Лавров Святослав Сергеевич

ФУНКЦИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ

Статья продолжает опубликованную в предыдущем номере журнала статью того же автора под тем же заголовком. Продолжена нумерация разделов и списка литературы.

3. ЯЗЫК ЛИСП

Абстрактные машины типа машин Тьюринга, имеющие дело со словами в некотором алфавите, оказались заметно проще аппарата рекурсивных функций в арифметике. В *канонических системах* Э. Поста (1943 г., описаны в [6], гл. 12) и в *нормальных алгорифмах* А. А. Маркова (идея которых возникла еще в 1947 г., если не раньше, см. [4]) слова преобразуются не побуквенно, а по значительно более общей схеме. Это позволило сделать действия над словами и их результаты еще более обозримыми. Легче стало доказывать различные утверждения о свойствах алгоритмов. Но и этот шаг был не последним.

Через двадцать с лишним лет после появления упомянутых в конце разд. **1** «Вычисления» работ А. Тьюринга, А. Черча и других авторов Дж.Мак-Карти предложил проект языка под названием Лисп (см. [1] и любые другие публикации по Лиспу). В нем учтен накопленный к тому времени опыт построения языков программирования для компьютеров. Мак-Карти не имел явно в виду создать новую модель абстрактной машины. Главной его целью было построить язык, ориентированный на обработку *символьных данных* – выражений, составляющих основу современной математической нотации. Что это такое, вскоре станет ясным. Вторая особенность Лиспа – использование *функционального стиля* программирования (о нем говорилось выше, в разд. **2**).

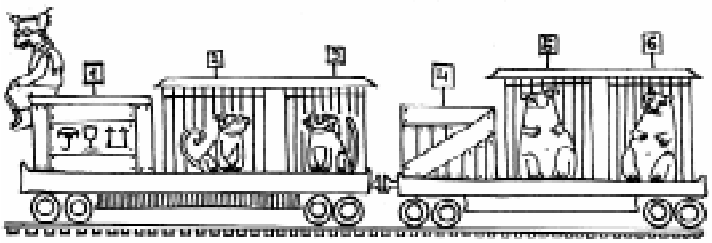
Далее описывается *базовый Лисп* – простой фрагмент языка, сохраняющий все его принципиальные свойства и возможности. *Базовый алфавит* состоит из следующих 40 букв: прописные латинские от A до Z, цифры от 0 до 9, знак ‘–’, ограничители ‘(’, ‘)’ и ‘ ’ (пробел). Первые 37 из них используются для записи слов, воспринимаемых как неделимые *атомы* – слова, состоящие из латинских букв и цифр, но начинающиеся обязательно с латинской буквы, а также отдельные цифры и знак ‘–’. Примеры (записанные без знаков препинания между атомами):

```
LISP  YEAR2000  QUOTE  7  -
```

Выражения Лиспа бывают двух типов: атомы и списки. *Списком* называется заключенная в круглые скобки конечная последовательность выражений – *элементов* списка, разделяемых пробелами, например:

```
(A B C)
(1 (2 3) (4 (5 6)))
(X)
( )
(((HAPPY BIRTHDAY TO YOU)))
```

Число элементов списка называется его *длиной*. Например, только что выписанные списки имеют следующие длины: 3, 3, 1, 0, 1 (но не 4!). В Лиспе допускается большая свобода в записи списков там, где это не приводит к недоразумениям. В частности, под пробелом понимается и перенос продолжения списка



Списком называется... конечная последовательность выражений — элементов списка, разделяемых пробелами, ...(1 (2 3) (4 (5 6))).

на другую строку текста. Например,

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

– это три способа записи одного и того же списка. Пробелы можно добавлять в любом месте списка, но не внутри атомов. Если хотя бы один из двух соседних элементов списка сам является списком, то пробел между ними можно опускать, так что список (1 (2 3) (4 (5 6))) может быть записан еще и так: (1(2 3) (4(5 6))), либо так: (1(2 3)(4(5 6))).

Наиболее употребительные из них T → T...

При описании грамматики языка будем прибегать к ряду *метаобозначений* (то есть обозначений внешнего уровня, не входящих в символику самого Лиспа): *atom* (атом), *list* (список), *expr* (выражение), '|' (заменяет русское слово «или»), '~ *sequence*' (последовательность объектов вида ~, разделенных пробелами – с учетом сказанного о них выше). В этих обозначениях некоторые из приведенных выше правил записываются так:

expr: *atom* | *list*
list: (*expr sequence*)

Выражения в Лиспе часто обладают *значениями*, которыми могут быть другие выражения. *Формой* называется выражение, которое обладает значением, если оно само и его окружение записаны правильно, или от которого этого хотя бы можно ожидать (если его вычисление завершается). В метаобозначениях связь между формой *form* и ее значением *expr* записывается так:

form → *expr*

Значение атома может быть обычным, если данный атом – это *константа*, или *функциональным*, если атом используется в программе как *имя функции* (*fname*). Атом еще одного класса – *переменная* (метаобозначение *var*) может иметь значение любого из этих типов, подробнее это будет разъяснено ниже. Символом → будем пользоваться независимо от типа значения. Начнем с констант. Наиболее употребительные из них – это

```
NIL → NIL
T → T
```

имеющие, таким образом, в качестве значений сами себя. Тем же свойством обладают все цифры, а также знак '–':

```
0 → 0   ...   9 → 9   - → -
```

Атом NIL трактуется двояко – как синоним пустого списка:

```
( ) → NIL   и   NIL → ( )
```

и как логическое значение «ложь». Принято считать, что любое значение, отличимое от NIL, представляет логическое значение «истина». Но стандартным изображением этого логического значения служит атом T.

Форма в виде списка – это обращение к функции (*fcall*), имеющее вид

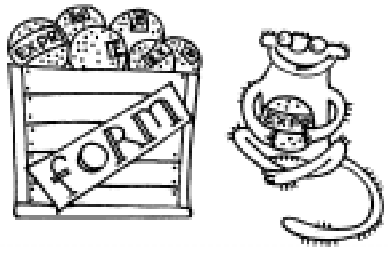
fcall: (*function arg sequence*)
function: *defexpr* | *fname*
arg: *expr*

где *defexpr* – это *определяющее выражение* функции, вид которого вскоре будет описан, *fname* – имя функции, определенной вместе с языком Лисп (*встроенной* в него), или атом, имеющий некоторое определяющее выражение своим функциональным значением, *arg* – выражение, служащее *аргументом* обращения.

Функции бывают *обычными* и *особыми* (*псевдофункциями*). Значение обращения вида (*function arg sequence*) к обычной функции *function* с определяющим выражением, заданным явно или под именем *fname*, вычисляется по следующим правилам:

1) поочередно вычисляются значения аргументов из *arg sequence*,

2) к этому набору значений (с сохранением их порядка) применяется функция *function*.



...связь между формой form и ее значением expr...

Часто, особенно в примерах, некоторые значения требуется задать явно, как бы в кавычках. Делается это в виде обращения (QUOTE *expr*) к особой функции QUOTE. Значение этой формы вычисляется по правилу:

(QUOTE *expr*) → *expr*

где *expr* в обоих вхождениях – одно и то же выражение. Обычно такая форма записывается в сокращенном виде: *'expr*. Примеры:

```
(QUOTE X) → X
'(F G H) → (F G H)
```

Появление в программе атома X или списка (F G H) без кавычки означало бы, как было сказано, требование найти значение этого атома или списка. Атомы NIL, T, все цифры и знак '–' не составляют исключения, но приведенное выше соглашение позволяет записывать эти формы-атомы без кавычек.

Функция QUOTE – это одна из особых встроенных функций (см. выше).

Опишем способ вычисления значения (семантику) некоторых обычных, в отличие от QUOTE, встроенных функций. Значением единственного аргумента функций CAR и CDR должен быть непустой список. Функция CAR выделяет из этого списка первый элемент, а функция CDR – остаток списка после отбрасывания первого элемента.

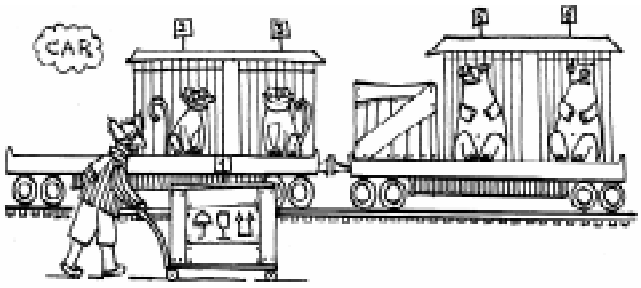
Пусть X → ($expr_1$ $expr_2$ ... $expr_n$). Тогда:

(CAR X) → $expr_1$

(CDR X) → ($expr_2$ ... $expr_n$)

Например,

```
(CAR '(Y 2 K)) → Y
(CDR '(Y 2 K)) →  (2 K)
```



Функция CAR выделяет из этого списка первый элемент...

Докапываясь до элементов из глубины списочной структуры, нередко приходится использовать сложные композиции функций CAR и CDR. Для них применяются сокращенные обозначения: CAAR (двукратное обращение к CAR), CDAR (обращение сначала к CAR – внутреннее обращение, а затем к CDR) и т. д. Так, например,

```
(CADAAR '((A B)(C D))((E F)(G H))) → B
(CADADR '(((A B)(C D))((E F)(G H))) → (G H)
```

Стоит запомнить, что функции CAR, CADR, CADDR, ... выделяют 1-й, 2-й, 3-й и т. д. элементы значения своего аргумента (разумеется, если это значение – список достаточной длины). Поэтому, например, функция CADADR выделяет второй элемент второго элемента этого значения, а функция CADAAR – второй элемент первого элемента его же первого элемента.

Двухместная функция CONS вставляет значение (вид его произволен) своего первого аргумента в начало списка, являющегося значением второго аргумента.

Пусть X → *expr* и Y → ($expr_1$ ... $expr_n$). Тогда: (CONS X Y) → (*expr* $expr_1$ ... $expr_n$)

Например,

```
(CONS 'A '(B C)) → (A B C)
(CONS '(1 2) '(3 4)) → ((1 2) 3 4)
(CONS 'X  NIL) → (X)
```

В последнем примере атом X вставляется в начало пустого списка.

Функции, проверяющие наличие или отсутствие некоторого свойства у набора значений своих аргументов и вырабатывающие, соответственно, значение T или NIL,

значение ... своего первого аргумента в начало списка...

называются *предикатами*, а обращения к таким функциям – *логическими выражениями* (*bool*). Поскольку за истину принимается не только T, но и любое значение, отличное от NIL, роль предикатов могут исполнить очень многие функции.

Предикат ATOM проверяет – не атом ли значение его аргумента:

```
(ATOM (CAR '(A))) → T
(ATOM (CDR '(A))) → T
```

(пустой список () считается атомом, поскольку это то же, что NIL),

```
(ATOM (CDR '(A B))) → NIL
(ATOM (CONS X Y)) → NIL
```

каковы бы ни были значения X и Y.

Предикат EQ проверяет, совпадают ли между собой атомы – значения его аргументов. Он дает значение NIL (вполне разумное) и в том случае, когда значение только одного из аргументов – список. Но проверить тождественность двух списков этот предикат не может. Поэтому значение (EQ *expr expr*) равно T, если значение *expr* – атом, и не определено, если это значение – список. Примеры:

```
(EQ T NIL) → NIL
(EQ T 'T) → T
(EQ (CAR '(A A)) (CDR '(A A))) → NIL
```

Имя COND дано особой функции, предназначенной для разветвления вычислений, точнее – для выбора одного из возможных путей их продолжения. Обращение к COND (или COND-*конструкция*) имеет вид (COND *path sequence*), где *path*: (*bool expr*), подробнее:

(COND (*bool*$_1$ *expr*$_1$) . . . (*bool*$_n$ *expr*$_n$))

и вычисляется по следующим правилам:

1) в ветвях (*bool*$_i$ *expr*$_i$) поочередно, начиная с первой, вычисляются значения *условий* *bool*$_i$,

2) если получено значение NIL, то переходим к следующей ветви,

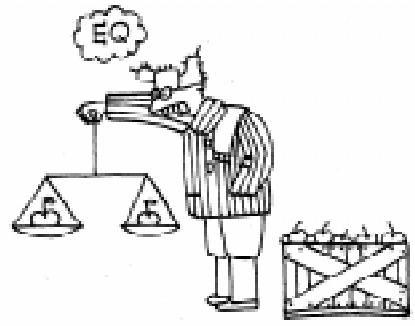
3) если значение *bool*$_i$ отлично от NIL, то *i*-я ветвь считается *выбранной*, и следующие ветви уже не рассматриваются, вычисляется значение выражения *expr*$_i$ в выбранной ветви, оно становится значением всей COND-конструкции,

4) если ни одна ветвь не была выбрана, то за значение конструкции принимается NIL.

Обычно последний случай стараются предотвратить и в последней ветви в качестве условия *bool*$_n$ ставят T. Пример: выражение

```
(COND ((EQ expr NIL) T) (T NIL))
```

имеет значение T, если значение *expr* равно NIL, и значение NIL при любом другом значении *expr*. Можно считать, что это – проверка списка *expr* на пустоту, а можно смотреть на него как на отрицание высказывания *expr* в языке Лисп. Однако записано оно в чересчур сложном виде. В данной COND-конструкции не нужна вторая ветвь (действует правило 4), а в первой ветви нужное значение вырабатывает уже условие (EQ *expr* NIL). Им-то и рекомендуется заменить всю конструкцию.



Предикат EQ проверяет, совпадают ли между собой атомы...

Определяющее выражение функции (LAMBDA-конструкция) имеет вид:

```
(LAMBDA (var sequence)  expr)
```

Оно обозначает функцию, вычисляемую с помощью выражения *expr* (*тела* определяющего выражения) после того, как переменные из *var sequence* (*параметры* функции) получат значения аргументов обращения к этой функции. В обращении к функции вида:

((LAMBDA (*var*$_1$ ... *var*$_m$) *expr*) *arg*$_1$ ... *arg*$_k$)

должно быть $k \le m$. При $k < m$ начальное значение переменных var_{k+1} ... var_m равно NIL.

Часто требуется одно и то же выражение использовать в программе многократно. Поэтому желательно связать его с именем, под которым оно и будет упоминаться. Определение константы

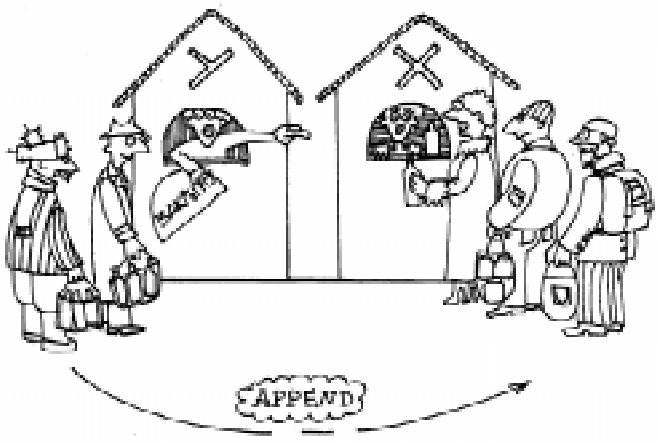
(CSETQ *atom expr*)

связывает атом *atom* со значением (не функциональным) выражения *expr*. Допускается переопределение значения атома, хотя идти на это обычно не рекомендуется.

Написать нетривиальную программу можно лишь с помощью функций, определенных в ней самой. В определении функции

(DEFINE *fname defexpr*)

атом *fname* связывается с явно заданным функциональным значением *defexpr*, то есть становится именем этого значения. Функциональное значение любой встроенной функции определено вместе со всем языком, но обычно не имеет в нем явного представления.



...определим рекурсивную функцию APPEND, вставляющую в начало списка Y ... все элементы списка X...

С учетом сказанного выше функцию NULL, проверяющую список L на пустоту, можно определить так:

```
(DEFINE NULL (LAMBDA (L) (EQ L NIL)))
```

Но эта же функция меняет логическое значение высказывания P на противоположное, поэтому может появиться желание дать ей второе имя – NOT:

```
(DEFINE NOT (LAMBDA (P) (NULL P)))
```

Принципиально важным способом задания функций является знакомая нам по разделу **2** *рекурсия* – обращение к функции из ее собственного определения. В качестве примера определим рекурсивную функцию APPEND, вставляющую в начало списка Y (ее второго параметра) все элементы списка X, сохраняя их порядок:

```
(DEFINE APPEND (LAMBDA (X Y) (COND
 ((NULL X) Y)
 (T (CONS (CAR X) (APPEND (CDR X) Y))) )))
```

В возникающей серии рекурсивных обращений к APPEND последним вставляется в начало формируемого списка значение (CAR X), выделяемое из X первым.

Порядок проверок можно изменить:

```
(DEFINE APPEND (LAMBDA (X Y) (COND
 (X (CONS (CAR X) (APPEND (CDR X) Y)))
 (T Y) )))
```

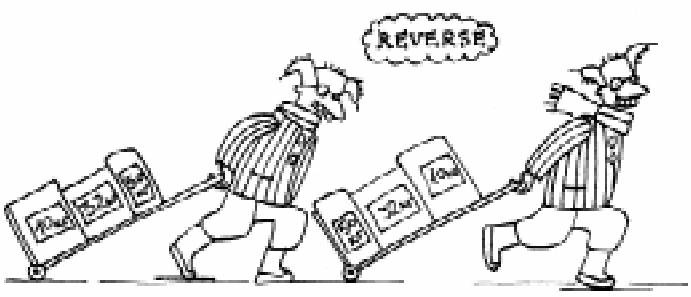
Определение стало на один атом короче, но на столько же непонятнее.

```
(APPEND
  '(RECURSIVE FUNCTIONS OF SYMBOLIC EXPRESSIONS)
  '(AND THEIR COMPUTATION BY MACHINE) )
```

→ (RECURSIVE FUNCTIONS OF SYMBOLIC EXPRESSIONS AND THEIR COMPUTATION BY MACHINE)

(название первой публикации Дж. Мак-Карти по Лиспу (1960)).

Рекурсия может быть и косвенной, когда определения нескольких функций содержат обращения друг к другу, замыкающиеся в один или несколько циклов. При этом любая из них может и не обращаться сама к себе прямо. Обращение к функции называется *рекурсивным*, если оно прямо или косвенно делается из определения этой функции.



Функция REVERSE переставляет элементы списка в обратном порядке...

Во время вычисления формы (*function arg sequence*) *уровнем рекурсии* называется число исполняющихся и еще не завершенных рекурсивных обращений к функции *function*. *Глубиной* рекурсии при обращении к функции с конкретными значениями аргументов называется максимальный уровень рекурсии, достигаемый при этом обращении. Согласно этим определениям, как уровень, так и глубина рекурсии при обращении к нерекурсивной функции могут быть равны только единице.

Пример. Функция REVERSE переставляет элементы списка в обратном порядке, X – еще не перевернутая, а Y – перевернутая часть списка, она-то в конце вычислений и становится их результатом.

```
(DEFINE REVERSE (LAMBDA (X Y) (COND
 ((NULL X) Y)
 (T (REVERSE (CDR X) (CONS (CAR X) Y))) )))
(REVERSE '(1 (2 3) (4 (5 6)))) → ((4 (5 6)) (2 3) 1)
```

В этом примере переменная Y получила при обращении к REVERSE лишь с одним аргументом значение NIL. Но к функции REVERSE можно обратиться и с двумя аргументами, тогда она делает нечто большее, чем было сказано:

```
(REVERSE '(C B A) '(D E F)) → (A B C D E F)
```

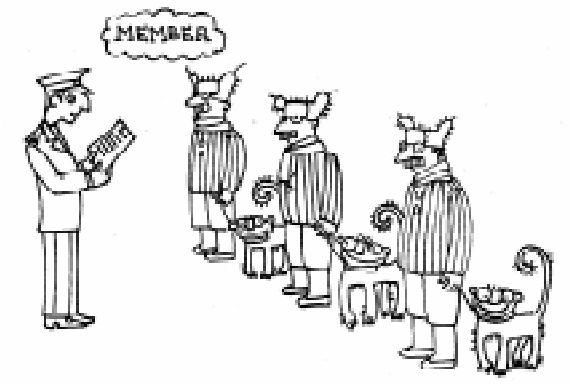
Еще два примера. Функция EQUAL действует аналогично EQ, но допускает в качестве первого аргумента произвольное выражение.

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

Первые две ветви блокируют рекурсивное обращение к EQUAL с атомарными значениями одного из аргументов. При рекурсивном обращении в третьей ветви длина списков X и Y уменьшается на 1, чем обеспечивается завершение со временем серии всех рекурсивных обращений.

Функция MEMBER просматривает список Y и вырабатывает с помощью EQUAL значение T, если встречает в нем элемент, совпадающий с X, и значение NIL – в противном случае.

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



Функция MEMBER просматривает список Y и вырабатывает с помощью EZUAL значение T...

Более сложные примеры встретятся ниже в заданиях 5 и 6.

Итак, базовый Лисп – это набор, состоящий из атомов NIL, T и некоторых других, из функций или конструкций: QUOTE (или ′), CAR, CDR, CONS, ATOM, EQ, COND, LAMBDA, CSETQ и DEFINE и описанных выше правил их композиции и вычисления полученных выражений. Этот язык может служить вполне приемлемой для программиста моделью абстрактной машины.

Программа на Лиспе (Лисп-программа) состоит из произвольной последовательности форм – обращений как к функциям и константам Лиспа, так и к тем, что были определены в самой программе до обращения к ним. «До обращения» не значит, что обращение к функции (или к константе) не может стоять в тексте программы выше места ее определения – ограничение относится только к временно́й последовательности исполнения определений и обращений.

Хороший стиль программирования на Лиспе предполагает, что в начале программы определяются все необходимые для ее работы функции, а затем с их помощью вычисляются одна или несколько форм. Отступление от этого правила обычно ничего не дает, лишь затрудняя понимание программы.

Базовый алфавит из-за нерасчленимости атомов не следует считать алфавитом программы, написанной на Лиспе. Фактически алфавит программы представлен совокупностью всех атомов, используемых в программе.

Список – это обобщение понятия слова. Списки, составленные только из атомов, полностью аналогичны словам. Возможность включать списки в некоторый список в качестве его элементов придает ему структурированность, иерархичность, не присущую слову согласно его определению, но хорошо отображаемую графами – деревьями. Это также способствует обозримости программ и данных. Списки образуют один из важнейших вариантов символьных данных, упоминавшихся в начале раздела.

На метауровне любой список можно считать словом в базовом алфавите. Такое слово можно при желании записать на ленте машины Тьюринга и подвергнуть его обработке по той или иной программе, например, проверяющей правильность расстановки скобок или даже исполняющей Лисп-программу, представленную этим словом.

Несколько слов об использовании функциональных аргументов, то есть аргументов, обладающих функциональными значениями. Параметры определяющего выражения не делятся на обычные и функциональные. Большей частью параметр, которому передано функциональное значение, используется в теле определяющего выражения как имя функции. Вот пример:

```
(DEFINE COMPOSITION (LAMBDA (F G X) (F (G X)) ))
```

Здесь в общем виде описано, что понимается под композицией двух функций. Так как в этом описании параметры F и G используются в качестве имен функций, то соответствующие им аргументы должны быть функциональными, как, например, в следующих определениях:

```
(DEFINE MCADR (LAMBDA (X) (COMPOSITION  'CAR 'CDR X) ))
(DEFINE MCDDR (LAMBDA (X) (COMPOSITION  'CDR 'CDR X) ))
(DEFINE MCDDADR (LAMBDA (X) (COMPOSITION  'MCDDR 'MCADR X) ))
```

Эти функции действуют точно так же, как CADR, CDDR и CDDADR:

```
(MCADR '(A B C)) → B
(MCDDR '(A B C)) → (C)
(MCDDADR '(A (B C D))) → (D)
```

Но если функциональный аргумент – это не имя встроенной функции, то есть если его значение имеет явное лисповское представление в виде некоторого определяющего выражения, то с ним можно обращаться и как с обычным выражением – выделять его элементы, исследовать его структуру и т. п.

Задание 5.

Описать на Лиспе модель арифметики, содержащую все основные операции над целыми числами. Натуральное число *n* представить в этой модели списком, содержащим *n* элементов, каждый из которых – это атом 1. При этом ( ) изображает число 0, (1) – число 1, (1 1) – число 2 и т. д. Отрицательное число – *m* изобразить, включив в начало списка атом '–', так что список (– 1) изображает число – 1, (– 1 1) – число – 2 и т. д.

Выполнение задания 5.

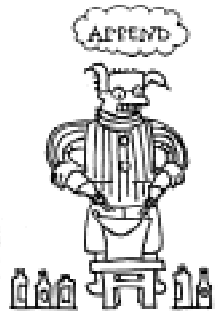
Определим сначала арифметические операции над натуральными числами. При принятых соглашениях функция APPEND из разд. **3** «Язык Лисп» с успехом моделирует сложение натуральных чисел:

```
(DEFINE PLUS (LAMBDA (M N) (APPEND M N)))
```

Функция DIFFERENCE с определением

```
(DEFINE DIFFERENCE (LAMBDA (M N) (COND
 ((NULL N) M)  ((NULL M) (CONS – N))
 (T (DIFFERENCE (CDR M) (CDR N))) )))
```

вычисляет разность натуральных чисел. Ее результат может быть отрицательным целым.



...функция APPEND моделирует сложение натуральных чисел.

С умножением все просто (сравнить с аксиомами M1 и M2 из разд. **1** «Вычисления»):

```
(DEFINE TIMES (LAMBDA (M N) (COND
 ((NULL N) NIL)
 (T (PLUS (TIMES M (CDR N)) M)) )))
```

Прежде чем заняться делением с остатком, определим арифметический предикат LESSP, проверяющий, что значение первого аргумента меньше значения второго

```
(DEFINE LESSP (LAMBDA (M N) (COND
 ((NULL N) NIL)  ((NULL M) T)
 (T (LESSP (CDR M) (CDR N))) )))
```

Будем считать также, что значение делителя N положительно. В противном случае обе последующие функции (QUOTIENT – частное и REMAINDER – остаток) вовлекаются в безысходную рекурсию. В QUOTIENT потребуется вспомогательная переменная L, где будет накапливаться значение частного:

```
(DEFINE QUOTIENT (LAMBDA (M N L) (COND
 ((LESSP M N) L)
 (T (QUOTIENT (DIFFERENCE M N) N (CONS 1 L))) )))
(DEFINE REMAINDER (LAMBDA (M N) (COND
((LESSP M N) M)
(T (REMAINDER (DIFFERENCE M N) N)) )))
```

которые довольно естественно объединяются в одну:

```
(DEFINE DIVIDE (LAMBDA (M N Q) (COND
 ((LESSP M N) (CONS Q (CONS M NIL)))
  (T (DIVIDE (DIFFERENCE M N) N (CONS 1 Q))) )))
```

Результатом становится список из двух элементов: частного и остатка от деления M на N.

Очень просто описывается алгоритм Евклида для нахождения наибольшего общего делителя двух положительных целых чисел:

```
(DEFINE GCD (LAMBDA (M N) (COND
   ((NULL N) M)
   (T (GCD N (REMAINDER M N))) )))
```

...проверяющий, что значение первого аргумента меньше значения второго...

Переходя к арифметике целых чисел без ограничений на знаки, запасемся предикатом MINUSP, отличающим отрицательные числа от неотрицательных:

```
(DEFINE MINUSP (LAMBDA (P) (COND
 ((NULL P) NIL) (T (EQ (CAR P) -)) )))
```

и одноместной функцией MINUS, меняющей знак числа на противоположный:

```
(DEFINE MINUS (LAMBDA (P) (COND
 ((NULL P) P)  ((EQ (CAR P) -) (CDR P))
 (T (CONS - P)) )))
```

Операция ZPLUS сложения целых чисел описывается так:

```
(DEFINE ZPLUS (LAMBDA (P Q) (COND
 ((NULL P) Q) ((NULL Q) P)
 ((MINUSP P)
  (COND
   ((MINUSP Q) (MINUS (PLUS (CDR P) (CDR Q) )))
   (T (MINUS (DIFFERENCE (CDR P) Q))) ))
((MINUSP Q) (MINUS (DIFFERENCE (CDR Q) P)))
(T (PLUS P Q)) )))
```

После этого уже совсем просто описать способ вычисления разности целых чисел:

```
(DEFINE ZDIFFERENCE (LAMBDA (P Q) (ZPLUS P (MINUS Q))))
```

На этом мы оборвем построение нашей модели.

Конец задания.

Не должно показаться удивительным, что все эти и многие другие арифметические функции встроены в «настоящий» Лисп, при реализации которого используются все возможности современных компьютеров, начиная с естественного для них представления чисел.

Читателю не составит труда выполнить следующее

Упражнение.

Написать на Лиспе определения функций: GREATERP (> для натуральных чисел), PRIM (следующее целое), TILDE (предыдущее целое), ZLESSP (< для целых) , ZGREATERP (> для целых), AND (логическое «и»).

Задание 7.

Предполагая, что упражнение выполнено, написать Лисп-программу для вычисления числа сочетаний C(n0, m0) по формуле

(1) $C(n, m) = C(n - 1, m - 1) + C(n - 1, m)$

без вычисления ненужных промежуточных значений.

Указания.

Нужны значения C(n, m) для всех n от 1 до $n0$ и для m от 0 до min($m0$, $n \div 2$) для каждого n (здесь $\div$ обозначает знак операции деления нацело). Вычисляемые значения C(n, m) помещать в список *curr*, а значения C($n - 1$, m) хранить в списке *prev*. В пограничных случаях формула (1) не действует. При $m = 0$ следует прибегнуть к формуле

(2) C(n, 0) = 1.

При $n = 2m$ значение C($2m - 1$, m) отсутствует в списке *prev*, но оно совпадает с C($2m - 1$, $m - 1$), так что

(3) C($2m$, m) = 2 C($2m - 1$, $m - 1$).

Удобно менять значение m в разных направлениях, в зависимости от четности n, например, увеличивать при нечетном n и уменьшать при четном. Тогда при нечетном n, дойдя до значения $m = m0$, заполнение списка *curr* следует прекратить и, передав нужную его часть в *prev*, начать составлять новый *curr* с элемента C($n + 1$, $m0$), вычисляемого по формуле (1) или, при $m0 = 0$, по формуле (2).

Текст от «;» до конца строчки служит примечанием, не влияющим на работу программы. Для большей выразительности в примечаниях, в отличие от текстов на Лиспе, будем применять строчный шрифт и курсив.

Выполнение задания 7.

```
(CSETQ INIT '((1)))              ; init – список, содержащий одно число 1
(CSETQ TWO '(1 1))                                   ; two – число 2.
(DEFINE ODDC (N M PREV CURR)
                ; вычисление C(n, m) для нечетного n от m=0 до m=min(m0, n div 2)
  (COND ((AND (EQUAL N N0) (EQUAL M M0))
   (CONS N0 (CONS MM (CONS (CAR CURR) ()))) )
                ; результат – список (n0 mm (CAR curr)), где n0 и mm – значения
                 ; аргументов обращения, (CAR curr) – искомое число сочетаний
  ((EQUAL M M0)      ; далее, в зависимости от результата сравнения m0 с нулем,
                     ; используются формулы (2) или (1).
   (COND ((EQUAL M0 ()) (EVENC (PRIM N) () () INIT))
    (T (EVENC (PRIM N) M (CDR CURR)
      (CONS (PLUS (CAR CURR) (CADR CURR)) NIL) )) ))
  ((NULL (CDR PREV))   ; в prev остался один элемент, применяем формулу (3)
   (EVENC (PRIM N) (PRIM M) CURR
     (CONS (TIMES TWO (CAR CURR)) NIL)))
  (T                        ; в остальных случаях прибегаем к формуле (1)
   (ODDC N (PRIM M) (CDR PREV)
     (CONS (PLUS (CAR PREV) (CADR PREV)) CURR)) ) ))

(DEFINE EVENC (N M PREV CURR)
                ; вычисление C(n, m) для четного n от m=min(m0, n/2) до m=0
  (COND ((AND (EQUAL N N0) (EQUAL M M0))
   (CONS N (CONS MM (CONS (CAR CURR) ()))) )        ; как в ODDC
  ((NULL PREV)          ; начинаем новый список curr, используем формулу (2)
   (ODDC (PRIM N) '() CURR INIT))
  ((NULL (CDR PREV))              ; то же, но для завершения списка curr
   (EVENC N '() '() (CONS '(1) CURR)))
  (T (EVENC N (TILDE M) (CDR PREV)
   (CONS (PLUS (CAR PREV) (CADR PREV)) CURR)) ) ))
```

Головная функция:

```
(DEFINE COMB (N M) (CSETQ MM M)              ; сохранить m в mm
  (COND ((GREATERP N (PLUS M M)) (CSETQ M0 M))
```

```
    (T (CSETQ M0 (DIFFERENCE N M))))          ; задание m0 " n/2
   (EVENC '() '() NIL INIT) )                 ; начало вычислений с C(0,0)
```

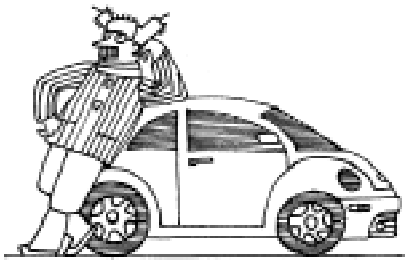
Некоторые результаты работы программы:

```
COMB(5,0) → ((1 1 1 1 1) NIL (1))                    (т. е. (5 0 1))
COMB(5,1) → ((1 1 1 1 1) (1) (1 1 1 1 1))            ( (5 1 5) и т. д.)
COMB(5,2) → ((1 1 1 1 1) (1 1) (1 1 1 1 1 1 1 1 1 1))
COMB(5,3) → ((1 1 1 1 1) (1 1 1) (1 1 1 1 1 1 1 1 1 1))
   . . .
COMB(6,3) → ((1 1 1 1 1 1) (1 1 1)
    (1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1))
                                                      ( (6 3 20) )
   . . .
COMB(6,6) → ((1 1 1 1 1 1) (1 1 1 1 1 1) (1))     ( (6 6 1) )
```

Конец задания.

Задание 8.

Составить Лисп-программу, полностью моделирующую работу машины Тьюринга (см. разд. **1** «Вычисления»).



...конкретный вариант машины Тьюринга...

Выполнение задания 8.

Чтобы зафиксировать конкретный вариант машины Тьюринга, в программу следует включить определения двух констант по следующей схеме:

(CSETQ ALPHABET '(*atom sequence*))

где атомы – это коды букв алфавита машины в нашей программе, и

(CSETQ STATESLIST '(*atom sequence*))

где тем же способом представлены все возможные состояния машины.

Для машины Тьюринга с таким алфавитом и списком состояний можно писать разные программы. Чтобы остановиться на одной из них, в Лисп-программе следует определить еще одну константу, для которой примем следующую структуру:

(CSETQ PROGRAM '(*instlist sequence*))

где все команды (инструкции), допустимые в некотором состоянии STATE, собраны в список *instlist* вида (STATE *instrest sequence*), в котором *instrest* (остаток команды) – это список вида (LETTER NEWSTATE NEWLETTER SHIFT).

Здесь STATE и LETTER – это текущие состояние и буква под головкой, которые служат для выбора исполняемой команды, NEWSTATE и NEWLETTER – новое состояние и записываемая буква, а SHIFT – признак сдвига головки.

Программа для машины Тьюринга требует задания исходного состояния и исходного слова – списка букв (элементов списка ALPHABET) в том порядке, в каком они записаны на ленте. Условимся, что исходное состояние стоит первым в списке STATESLIST. Запуск машины организует функция TURING:

```
(DEFINE TURING (LAMBDA (INPUTWORD)
 (FINDSTATE (CAR STATESLIST) NIL (CAR INPUTWORD)
            (CDR INPUTWORD) PROGRAM) ))
```

Функции FINDSTATE и FINDINSTRUCTION подготавливают очередной шаг. Аргументы функции FINDSTATE: текущее состояние STATE, перевернутый (чтобы все изменения происходили в начале, а не в конце списка) список LEFTPART содер-

жимого ленты слева от головки, буква LETTER под головкой, список RIGHTPART букв ленты справа от головки и еще не просмотренная часть PROGRAMREST программы. Функция разыскивает в этой части список инструкций для текущего состояния STATE. В случае неуспеха этого поиска работа машины завершается, а функция RESULT компонует из частей ленты результирующее слово.

```
(DEFINE FINDSTATE (LAMBDA
 (STATE LEFTPART LETTER RIGHTPART PROGRAMREST)
  (COND
   ((NULL PROGRAMREST) (RESULT LEFTPART LETTER RIGHTPART) )
   ((EQ STATE (CAAR PROGRAMREST))
    (FINDINSTRUCTION
     STATE LEFTPART LETTER RIGHTPART (CDAR PROGRAMREST) ))
   (T (FINDSTATE
     STATE LEFTPART LETTER RIGHTPART (CDR PROGRAMREST) )) )))
```

Функция FINDINSTRUCTION ищет в списке команд INSTRUCTIONSLIST для состояния STATE команду, соответствующую букве LETTER. Если команда нашлась, то происходит обращение к функции STEP для исполнения очередного шага работы машины. При неудаче вызывается функция RESULT – работа программы завершена.

```
(DEFINE FINDINSTRUCTION
 (LAMBDA (STATE LEFTPART LETTER RIGHTPART INSTRUCTIONSLIST)
 (COND
  ((NULL INSTRUCTIONSLIST) (RESULT LEFTPART LETTER RIGHTPART))
  ((EQ LETTER (CAAR INSTRUCTIONSLIST))
   (STEP (CADAR INSTRUCTIONSLIST) LEFTPART
    (CADDAR INSTRUCTIONSLIST) RIGHTPART
    (CADDDAR INSTRUCTIONSLIST) ))
  (T (FINDINSTRUCTION STATE LEFTPART LETTER RIGHTPART
(CDR INSTRUCTIONSLIST) )) )))
```

Функция STEP записывает под головкой новую букву NEWLETTER, сдвигает, если значение параметра SHIFT этого требует, головку вправо или влево (STILL – стоять на месте, RIGHT – сдвиг вправо, LEFT – влево) и переводит машину в состояние NEWSTATE. При этом, если часть ленты в направлении сдвига представлена пустым списком, то, в соответствии со сказанным в разделе **1** о моделировании бесконечной ленты, под головкой размещается пустая клетка, как бы выделенная из этого списка, остающегося пустым. Условимся, что в списке ALPHABET буква, соответствующая пустой клетке, стоит первой.

...работа программы завершена.

```
(DEFINE STEP
 (LAMBDA (NEWSTATE LEFTPART NEWLETTER RIGHTPART SHIFT)
  (COND
   ((EQ SHIFT 'STILL)
    (FINDSTATE NEWSTATE LEFTPART NEWLETTER RIGHTPART PROGRAM) )
   ((EQ SHIFT 'RIGHT)
    (FINDSTATE NEWSTATE (CONS NEWLETTER LEFTPART)
     (COND ((NULL RIGHTPART) (CAR ALPHABET))
      (T (CAR RIGHTPART)) )
     (COND ((NULL RIGHTPART) NIL) (T (CDR RIGHTPART)))
     PROGRAM) )
   ((EQ SHIFT 'LEFT)
    (FINDSTATE NEWSTATE
     (COND ((NULL LEFTPART) NIL) (T (CDR LEFTPART)))
     (COND ((NULL LEFTPART) (CAR ALPHABET)) (T (CAR LEFTPART)))
     (CONS NEWLETTER RIGHTPART) PROGRAM) ))))
```

Функция RESULT совсем проста по сравнению с предыдущими:

```
(DEFINE RESULT (LAMBDA (LEFTPART LETTER RIGHTPART)
 (REVERSE LEFTPART (CONS LETTER RIGHTPART)) ))
```

В качестве примера приведем слегка упрощенную программу сложения двух натуральных чисел из [5] (§ 5.2, пример 4). В данном варианте число $m \geq 0$ представлено участком ленты, на котором записано слово из m букв ONE. Итак:

```
(CSETQ ALPHABET '(ZERO ONE AST))
```

где буква ZERO представляет пустую клетку, AST – разделитель слагаемых

```
(CSETQ STATESLIST '(Q0 Q1 Q2)).
```

В начальном состоянии Q0 разыскивается и заменяется на ONE разделитель AST, в состоянии Q1 разыскивается правый конец ленты, в состоянии Q2 лишняя единица, возникшая в результате замены AST, стирается и работа программы завершается.

```
(CSETQ PROGRAM
 '((Q0 (ONE Q0 ONE RIGHT) (AST Q1 ONE STILL))
   (Q1 (ONE Q1 ONE RIGHT) (ZERO Q2 ZERO LEFT))
   (Q2 (ONE Q2 ZERO STILL)) ) )
```

Результаты нескольких обращений к программе TURING:

```
(TURING '(ONE ONE ONE ONE AST)) → (ONE ONE ONE ONE ZERO ZERO)
```

Одна из двух пустых клеток за концом слова-результата появилась при сдвиге за правый конец текущего слова в состоянии Q1, вторая – при замене ONE на ZERO в состоянии Q2.

```
(TURING '(ONE ONE ONE AST ONE)) → (ONE ONE ONE ONE ZERO ZERO)
(TURING '(AST ONE ONE ONE ONE)) → (ONE ONE ONE ONE ZERO ZERO)
(TURING '(AST)) → (ZERO ZERO)
```

Конец задания.

Успешное выполнение задания показало, что язык Лисп с теми возможностями, которые были описаны в разд. **3** и использованы в настоящем разделе, является не менее мощной абстрактной машиной, чем другие. Тезис Черча не позволяет ожидать большей мощности, но, чтобы окончательно в этом убедиться, следовало бы промоделировать язык Лисп, например, на машине Тьюринга. Но это – упражнение для особых любителей.

Литература.

3. *Лавров С.С., Силагадзе Г.С.* Автоматическая обработка данных. Язык Лисп и его реализация / Библиотечка программиста. М.: Наука, Физматлит, 1978.

4. *Марков А.А., Нагорный Н.М.* Теория алгорифмов. М.: Наука, Физматлит, 1984.

5. *Мендельсон Э.* Введение в математическую логику. / Пер. с англ., 2-е изд. М.: Наука, Физматлит, 1976.

6. *Минский М.* Вычисления и автоматы. / Пер. с англ. М.: Мир, 1971.

Лавров Святослав Сергеевич,
доктор технических наук.
профессор.