Городняя Лидия Васильевна,
Лавров Святослав Сергеевич

ФУНКЦИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ

Эта статья продолжает публикации второго автора под тем же заглавием из номеров 2 и 3–4 журнала за этот год. Для удобства нумерация разделов и заданий – сквозная. Продолжена и нумерация списка литературы.

5. ПРИНЦИПЫ РЕАЛИЗАЦИИ ЯЗЫКА ЛИСП

Традиционно система программирования для языка Лисп, как и для многих других языков, содержит пару интерпретатор-компилятор. Грубо говоря, *интерпретатор* хранит текст программы в исходном виде. Просматривая этот текст, интерпретатор исполняет одну его единицу за другой. По мере необходимости исполнение каждой единицы повторяется или начинается заново, не будучи завершено. *Компилятор* же преобразует весь текст в программу на машинном языке и передает компьютеру работу по ее исполнению.

На самом деле между этими подходами трудно установить формальную границу. Любой интерпретатор содержит в себе элементы, реализация которых описывается в машинных терминах: структура памяти, подпрограммы некоторых базовых функций, первичная обработка текста программы и т. п. Любая компилированная программа содержит интерпретируемые элементы: например, обращения к файловой системе и другим элементам ОС. На практике достоинства интерпретации проявляются при отладке программ, а преимущества компиляции – при эксплуатации готового программного продукта. Более подробно обсуждать эту тему не будем.

Реализация как интерпретатора, так и компилятора для Лиспа достаточно подробно описана в работе [1].

В интерпретаторе можно выделить следующие крупные части:

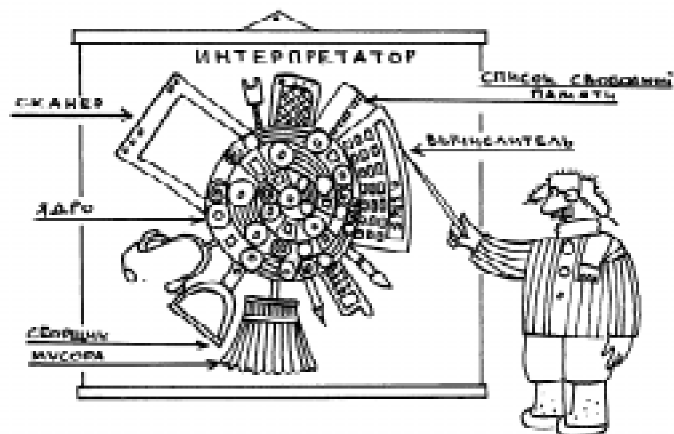
– программу предварительного просмотра текста, часто называемую *лексическим анализатором* или *сканером*,

– исполнитель программы в ее внутреннем, машинном, представлении, построенном сканером, это и есть собственно интерпретатор, называемый также *вычислителем*,

– машинно-ориентированное *ядро*, поддерживающее работу вычислителя,

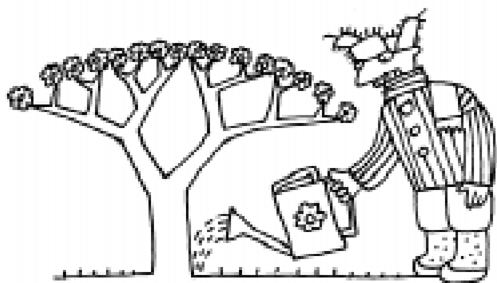
– подпрограммы ввода-вывода лисповских выражений с преобразованием их во внутреннее представление и обратно,

– подпрограмму управления памятью, ее существенные части – *список свободной памяти* и *сборщик мусора*.



В интерпретаторе можно выделить следующие крупные части.

В этой статье будет более детально рассмотрен и описан



Внутреннее представление списков и атомов любого класса основано на двоичных деревьях.

средствами базового Лиспа только вычислитель. Остальные части интерпретатора довольно сильно зависят от компьютера, на котором они реализуются, и поэтому будут описаны только в самых общих чертах. Мы не будем рассматривать реализацию функциональных аргументов. Обычно Лисп-интерпретатор предусматривает их использование, но сейчас это чрезмерно загромоздило бы изложение.

Жаждущим пополнить свои знания рекомендуются источники [5] и [6].

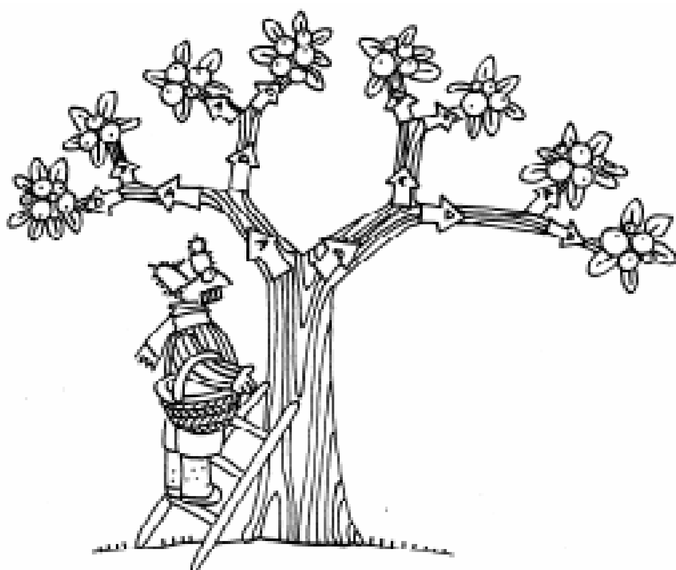
5.1. ВНУТРЕННЕЕ ПРЕДСТАВЛЕНИЕ ДАННЫХ

Внутреннее представление списков и атомов любого класса основано на двоичных деревьях. Степень каждой их нетерминальной вершины (узла) равна двум. Такая вершина соответствует одному элементу списка. Левое из подвешенных к ней поддеревьев представляет этот элемент, правое – продолжение списка. Пример дерева, соответствующего списку (A (B C)) показан на рисунке 1.

Каждый лист (терминальная вершина) дерева представляет атом. При этом лист, подвешенный к узлу справа, может быть только атомом NIL, символизирующим пустоту продолжения списка.

На более низком уровне каждой вершине двоичного дерева соответствует машинная ячейка. Структура такой ячейки практически одна и та же для узлов и листьев дерева. Ячейка содержит два *регистра* (A- и D-регистр), иначе называемых *указателями*, и несколько *битов признаков*. В Лиспе принято обозначение (X.Y) для ячейки, содержащей в своих регистрах значения X и Y. Такое выражение называется *точечной парой*.

Число разрядов регистра достаточно для хранения адреса любой ячейки. A-регистр узла указывает на левое поддерево этого узла, D-регистр – на правое. A-регистр листа указывает на имя атома в программе (*внешнее имя*), D-регистр – на значение атома. Такая ячейка называется также *информационной ячейкой* (ИЯ) атома. Содержимое одного из битов призна-



Ячейка содержит два регистра (A- и D-регистр), иначе называемых указателями...

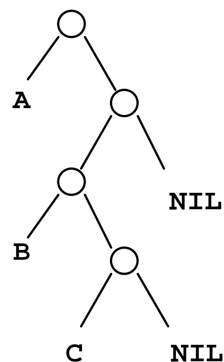


Рисунок 1

ков помогает отличать листья от узлов. Имеет смысл также отличать константы от переменных, имена функций от иных имен, встроенные функции от определяемых, обычные от особых и т. п. Еще один бит служит для пометки ячеек, попадающих в мусор.

Например, для констант T и NIL их ИЯ содержат свои собственные адреса в D-регистре и признаки атома и константы в соответствующих битах.

Для организации связи атомов с приписанными им значениями можно использовать и другой механизм – так называемый ассоциативный список, построенный из ячеек, содержащих указатели на ИЯ имен аргументов функций и их текущие значения. Например, ассоциативный список ((ONE.1) (TWO.2)) сопоставляет числовые значения символьным именам, а список ((HEAD.CAR) (TAIL.CDR) (DOT.CONS)) мог бы хранить синонимы для имен базовых операций Лиспа на участке программы, использующем эти операции в качестве функциональных аргументов, но вопросы реализации последних авторы обещали не затрагивать.

Список ((T.T) (NIL.NIL)) мог бы указывать на значения констант T и NIL в соответствии с семантикой базового Лиспа. Однако информационные ячейки этих атомов гораздо проще и быстрее приводят к той же цели. Чем более загружен текущий ассоциативный список, тем медленнее обнаруживались бы в нем эти значения.

Поэтому ассоциативный список обычно применяется там, где он действительно необходим – для хранения текущих значений аргументов функций. Ассоциативный список работает как стек: при многократных определениях работает самое новое, то есть расположенное ближе к началу списка. Например, ассоциативный список может принять структуру:

```
((ONE.1) (TWO.2) (ONE.FIRST) (TWO.SECOND))
```

если функция ORDINAL с именами аргументов ONE и TWO обращается к функции CARDINAL, использующей те же имена. Тогда для аргумента внутреннего обращения (последнего по времени) выбирается числовое значение, а значение в виде порядкового числительного будет доступно лишь после завершения внутреннего обращения и возврата в ORDINAL, когда ассоциативный список вновь примет вид:

```
((ONE.FIRST) (TWO.SECOND))
```

возникший при внешнем обращении.

5.2. РЕАЛИЗАЦИЯ НЕКОТОРЫХ ВСТРОЕННЫХ ФУНКЦИЙ

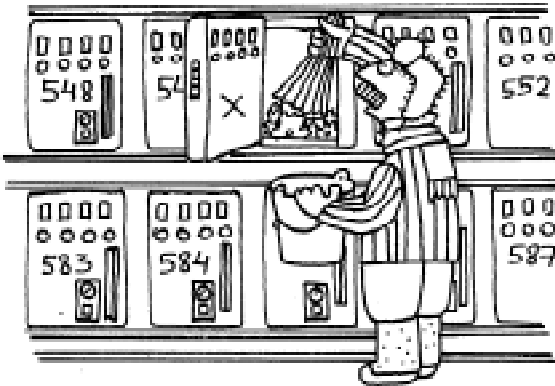
Значения аргумента или аргументов встроенной функции – это адреса ячеек, соответствующих элементам списков или атомам. Функции CAR и CDR извлекают, соответственно, содержимое, то есть также адрес, из A- или D-регистра такой ячейки. Имя CAR означает Contents of A-Register, аналогично для CDR. Значение CAR – это адрес элемента списка или адрес внешнего имени атома. Для CDR – это адрес продолжения списка или адрес значения константы. В базовом Лиспе применение этих функций к атомам не предусмотрено.

Значение функции ATOM определяется содержимым соответствующего бита признака ячейки. Аналогично могут быть реализованы и некоторые другие предикаты, необходимые для работы вычислителя. Они должны проверять значения других битов признаков или наличие того или иного сочетания этих значений.

Функция EQ просто проверяет равенство значений своих аргументов.

Заметно сложнее реализуется функция CONS. Прежде всего, она проверяет, не пуст ли список свободной памяти. Если пуст, то запускается процедура сборки мусора, которая может сделать этот список непустым. Если сделать это не удалось, то работа Лисп-программы аварийно завершается с выдачей соответствующего сообще-

ния. При удаче из списка свободной памяти извлекается его первая ячейка, в ее A- и D-регистры заносятся аргументы обращения к CONS, в один из битов признаков – код списочной ячейки, (то есть не атома), а адрес ячейки становится значением функции CONS. Таким образом, CONS может строить как списочные ячейки, так и точечные пары. Любой список можно переписать в точечных обозначениях. Например, список (A (B C)) в виде: (A.((B.(C.())).())). Сравните эту запись с рисунком из раздела 5.1. Возможны смешанные обозначения. Но мы ударились в детали, не слишком нужные для наших целей.



Сборка мусора начинается с пометки ячеек.

Сборка мусора начинается с пометки ячеек, достижимых из сравнительно небольшого числа рабочих ячеек системы программирования. К ним относятся ячейки с промежуточными результатами вычислений, активная часть стека, ассоциативный список, информационные ячейки атомов и др. Мусорщик очищает память от всех непомеченных ячеек, которые уже никогда не смогут быть использованы вычислителем, и возвращает их в список свободной памяти.

Стек необходим и для хранения адреса возврата после вычисления функции, для запоминания указателя на предыдущую вершину стека и т. п. Все задачи стека можно решить с помощью ассоциативного списка. Но мы не будем говорить об этом подробнее.

Функция QUOTE, очевидно, должна извлечь второй элемент списка (QUOTE *expr*), служащего обращением к ней. Ее реализация включена в модель вычислителя выражений Лиспа (см. ниже, раздел 5.4).

Функция CSETQ должна поместить значение своего второго аргумента в D-регистр ячейки, на которую указывает ее первый аргумент. Кроме того, она должна пометить в одном из битов признаков этой ячейки, что соответствующий атом отныне становится константой.

Функция DEFINE реализуется аналогично, но она должна также проверить, хотя бы поверхностно, что значение второго аргумента – это действительно LAMBDA-конструкция. Пометка делается в другом бите признака.

Такая реализация функций CSETQ и DEFINE не препятствует использовать имена определяемых в программе констант и функций в качестве имен аргументов функций. Но прибегать к таким приемам (в программировании их часто называют грязными) настоятельно не рекомендуется – многие существующие реализации Лиспа могут схватить вас за руку и высказать все, что думают о вас «старшие товарищи».

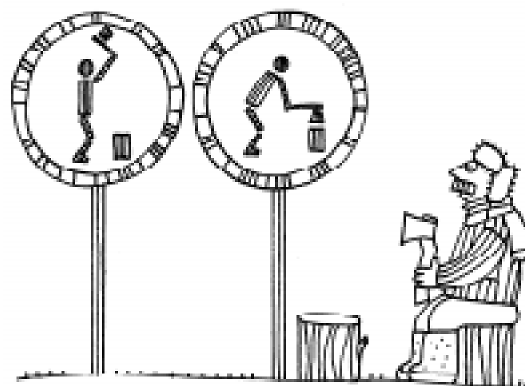
Пополнять ассоциативный список можно операцией CONS или описанной в предыдущем разделе функцией APPEND, но лучше для этого завести более удобную функцию PAIR. Функция ASSOC ищет в ассоциативном списке элемент (<имя аргумента>.<его значение>).

Пусть X $\rightarrow$ (X_1 X_2,..., X_m) – список имен аргументов функции *fn*, Y $\rightarrow$ (Y_1 Y_2,..., Y_n) – список значений начальной части этих аргументов, $m \geq n$, *al* – текущий ассоциативный список. Тогда, в соответствии с разделом 4:

$$\text{(PAIR X Y al)} > ((X_1.Y_1)\ (X_2.Y_2)...(X_n.Y_n)\ (X_{n+1}.\text{NIL})...(X_m.\text{NIL})\ \text{al})$$

В ассоциативный список занесено все необходимое для начала вычисления функции *fn*. По завершении вычисления функции *fn* ассоциативный список *al* должен быть восстановлен в прежнем виде.

Приведенная выше связь между общим видом обращения к PAIR и результатом его исполнения вместе со следующими за этим фразами дает достаточно полную спецификацию этой функции. В общем случае под спецификацией как раз и понимают исчерпывающее описание связи между исходными данными для вычисления функции и получаемым результатом.



...описание делается на языке, более близком и понятном человеку...

Предполагается, что такое описание делается на языке, более близком и понятном человеку, чем язык, на котором предполагается описать саму функцию. Несколько примеров на вычисление функции обычно не обладают достаточной полнотой. Модель интерпретатора, приведенная в этой статье, как и интерпретатор и компилятор, описанные в книге [1], – это тоже спецификации своего рода.

Правила:

(ASSOC 'X$_i$ '(... (X$_i$.Y$_i$) ...))) → (X$_i$.Y$_i$)

при **X$_i$**, не входящем в **X$_1$,...,X$_{i-1}$,**

(ASSOC 'X NIL) → NIL при любом атоме **X**

завершают спецификацию функции ASSOC.

Упражнение.

Написать на базовом Лиспе определение функций PAIR и ASSOC.

Решение.

```
(DEFINE PAIR (LAMBDA (xl yl al) (COND
  (xl (COND
    (yl (CONS (CONS (CAR xl) (CAR yl))
```

– здесь и ниже внутреннее обращение к CONS создает точечную пару,

```
      (PAIR (CDR xl) (CDR yl)) ))
    (T (CONS (CONS (CAR xl) NIL) (PAIR (CDR xl) NIL) ))
  (yl (ERROR 'value_list_is_too_long))
  (T al) )))
```

ERROR – псевдофункция, выдающая заданное ей диагностическое сообщение и вырабатывающая значение NIL.

```
(DEFINE ASSOC (LAMBDA (x al) (COND
  ((EQ x (CAAR al)) (CAR al))
  (al (ASSOC x (CDR al)))
  (T NIL) )))
```

Таким образом, информационные ячейки естественно возникают как машинные эквиваленты листьев двоичного дерева, представляющего выражение. Здесь, говоря о естественности, авторы слегка покривили душой. Но при построении абстракций правда жизни вынужденно страдает чаще многого другого.

5.3. ВВОД-ВЫВОД И СКАНЕР

Если мы располагаем представлением всех данных программы в описанном в разделе 5.1 виде, то организовать вывод на экран или на печать любого выражения не составляет труда. При выводе списка надо вывести открывающую скобку, затем все элементы списка и, наконец, закрывающую скобку. При выводе атома – вывести его внешнее имя, доступ к которому предусмотрен в ИЯ. Надо еще своевременно переходить с одной строки выводимого текста к другой, но это скорее эстетическая, чем техническая, проблема.

При вводе выражений для каждого вводимого атома надо определить, возник ли он впервые, или уже встречался ранее. Иначе говоря – нужно ли создавать для него ИЯ или нет. Для этого нужен механизм поиска ИЯ по внешнему имени *name* – последовательности C_1, ..., C_k составляющих его литер. Простой метод – составить таблицу или список точечных пар (<имя>.<адрес ИЯ>) и каждый раз ее (его) просматривать – работает медленно.

Обычно прибегают к помощи функции расстановки (*hash*-функции). При проектировании интерпретатора грубо оценивается максимальное число L атомов в интерпретируемых программах, скажем, L = 1024. Функция hash сопоставляет каждому имени число из диапазона от 0 до L-1. Создается массив из L указателей на списки имен с одним и тем же значением *I = hash(name)*. Таким приемом таблица, упомянутая в предыдущем абзаце, разбивается на L коротких таблиц со средней длиной меньше 1. Полученная структура называется *расстановочным полем*, а упомянутый массив – его *оглавлением*. Просмотр *I*-й таблицы осуществляется очень быстро. Если имя name в ней не встретилось, в частности, если таблица пуста, то атом ни в программе, ни во вводимом по ходу ее исполнения атоме ранее не появлялся. Создается, заполняется и присоединяется к таблице новая пара. Если имя встретилось, то вместе с ним обнаруживается и адрес ИЯ. Метод тем более эффективен, чем равномернее распределены возможные значения функции *hash* в упомянутом диапазоне.

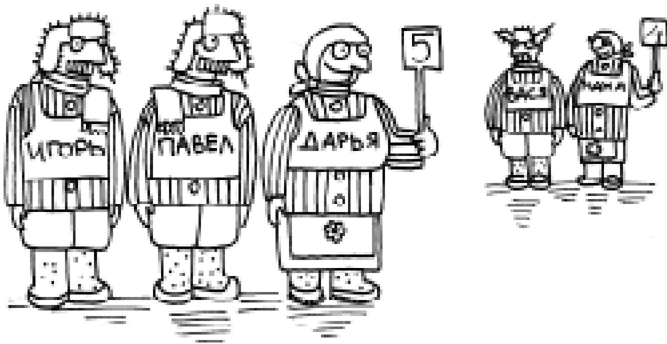
Упражнение.

Предложить способ вычисления функции расстановки.

Решение не может быть однозначным. Чаще всего выбирается вид коэффициентов K_1, K_2, ..., и I вычисляется по формуле $I = (\sum_{j=1}^{k}(K_j code(C_j)) \bmod L$, где code(C) – код литеры C, например, в системе ASCII. Выбор K_j=1 слишком примитивен, но приемлем, если L выбрано с достаточным (в среднем) запасом. Выбор K_j=j обычно дает вполне удовлетворительный результат.

В начале создания сканера следует заполнить расстановочное поле информационными ячейками всех атомов, определенных в языке Лисп. Итак, способ ввода атомов описан.

Для каждого вводимого элемента списка должна быть создана новая списочная ячейка – узел двоичного дерева. Сразу после завершения построения левого и правого поддеревьев



Функция hash сопоставляет каждому имени число...

этого узла адреса их вершин с помощью CONS помещаются в A- и D-регистры такой ячейки. Сказанным исчерпывается краткая спецификация процедуры ввода.

Сканер отличается от нее лишь тем, что по ходу ввода текста программы надо исполнять все содержащиеся в ней обращения к функциям CSETQ и DEFINE для формирования ИЯ констант и функций. Достаточно проделать это лишь один раз, так что эти обращения уже не нужно включать во внутреннее представление программы.

5.4. РЕАЛИЗАЦИЯ ОПРЕДЕЛЕНИЙ ФУНКЦИЙ И ИНТЕРПРЕТАТОРА В ЦЕЛОМ

Построение модели вычислителя начнем с описания нескольких вспомогательных функций. В разделе 4 была подробно описана семантика COND-конструкции вида (COND ($bool_1$ $expr_1$) ...). Ее вычисление с учетом состояния ассоциативного списка выполняет входящая в модель интерпретатора функция EVCON, выбирающая из списка ветвей нужную. Например,

```
(EVCON '((T YES) ...) '((T.T)  (NIL.NIL))) → YES,
(EVCON '((NIL NO) (T YES) ...) '((T.T) (NIL.NIL))) → YES,
(EVCON '( ) '((T.T))) > NIL,
(EVCON '((T YES) (1 ONE) ...) '((T.NIL) (NIL.NIL))) → ONE,
```

– плачевное последствие изменения значения атома T,

```
(EVCON '((NIL NO) (Y TRUE) ...)  '((Y.T) (NIL.NIL))) → TRUE,
(EVCON '((NIL NO) (Y TRUE))  '((Y.NIL) (NIL.NIL))) → NIL,
```

поскольку ни одна из ветвей не была выбрана.

Упражнение.

Пусть EVCON – имя функции, вычисляющей значение COND-конструкции E по списку ее ветвей. Напишите выражение, вырабатывающее значение E с учетом значений атомов, связанных с помощью ассоциативного списка AL.

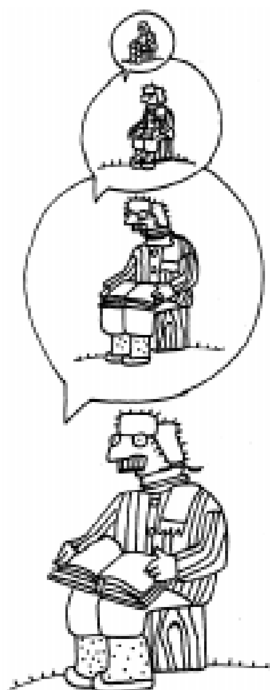
Решение:

```
(EVCON (CDR E) AL).
```

Здесь, в результате обращения к CDR, от E сохраняется только полный список ветвей. Интерпретация функции EVCON будет описана немного ниже. Семантика условного выражения была описана в разделе 4 достаточно подробно, а результаты работы EVCON продемонстрированы на многих примерах.

Этот прием – включение в программу специфицированных, но не описанных окончательно процедур – не имеет прямого отношения к функциональному программированию. Он оказывается вынужденным, когда используемые процедуры (в частности, функции), вовлечены в косвенную рекурсию. При этом и процесс чтения текста процедуры человеком становится, условно говоря, рекурсивным – перечитывай, пока не разберешься. Хорошая словесная спецификация может его облегчить.

Более сложные случаи вычисления выражений строятся как результат применения стоящей в голове списка функции к



...процесс чтения текста процедуры человеком становится... рекурсивным

ее аргументам, расположенным в этом же списке. Функция может быть представлена именем или LAMBDA-конструкцией.

Это выполняется другой важной частью определения интерпретатора – функцией APPLY, которую мы также вскоре определим. Она применяет заданную первым аргументом функцию к списку аргументов, представленному вторым аргументом, при данном в третьем аргументе ассоциативном списке. Для ее работы необходима функция, вычисляющая список значений выражений из заданного списка с учетом состояния ассоциативного списка. Согласно лисповской традиции, назовем ее EVLIS. Например:

```
(EVLIS '(A NIL B (CAR '(1 2))) '((B.BINARY) (A.ATOM) (T.T))) →
      → (ATOM NIL BINARY 1)
```

Упражнение.

Напишите определение функций EVLIS, EVCON в предположении, что функция EVAL определена согласно вышеприведенной спецификации.

Решение:

```
(DEFINE EVLIS (LAMBDA (xl al) (COND
  (xl (CONS (EVAL (CAR xl) al) (EVLIS (CDR xl) al)))
  (T NIL) )))
(DEFINE EVCON (LAMBDA (bl al) (COND
  (bl (COND
     ((EVAL (CAAR bl) al) (EVAL (CADAR bl) al))
     (T (EVCON (CDR bl) al)) ))
  (T NIL) )))
```

Задание 9.

Опираясь на сказанное выше и на следующие примеры

```
(APPLY  CAR '((1 2)) al) → 1
(APPLY  '(LAMBDA (x) (CAR x)) '((1 2)) al) → 1,
```

написать определение функции APPLY при условии, что функция CALL осуществляет применение встроенных функций к их аргументам, заданным в виде списка значений.

Выполнение задания 9:

```
(DEFINE APPLY (LAMBDA (fn args al) (COND
  ((ATOM fn) (COND
    ((MEMBER fn '(CAR CDR CONS ATOM EQ))
```

Это условие можно заменить проверкой кодов признаков атома *fn*,

```
        (EXEC (CONS 'fn args)) )
```

где EXEC исполняет (executes) выражение, переданное ей в качестве обращения к встроенной функции,

```
    (T (APPLY (CDR fn) args al)) ))
```

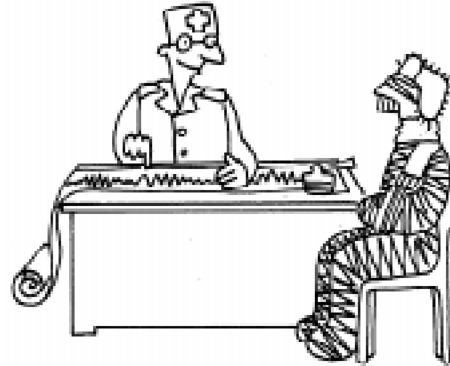
– случай, когда функция *fn* должна была быть определена в Лисп-программе, и сканер занес определяющее выражение *fn* в D-регистр ее информационной ячейки (это также можно проверить, обследуя код признака ИЯ),

```
  ((EQ (CAR fn) 'LAMBDA)
    (EVAL (CADDR fn) (PAIR (CADR fn) args al)) )
```

– случай функции, заданной ее определяющим выражением

```
  (T (ERROR 'undefdfunction)) )))
```

Все разрешенные способы задания были охвачены, и диагностика для пользователя здесь, как и во внутренней COND-конструкции, формально не нужна, но не ошибается тот, кто ничего не делает. Некоторые диагностические сообщения больше помогают разработчику интерпретатора, чем пользователю.



Некоторые диагностические сообщения больше помогают разработчику ... чем пользователю.

Основой реализации интерпретатора является функция EVAL (evaluate), вычисляющая произвольные выражения языка с учетом состояния ассоциативного списка AL. Спецификация такой функции для базового Лиспа достаточно полно была дана в разделе 4 и может быть дополнена следующими примерами:

```
(EVAL NIL AL) → NIL
(EVAL T AL) → T
(EVAL 'X AL) → (CDR (ASSOC X AL))
(EVAL '(QUOTE EXPR) AL) → EXPR
(EVAL '(COND ((T YES) ...)) '((T.T) ...) ) → YES
(EVAL '(CAR A) '((A 1 2 3) (NIL.NIL)) ) → 1
```

– здесь учтено, что (A 1 2 3) обозначает то же, что (A.(1 2 3)).

Задание 10.

Написать на базовом Лиспе определение функции EVAL, способной от данного списочного представления выражения *e* перейти к его значению с учетом заданного ассоциативного списка *al*, хранящего значения атомов.

Выполнение задания 10:

```
(DEFINE EVAL (LAMBDA (e al) (COND
  ((ATOM e) (COND
    ((CONSTP e) (CDR e))
    (T (LAMBDA (v)
      (COND (v (CDR v)) (T (ERROR 'undefdvalue))) )
        '(ASSOC e al) ) )
  ((EQ (CAR e) 'QUOTE) (CADR e))
  ((EQ (CAR e) 'COND) (EVCON (CDR e) al))
  (T (APPLY (CAR e) (EVLIS (CDR e) al) al)) )))
```

Здесь CONSTP – функция, проверяющая по кодам признаков, является ли ее аргумент константой.

Это определение дает значение выражений в случаях, если выражение является именем константы, значением переменной (по допущению, любого атома – не константы), значением в кавычках (случай замены QUOTE на ' не предусмотрен), условным выражением или же обращением к произвольной функции при вычисленных аргументах.

Здесь авторы уклонились от включения диагностики ошибок во внешние, а не в самую внутреннюю COND-конструкцию. Использование LAMBDA-конструкции позволяет исключить двойной поиск значения переменной в ассоциативном списке.

Можно еще поработать с таким определением интерпретатора и более четко локализовать его зависимость от различных категорий объектов: самоопределимые атомы (NIL, T, 1, 2, ... и иные константы), обычные встроенные функции, обрабатыва-

Желающие могут поэкспериментировать с самодельным интерпретатором...

ющие предварительно вычисленные аргументы (CAR, CDR, CONS, ATOM, EQ, ...), особые функции, управляющие обработкой аргументов без их предварительного вычисления (QUOTE, CSETQ, COND, DEFINE,...) и конструкторы функций, строящие функциональные значения из обычных выражений (LAMBDA, ...).

Желающие могут поэкспериментировать с самодельным интерпретатором, превращая его в минимодель ядра любого языка программирования, используя какую-нибудь Лисп-систему, например, GNU Clisp [7] (с точностью до имен отдельных функций: DEFINE – defun и т. п.).

Упражнение.

Пусть READ и PRINT – встроенные функции, обеспечивающие прием с клавиатуры и вывод на экран произвольных данных языка Лисп (см. раздел 5.3). Напишите определение рабочего цикла интерпретации последовательности выражений.

Решение.

```
(DEFINE CIRCLE (LAMBDA (al) (COND
  ((REQUEST 'input_file_is_empty) NIL)
  (T (PRINT (EVAL (PRINT (READ)) al)) (CIRCLE al)) )))
```

Здесь REQUEST – запрос к операционной системе на проверку ситуации, описанной в аргументе.

На этом завершается описание принципов реализации языка Лисп. На протяжении десятилетий Лисп зарекомендовал себя как простая и удобная операционная среда функционального программирования. Дж. Мак-Карти блестяще воплотил в нем идею использования рекурсивных функций символьных выражений как аппарата решения разнообразных задач, которые было модно относить к сфере искусственного интеллекта.

Литература.

5. *Хендерсон П.* Функциональное программирование. Применение и реализация. Пер. *Петровой Л.Т.* под ред. *Ершова А.П.* М.: Мир, 1983.

6. *Филд А.*, *Харрисон П.* Функциональное программирование. М.: Мир, 1993.

7. www.lisp.org – сайт ассоциации пользователей Лиспа, с которого можно скачать свободно распространяемую реализацию GNU Clisp для эксперимента.

Городняя Лидия Васильевна,
канд. физ.-мат. наук,
старший научный сотрудник
Института систем информатики
им. А.П. Ершова, Новосибирск.

Лавров Святослав Сергеевич,
доктор технических наук.
профессор.

Наши авторы, 2002.
Our authors, 2002.