==> 1.666666667
```

Для обратного преобразования — вещественного числа в целое — служит функция **FIX**.

Функция FIX:

(FIX x). Обычная функция. Она вырабатывает целочисленное значение, которое соответствует вещественному числу x с отброшенной дробной частью.

Примеры:

```
(FIX 3.7) ==> 3
(FIX -3.7) ==> -3
(FIX 3.2) ==> 3
(FIX -3.2) ==> -3
```

У функции **MAPCAR** изменен порядок записи аргументов, по сравнению с одноименной функцией языка COMMON LISP, Приведем пример:

```
(MAPCAR '(1 2 3 4 5) '(LAMBDA (X) (TIMES X X))) ==>
==> (1 4 9 16 25)
```

Для работы с текстовыми данными есть следующие функции.

Функция EXPLODE:

(EXPLODE x). Обычная функция. Ее значением является список, содержащий однолитерные идентификаторы, из литер которых состоит внешнее представление объекта (обычно **x** — атом или строка).

Примеры:

```
(EXPLODE '!") ==> (!! !")
(EXPLODE 'ABC) ==> (A B C)
(EXPLODE "ABC") ==> (!" A B C !")
```

Функция COMPRESS:

(COMPRESS x). Обычная функция. Аргумент функции должен быть списком из однолитерных идентификаторов. Значением функции является объект, внешнее представление которого состоит из литер, содержащихся в списке.

Если создается идентификатор, то для правильного выполнения функции **EQ**, рекомендуется применить к нему функцию **INTERN**, имеющую один аргумент.

Примеры:

```
(INTERN (COMPRESS '(A B C))) ==> ABC
(COMPRESS '(!" A B C !1 !"))==> "ABC1"
```

Для того чтобы преобразовать идентификатор в строку, можно определить функцию **STRING**:

```
(DE STRING (X)
    (COMPRESS
            (CONS '!" (APPEND (EXPLODE X) '(!")))))
```

Приведем примеры выполнения этой функции:

```
(STRING 'TABLE) ==>"TABLE"
(STRING 'A1) ==> "A1"
```

Для перевода строки в идентификатор можно определить функцию **ID**. При этом требуется вспомогательная функция **CUT**.

Определение может быть таким:

```
(DE CUT (X)
    (COND ((NULL (CDDR X)) (RPLACD X NIL))
            (T (CUT (CDR X)))))
(DE ID (X)
    (SETQ X (EXPLODE X))
    (CUT X)
    (INTERN (COMPRESS (CDR X))))
```

Рассмотрим теперь особенности операции ввода и вывода. Для ввода литер можно воспользоваться функцией **READCH**, которая выполняет в точности то же самое, что и функция **READ-CHAR** диалекта MULISP-85.

Для того чтобы открыть файл ввода с именем **F.LSP** и настроить функции ввода на работу с этим файлом, следует вычислить следующее выражение:

```
(RDS (OPEN "F.LSP" 'INPUT))
```

Для того чтобы открыть файл вывода с именем **G.TXT** и

настроить функции вывода на работу с этим файлом, следует вычислить выражение

```
(WRS (OPEN "G.TXT" 'OUTPUT))
```

После окончания операций ввода следует закрыть файл ввода и настроить функции ввода на работу с терминалом. С этой целью необходимо выполнить **(RDS NIL)**.

После завершения операций вывода для закрытия файла вывода и настройки функций вывода на терминал необходимо вычислить выражение **(WRS NIL)**.

Приведем определение функции **(COPYEXPR A B)**, которая копирует лисп-выражения из файла **A** в файл **B** (имена файлов при этом задаются литерными строками). Считается, что в конце файла **A** находится выражение **NIL**, которое нигде больше в файле не встречается. Определение может быть таким:

```
(DE COPYEXPR (A B)
(PROG (X) (RDS (OPEN A 'INPUT))
        (WRS (OPEN B 'OUTPUT))
L (SETQ X (READ)) (PRINT X)
        (COND ((NOT X) (WRS NIL) (RDS NIL) (RETURN ())))
(GO L)))
```

Теперь рассмотрим, как работать с макросами. Макроопределение осуществляется следующей функцией.

Функция DM:

(DM m (x) e_1 e_2 ... e_k). Особая функция. Конструкции m и x должны быть идентификаторами, а $e_1, e_2, \ldots, e_k$ — вычислимыми выражениями. Функция определяет макрос с именем **m**, параметром **x** и телом $e_1, e_2, \ldots, e_k$.

Если в дальнейшем в некотором вычисляемом выражении встретится макрокоманда **(m s_1 s_2 ... s_k)**, то ее обработка выполняется следующим образом:

a) параметру макроса x присваивается вся макрокоманда;
b) последовательно вычисляются выражения из тела макроопределения, причем значение последнего вычисленного выражения подставляется вместо макрокоманды;
c) вычисляется подставленное вместо макрокоманды выражение.

В качестве примера приведем определение функции **POP**

(она не является встроенной в STANDARD LISP).

```
(DM POP (Y)
    (LIST 'SETQ (CADR Y)
          (LIST 'CDR (CADR Y))))
```

Литература

1. Steele, G. Common Lisp: The Language. — Digital Press, 1984.

2. Hearn, A.C. STANDARD LISP. — SIGPLAN NOTICES, ACM, Vol. 4, No. 9, September 1966,

3. Лавров С.С, Силагадзе Г.С. Автоматическая обработка данных. Язык лисп и его реализация. — М.: Наука, 1978.

4. Маурер У. Введение в программирование на языке лисп. — М.: Мир, 1976.

ОГЛАВЛЕНИЕ

Учебное издание

Семёнов Михаил Юрьевич ЯЗЫК
ЛИСП ДЛЯ ПЕРСОНАЛЬНЫХ ЭВМ

Редактор И.А.Волкова

Н/К Подписано в печать 4.10.89 г. Л-15511. Формат 60x84/16. Бумага
№ 1. Офсетная печать. Заказное. Усл.печ.л. 4,75. Уч.-изд.л. 3,98.
Тираж 400 экз. Заказ № 199.Цена 15 коп.

Ордена "Знак Почёта" Издательство Московского университета.
103009, Москва, ул. Герцена, 5/7. Ротапринт НИВЦ МГУ. 119899,
Москва, Ленинские горы.