РЕФАЛ-5

РУКОВОДСТВО ПО ПРОГРАММИРОВАНИЮ И СПРАВОЧНИК

Турчин В.Ф.

http://refal.ru/rf5_frm.htm

Оглавление

1. БАЗИСНЫЙ РЕФАЛ

1.1. ОПРЕДЕЛЕНИЕ ПРОСТОЙ ФУНКЦИИ

Рассмотрим следующую простую задачу. Вы хотите задать функцию, которая определяет для любой заданной строки символов, является ли она *палиндромом***.** Палиндром - это строка, одинаково читаемая как слева направо, так и справа налево. Имеются два очевидных способа уточнить такое определение палиндрома. Первый - это определить процедуру реверсирования строки (записи посимвольно задом наперёд), а затем сравнивать реверсированную строку с исходной. Очевидно, алгоритм, создаваемый на основе такого уточнения, будет не самым эффективным. Если строка оканчивается символом, отличным от начального, то она не является палиндромом, а это может быть установлено без реверсирования всей строки. Поэтому мы примем за основу другое уточнение, которое можно записать в обычной математической форме:

1) Пустая строка является палиндромом.

2) Строка из одного символа является палиндромом.

3) Если строка начинается и оканчивается одним и тем же символом, то она является палиндромом тогда и только тогда, когда строка, полученная из неё путём удаления начального и конечного символов, является палиндромом.

4) Если не выполнено ни одно из вышеприведённых условий, строка палиндромом не является.

Пусть **`Pal`** - имя функции, которую мы хотим определить, и пусть она принимает значение **`True`** в случае, когда её аргумент - палиндром, и **`False`** в противном случае. Определение в РЕФАЛе почти буквально следует математическому определению:

```
Pal { = True;
    s.1 = True;
    s.1 e.2 s.1 = <Pal e.2>;
    e.1 = False;  }
```

Эта программа начинается с имени функции, за которым следует *блок*, заключенный в фигурные скобки. Блок - это список *предложений*, разделяемых точкой с запятой. В нашем случае имеем четыре предложения, каждое из которых соответствует одному пункту в приведённом выше математическом описании палиндрома. Предложение представляет собой правило замены. Его левая часть представляет образец аргумента функции, правая же часть после знака равенства является заменой для вызова функции.

В первом предложении аргумент функции **`Pal`** — пустая строка, поэтому значением функции должно быть **`True`** .

Во втором предложении аргумент задан как *свободная переменная символа* **`s.1`**. Этому образцу будет удовлетворять любой символ, например, буква, но только *один* символ. Таким образом, предложение читается: если аргумент - единственный символ, заменить вызов на **`True`** .

Третье предложение представляет случай, когда аргумент начинается и оканчивается одним и тем же символом **`s.1`**, и содержит любое *выражение* **`e.2`** между ними. Выражение является самой общей структурой данных в РЕФАЛе; в частности, оно может быть строкой символов. Правая часть этого предложения есть вызов функции **`Pal`** от аргумента **`e.2`**. Будем использовать круглые скобки для

формируемых структур данных, а угловые — для обозначения вызова функции. Записываем **<Pal e.2>** в случае, который соответствует математическому обозначению ***pal*(e$_2$)** .

Наконец, аргументом в последнем предложении является произвольное выражение (строка). Каждое предложение в определении функции в РЕФАЛе применяется только тогда, когда ни одно из предшествующих предложений не было применимо. Таким образом, последнее предложение означает: если ни один из вышерассмотренных случаев не имел места, тогда при любом аргументе значением вызова функции является **False** .

Точное задание алгоритмической семантики РЕФАЛ-программ даётся через определение *абстрактной РЕФАЛ-машины* (см. Разд. 1.5), которая выполняет РЕФАЛ-программы, т.е. вычисляет функции, определяемые в РЕФАЛе. РЕФАЛ-машина работает пошагово, причём каждый шаг состоит в выполнении одного предложения. Если, например, мы даём РЕФАЛ-машине вызов **Pal** с аргументом **'revolver'**, т.е. **<Pal 'revolver'>**, то рабочее пространство РЕФАЛ-машины (которое мы будем называть *полем зрения*) проходит через следующие стадии:

```
<Pal 'revolver'>
 <Pal 'evolve'>
  <Pal 'volv'>
   <Pal 'ol'>
     False
```

Это демонстрирует детальную картину процесса вычисления. Программист может управлять этим процессом с помощью трассировщика. В простейшем случае он может просматривать и выдавать на печать все последовательные стадии процесса, как это было описано выше.

1.2. СИМВОЛЫ И ВЫРАЖЕНИЯ

Теперь начнём систематическое определение РЕФАЛа. Краткое описание синтаксиса языка приведено также в Разделе справочника B.

Символ в РЕФАЛе — это минимальный синтаксический элемент структур данных. Используются следующие виды символов:

- символы—имена, называемые также *идентификаторами*;
- знаки клавиатуры;
- неотрицательные целые числа (*макроцифры*);
- действительные числа.

Идентификатором является последовательность знаков, которая начинается с заглавной буквы и может включать буквы, цифры, чёрточки - и подчёркивания _. Размер букв, за исключением первой, не играет роли; так, **SUM1** и **Sum1** представляют один и тот же идентификатор. Чёрточки и подчёркивания также взаимозаменяемы: **Sum-1** и **Sum_1** означают одно и то же. Идентификаторы не должны разделяться пробелами и знаками переноса. Общая длина идентификатора не должна превышать 15 знаков.

Правильными идентификаторами являются:

```
A    X1    Y2g3h56c    CrocoDile    Hit-and-run
```

Не являются правильными идентификаторами:

```
a    x1    16x    Me+You    Input/output
```

Все печатные знаки компьютерной системы могут использоваться в качестве символов. Чтобы отличать знаки от символов-имён, мы будем заключать первые в кавычки. Для выделения строки знаковых символов будем заключать в кавычки всю строку. Могут использоваться как одинарные **'**, так и двойные **"** кавычки, но открывающая и закрывающая кавычки должны быть одного типа. Таким образом, **'A'** есть один символ, в то время как **'A+B'** есть строка (последовательность) трёх символов: знаков **'A'**, **'+'** и **'B'**.

Для представления самих кавычек мы дублируем их, независимо от того, изолированы они или находятся внутри строки, ограничиваемой того же вида кавычками. Так **'*,'''** является строкой из трёх символов: звёздочки, запятой и одинарной кавычки. Это может быть также записано как **"*,'"**.

Односимвольные идентификаторы не следует путать с алфавитными символами: **A** отличается от **'A'**. Алфавитные символы разных регистров также различаются: **'A'** отлично от **'a'**.

Макроцифрами в нашей реализации РЕФАЛа-5 являются положительные целые числа в диапазоне от 0 до 2^{32}-1. Превосходящие 2^{32}-1 числа могут быть составлены из макроцифр с использованием основания 2^{32} , так же как десятичные целые числа составляются из обычных (десятичных) цифр; этим и объясняется имя "макроцифра". Для представления отрицательных целых чисел мы помещаем **'-'** перед цифрами, так же, как это делается при использовании десятичных цифр. Подобно буквам алфавита, которые отличаются от букв-идентификаторов, цифры отличаются от числовых символов. **'1'** — это не то же самое, что **1**. В отличие от первой единицы — обычной десятичной цифры — вторая единица рассматривается как макроцифра.

Примеры: **3306** является символом-макроцифрой с численным значением 3306. **'-'25** является последовательностью двух символов; за алфавитным символом **'-'** следует макроцифра **25**. Вместе они будут восприниматься арифметическими функциями как число -25. (В программах все знаки должны заключаться в кавычки; при считывании данных функцией **Input** [см. Разд.2.3] кавычки не являются обязательными.) В следующем примере:

```
2543 88918 9
```

является последовательностью трёх макроцифр, которая будет пониматься как

$$2543*2^{64} + 88918*2^{32} + 9$$

ЗАМЕЧАНИЕ: Если вы записываете нечто вроде **'--'25** , это не будет считаться синтаксической ошибкой. Это вполне допустимая строка из трёх символов. Ошибочной она станет только тогда, когда вы попытаетесь использовать такую строку в качестве операнда в арифметической функции.

Действительное число в языке РЕФАЛ-5 всегда представлено символом, занимающим одно машинное слово. Действительное число может быть положительным, отрицательным или нулём. В программе действительные числа записываются в обычном виде с точкой, отделяющей дробную часть от целой, и с знаком **E**, отделяющим десятичный порядок. Примеры:

```
215.73
-18E+15
0.003E-7
```

(см. в Разделе справочника B более подробный синтаксис). Символы, представляющие действительные числа, отличаются от макроцифр: **215** - не то же самое, что **215.0**.

Вообще говоря, пробелы не считаются символами; пробелы и символы переноса используются для

разделения лексических единиц РЕФАЛа всюду, где это необходимо, и для более удобного размещения их на странице. Единственной ситуацией, когда пробел становится символом, является случай его использования внутри кавычек. Если `''` - это символ кавычек, то `' '` - это символ пробела. При использовании между строками, заключёнными в кавычки, пробел говорит о том, что снять кавычки можно только с каждой из них по отдельности. Так, **`'a' 'b'`** является строкой из двух символов, **a** и **b** , в то время как **`'a''b'`** - это строка из трёх символов: **a**, кавычки и **b**.

Для создания структур данных в РЕФАЛе используются скобки. Не заключаемые в кавычки круглые скобки — это не символы, но *специальные знаки* РЕФАЛа. Мы также называем их *структурные скобки* в отличие от угловых *вычислительных* скобок. Структурные скобки следует объединять в пары согласно общепринятым простым правилам. Будем называть произвольную последовательность символов, включающую и правильно расставленные скобки, *объектным выражением.* Более точно, объектное выражение — это последовательность конечного числа *термов*, где каждый терм является либо одним символом, либо выражением, заключённым в скобки. Количество термов в выражении может быть и нулём, так что пустое объектное выражение (буквально ничего не содержащее) является допустимым. Вот некоторые примеры выражений:

```
A
(A'+'B)'*'(C'-'D)
Begin (Ho-ho-ho '(' ('A joke')) End
()
(()'100'100() (()) ) [[
```

Примеры последовательностей, не являющихся (объектными) выражениями:

```
) End
A ( B)((C)
( A ')';
```

ЗАМЕЧАНИЕ: Заключение в кавычки каждого знака, как в одном из рассматриваемых выше примеров:

```
(A'+'B)'*'(C'-'D)
```

может представлять неудобство. Следует, однако, иметь в виду, что подобное заключение необходимо только в тексте РЕФАЛ-программы, где появление больших выражений такого рода маловероятно. Такое выражение типично для данных. Когда оно набирается либо читается из файла как данные, при использовании специальной функции **`Input`**, кавычками окружаются только скобочные символы для того, чтобы отличать их от структурных скобок. Мы могли бы, таким образом, набрать:

```
(A + B) * (C - D)
```

В алгебре мы используем выражения для представления некоторой последовательности операций над числами и переменными; скобки в выражении отмечают порядок выполнения операций. Если выражение, приведённое выше, понимать как алгебраическое, то оно может быть представлено в виде дерева, показанного на *Рис. 1.1*.

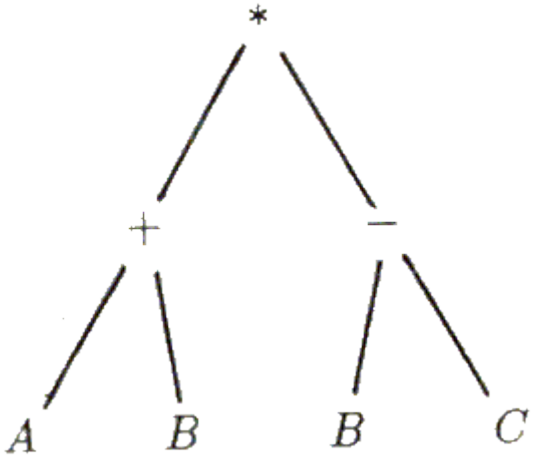


Рис. 1.1 Дерево для алгебраического выражения

Следует подчеркнуть, что концепция выражения в РЕФАЛе является более общей. Мы не предполагаем никакой специальной интерпретации выражений; они могут быть использованы различным образом. РЕФАЛ-выражение просто является структурированным объектом, который сконструирован некоторым способом из символов и структурных скобок. Для каждой структурной скобки в нем имеется в точности одна парная скобка противоположного вида. Вместе они образуют нечто вроде коробки или кармана. Они выделяют такую подсистему во включающей системе, которая является частью целого, но в то же время сохраняет и собственную целостность. Если вы выделяете одну границу в такой подструктуре, то вторая граница определяется единственным образом. Отношение между системой и её подсистемами является очень важным аспектом окружающего мира. Когда мы создаём символьные модели мира, структурные скобки моделируют такое отношение. Если фортепиано (Piano), скрипка (Violin) и альт (Viola) находятся в квартире Энн (Ann apartment), а виолончель (Cello) и контрабас (Bass) - в квартире Боба (Bob apartment), то эта ситуация может быть промоделирована РЕФАЛ-выражением:

```
Ann-apt(Piano Violin Viola) Bob-apt(Cello Bass)
```

РЕФАЛ-выражение может быть представлено деревом так же, как и алгебраическое выражение. Однако, если мы понимаем выражение именно как РЕФАЛ-выражение, без какой-либо интерпретации, дерево должно кое-чем отличаться. На *Рис. 1.1* мы располагали операции выше их аргументов. Но если мы не интерпретируем выражение как алгебраическое, то **A + B** является просто конкатенацией трёх символов, и все они должны располагаться на одном и том же уровне. Дерево должно выглядеть, как это показано на *Рис. 1.2*. Листьями (конечными узлами) дерева являются символы; все другие узлы - это заключённые в скобки подвыражения.

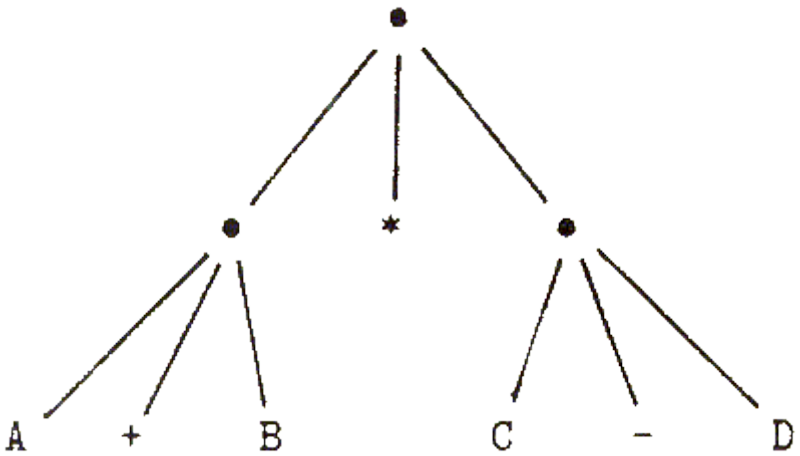


Рис. 1.2 Дерево РЕФАЛ-выражения

Другим способом двумерного графического представления выражений является соединение парных скобок линиями, как бы связывание их верёвочками; иерархия подвыражений при этом становится более наглядной (см. *Рис. 1.3*). Когда РЕФАЛ-выражения представлены в компьютере, адрес пары скобок хранится вместе с каждым членом пары, так что становится возможным совершать скачок от каждой скобки к её дополняющей за один шаг, как бы пробегая вдоль линий рисунка.

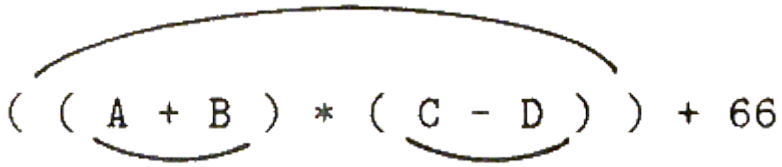


Рис. 1.3 Парные скобки соединены для большей наглядности.

Упражнение 1.1

Записать строку **`Joe's Pizza is "cute"`**, используя одинарные и двойные кавычки в качестве ограничителей.

Упражнение 1.2

Записать -2^{36} как целое число в РЕФАЛе.

1.3. СОПОСТАВЛЕНИЕ С ОБРАЗЦОМ

Выражения-образцы (или просто *образцы*) РЕФАЛа отличаются от объектных выражений тем, что они могут включать *свободные переменные*. Вот некоторые примеры свободных переменных:

`e.1  e.Data1  e.data1  s.X  s.Free-var  t.1  t.25`

Свободные переменные состоят из указателя типа, точки и индекса. Индексом переменной является идентификатор либо целое число. Если индекс является однобуквенным идентификатором или одноразрядным числом, точка может быть опущена. Идентификатор, используемый как индекс после точки, может начинаться как со строчной буквы, так и с прописной. Есть три указателя типа. Они обозначаются малыми буквами **`s`**, **`t`** и **`e`**; соответствующие им переменные называются s-, t- и e-переменными. Различие между ними состоит в множествах их допустимых значений. Значением s-переменной может служит только один символ. Значением t-переменной может быть любой терм (напомним, что терм - это либо символ, либо выражение в структурных скобках). Значением e-переменной может быть любое выражение.

Образец можно рассматривать как множество объектных выражений, которые могут быть получены в результате придания допустимых значений всем переменным. Так, образец **`A e.1`** представляет множество всех объектных выражений, начинающихся с символа **`A`**. Всем вхождениям одной и той же переменной следует придавать одно и то же значение. Образец **`s.1 e.2 s.1`** можно рассматривать как множество всех объектных выражений, которые начинаются и заканчиваются одним и тем же символом и состоят по крайней мере из двух символов.

Но выражения-образцы выполняют также ещё одну функцию: они описывают декомпозицию объектных выражений, в ходе которой переменные образца отождествляются с некоторыми частями объектного выражения, то есть принимают некоторые значения. С этой точки зрения, **`A e.1`** является процедурой, которая проверяет, что первым символом данного объектного выражения является **`A`**, и присваивает остальную его часть переменной **`e.1`** в качестве значения. Эта процедура известна как

сопоставление с образцом.

Для заданных объектного выражения ***E*** и образца ***P*** будем обозначать операцию сопоставления ***E*** с ***P*** через ***E*** : ***P*** , где двоеточие - ещё один специальный знак РЕФАЛа, знак сопоставления. Будем говорить также, что ***E*** *распознаётся* как (частный случай) ***P***. В паре ***E*** : ***P*** назовём ***E*** *аргументом*, а ***P*** - *образцом* операции сопоставления.

Если существует подстановка ***S*** для переменных в ***P*** такая, что применение ***S*** к ***P*** приводит к ***E***, тогда будем говорить, что сопоставление, или распознавание, является успешным. Переменные в ***P*** принимают значения, предписываемые подстановкой ***S***. В противном случае сопоставление считаем неудавшимся. Если имеется несколько способов присваивания значений свободным переменным в образце, при которых сопоставление может быть успешно, то тогда выбирается такой способ, при котором самая левая e-переменная принимает наиболее короткое значение. Если это не приводит к однозначности, подобный выбор делается для следующей (самой левой) e-переменной, и т.д. Иными словами, при выполнении распознавания аргумент ***E*** обозревается слева направо и выбирается первое успешное сопоставление ***E*** с ***P***.

Приведём несколько примеров. Сопоставление

```
A B C : A e.1
```

считается успешным , если **e.1** принимает значение **B C** . Сопоставление

```
('ABC') '++' : s.X e.1
```

терпит неудачу, поскольку левая скобка не символ, так что она не может быть сопоставлена с **s.X**. С другой стороны, её нельзя игнорировать. Однако,

```
('ABC') '++' : (s.X e.1) e.Out
```

становится успешным, когда **s.X** присваивается **'A'**, **e.1** присваивается **'BC'**, а **e.Out** присваивается **'++'**.

Распознавание **'++'** как **s.1 e.2 s.1** является успешным — при **s.1**, принимающем значение **'+'**, и при присвоении **e.2** пустого выражения. Но сопоставление

```
'+' : s.1 e.2 s.1
```

терпит неудачу.

Следующее сопоставление:

```
A B '+' C '+' D E F  :  e.1 '+' e.2
```

является примером ситуации, когда имеется более чем один способ придания значений для сопоставления. Согласно приведённому выше определению, символ **'+'** в образце будет идентифицироваться с первым символом **'+'** в аргументе (сопоставление слева-направо). Таким образом, **e.1** примет значение **A B**, а **e.2** — станет цепочкой **C '+' D E F**.

В следующем примере:

```
A B '-' (C '+' D E F)  :  e.1 '+' e.2
```

сопоставление потерпит неудачу, так как единственный символ **'+'** в аргументе находится внутри скобок. При сопоставлении его с **'+'** в образце мы должны были бы присвоить **e.1** и **e.2** такие

значения, которые несбалансированы по скобкам, а это невозможно. Структурные скобки в РЕФАЛе — вещь весьма серьёзная. Части выражений, которые сами не являются выражениями, не могут обрабатываться самостоятельно ни в каком контексте.

Упражнение 1.3

Запишите образцы, которые могут быть заданы следующими словесными описаниями: (a) выражение, оканчивающееся двумя одинаковыми символами; (b) выражение, которое содержит по крайней мере два одинаковых терма на верхнем уровне структуры; (c) непустое выражение.

Упражнение 1.4

Определите результаты следующих сопоставлений:

```
(a) 'abbab' : e.1 t.X t.X e.2
(b) 'ab(b)ab' : e.1 t.X t.X e.2
(c) A(B) C D (A (B)) : e.6 e.4(e.6)
(d) '16 0' : 16 e.Z
```

1.4. ПРЕДЛОЖЕНИЯ И ПРОГРАММЫ

РЕФАЛ-программа есть перечень функциональных определений. Она может также содержать *внешние ссылки* (которые будут обсуждаться в Главе 2) и *комментирующие строки*. Комментирующая строка начинается с символа звёздочки *****, и не влияет на выполнение программы. Комментирующие строки могут вставляться как внутрь функциональных определений, так и между ними. Порядок функциональных определений в программе не имеет значения.

Функциональное определение имеет одну из следующих форм:

имя-функции { *блок* }

$ENTRY *имя-функции* { *блок* }

Значимость префикса **$ENTRY** будет рассмотрена в Главе 2 наряду с применением внешних ссылок.

Имя функции является идентификатором. *Блок* есть перечень предложений, разделяемых точками с запятой; последнее предложение также может заканчиваться точкой с запятой. Порядок предложений в функциональном определении является существенным.

Предложение имеет вид:

левая-часть **=** *правая-часть*

где левая часть является выражением-образцом, а правая часть является общим *выражением*. Напомним, что объектное выражение может включать только символы и структурные скобки, а выражение-образец может, кроме этого, включать и свободные переменные. Общее же выражение, в дополнение к составляющим выражения-образца, может включать ещё *вычислительные скобки* < и >. Они входят в него как части вызовов функций:

<*имя-функции аргумент*>

где *имя-функции* является либо идентификатором, либо одним из четырёх знаков арифметических операций (**+**, **-**, *****, и **/**); а *аргумент* является общим РЕФАЛ-выражением. Таким образом, вычислительные скобки могут быть вложенными. Заметим, что имя функции должно следовать непосредственно за открывающей вычислительной скобкой, оно не допускает включения пробелов или

конца строки.

Имеются следующие важные ограничения на правую часть предложения: правая часть может включать только такие свободные переменные, которые входят также и в левую часть (см. следующий раздел). Имеется также техническое ограничение: индексы переменных должны быть уникальными, т.е. можно использовать **e.X** и **s.1** любое число раз, но если уже используется **e.X**, то запрещено использовать **s.X** или **t.X** в этом же предложении; использование **s.1** исключает применение e- и t-переменных с индексом **1**.

Наряду с комментирующими строками, предложения могут включать длинные комментарии, которые начинаются комбинацией **/*** и заканчиваются комбинацией ***/**. Они могут появляться всюду при условии, что ими не нарушаются лексические элементы (идентификаторы, числа, строки, переменные) и левая вычислительная скобка не отрывается от следующего за ней имени функции.

1.5. АБСТРАКТНАЯ РЕФАЛ-МАШИНА

РЕФАЛ-машиной называется абстрактное устройство, которое выполняет РЕФАЛ-программы. Она имеет два потенциально бесконечных хранилища информации: *поле программы* и *поле зрения*. Поле программы содержит РЕФАЛ-программу, которая загружается в машину перед запуском и не изменяется в ходе выполнения. Поле зрения хранит выражение без свободных переменных; такие выражения называются определёнными. Поле зрения (т.е. выражение в этом поле) изменяется в ходе работы машины.

Работа РЕФАЛ-машины осуществляется в пошаговом режиме. Каждый шаг выполняется следующим образом. Если выражение в поле зрения не включает вычислительных скобок (мы будем называть такие выражения пассивными), РЕФАЛ-машина приходит к нормальному останову. В противном случае она выбирает в поле зрения одно из подвыражений вида <***F E***>, где ***F*** является функциональным символом, а ***E*** — выражением, для того чтобы трансформировать его, используя определение ***F*** из программы. Это подвыражение называется *первичным активным подвыражением*. Оно определяется как первое (слева) подвыражение <***F E***>, такое, что ***E*** пассивно, т.е. не содержит вычислительных скобок.

Первичное подвыражение <***F E***> трансформируется следующим образом. РЕФАЛ-машина сравнивает ***E*** с последовательными предложениями из определения ***F***, начиная с первого, осуществляя поиск первого *применимого* предложения.

Предложение применимо, если ***E*** может быть распознано как его левая часть ***L***, т.е. сопоставление ***E : L*** является успешным. После нахождения первого применимого предложения РЕФАЛ-машина копирует его правую часть ***R*** и применяет к ней подстановку, полученную при сопоставлении ***E : L***. Таким образом, свободные переменные в ***R*** замещаются значениями, которые они должны принять для успешного сопоставления. Затем определённое выражение, сформированное при этом, замещает первичное активное подвыражение <***F E***> в поле зрения. Этим завершается текущий шаг и машина переходит к выполнению следующего шага. Если же применимых предложений в определении ***F*** не имеется, РЕФАЛ-машина приходит к аварийному останову *'Отождествление невозможно'*.

Для того, чтобы вычислить значение некоторой функции ***F*** с аргументом ***E*** (который должен быть объектным выражением), <***F E***> помещается в поле зрения РЕФАЛ-машины и машина запускается. Если после конечного числа шагов РЕФАЛ-машина приходит к нормальному останову, то содержимое поля зрения (также объектное выражение) является значением функции. Если же машина зацикливается или приходит к аварийному останову, значение функции считается неопределённым.

Порядок вычисления вложенных вызовов функций, используемый в РЕФАЛ-машине, называется

аппликативным или порядком типа *изнутри-наружу*. Перед запуском вычисления любого вызова функции должны быть завершены все вычисления вызовов функций в аргументе.

Кроме функций, определяемых в РЕФАЛе посредством предложений, существует несколько функций, которые не должны (а во многих случаях и не могут) определяться в РЕФАЛе, но всё-таки могут применяться, поскольку они *встроены* в систему. К этой категории, в числе прочих, относятся функции ввода-вывода и функции выполнения арифметических операций. Реализация РЕФАЛ-машины на компьютере отличается от абстрактной РЕФАЛ-машины тем, что перед просмотром определения функции в поле программы система проверяет, входит ли имя функции в список имеющихся встроенных функций. Если оно обнаружено в списке, активизируется соответствующая подпрограмма, которая выполняет требуемые операции и замещает вызов функции в поле зрения на его вычисленное значение.

1.6. ДАЛЬНЕЙШИЕ ПРИМЕРЫ ФУНКЦИЙ

Пример 1. Замена плюсов на минусы

Допустим, желательно иметь функцию, которая обрабатывает символьные строки и конвертирует каждый знак **'+'** в **'-'**. Сперва определим алгоритм словесно:

1. Взять первый символ аргумента. Если это **'+'**, заменить его на **'-'**. Теперь применить трансформацию к оставшейся части аргумента так, чтобы результат был дописан к символу **'-'**.
2. Если первый символ не **'+'**, оставить его без изменения и применить трансформацию к оставшейся части так же, как и в пункте 1.
3. Если строка пустая, выходная строка также пуста. Конец работы.

Теперь выразим это на РЕФАЛе. Пусть именем функции служит **Chpm** :

```
* Chpm, Change Plus to Minus
Chpm {
*1.Symbol '+' encountered.Replace by '-':
  '+' e.1 = '-' <Chpm e.1>;
*2.Any symbol distinct from '+':
  s.2 e.1 = s.2 <Chpm e.1>;
*3.Empty string:
   =  ;
     }
```

Далее приведена последовательность вычислений для случая, когда аргументом **Chpm** является **'ab+c-+d'** :

```
<Chpm 'ab+c-+d'>
'a'<Chpm 'b+c-+d'>
'ab'<Chpm '+c-+d'>
'ab-'<Chpm 'c-+d'>
'ab-c'<Chpm '-+d'>
'ab-c-'<Chpm '+d'>
'ab-c--'<Chpm 'd'>
'ab-c--d'<Chpm >
'ab-c--d';
```

Имеется лучшее решение проблемы. За один шаг РЕФАЛ-машины можно найти первый символ **'+'** и заменить его на **'-'**. Если в строке нет ни одного символа **'+'**, она трансформируется в самое себя:

```
Chpm {
  e.1 '+' e.2 = e.1 '-' <Chpm e.2>;
  e.1 = e.1; }
```

При том же аргументе, что и выше, вычисление производится всего за три шага:

```
<Chpm 'ab+c-+d'>
'ab-'<Chpm 'c-+d'>
'ab-c--'<Chpm 'd'>
'ab-c--d'
```

Пример 2. Факториал

Определим в РЕФАЛе функцию факториал для целых неотрицательных чисел. Арифметические функции в РЕФАЛе (см. Главу 2 и Раздел справочника С) имеют имена **Add**, **Sub**, **Mul**, **Div**. Они могут быть также представлены символами **+**, **-**, *****и **/** соответственно.

```
Fact { 0 = 1;
       s.N = <* s.N <Fact <- s.N 1>>>; }
```

Приведём вычисление **<Fact 3>**:

```
<Fact 3>
<* 3 <Fact <- 3 1>>>
<* 3 <Fact 2>>
<* 3 <* 2 <Fact <- 2 1>>>>
<* 3 <* 2     <Fact 1>>>
<* 3 <* 2     <* 1 <Fact <- 1 1>>>>>
<* 3 <* 2     <* 1 <Fact 0>>>>
<* 3 <* 2     <* 1 1>>>
<* 3 <* 2     1>>
<* 3 2>
6
```

Пример 3. Перевод слова

Иногда функции определяются с помощью таблицы. Предположим, мы хотим транслировать итальянские слова в английские. Мы, разумеется, должны различать определение таблицы и определение функции, которая осуществляет перевод с использованием таблицы. Пусть первая именуется **Table** — это функция, не имеющая переменных, которая возвращает таблицу в качестве значения. Пусть вторая называется **Italian**, это функция одной переменной для перевода итальянских слов. Сразу же можно заметить, что функция, которая действительно использует таблицу, должна иметь её в качестве одного из своих аргументов. Поэтому введём функцию **Trans** от двух аргументов, которую будем вызывать в формате:

```
<Trans (e.Word) e.Table>
```

Скобки отделяют первый аргумент — слово, которое нужно перевести, от второго аргумента — таблицы, которая при этом используется. Теперь можно определить **Ital-Engl** через вызов **Trans**:

```
* Translate an Italian word into English
Ital-Engl { e.W = <Trans (e.W) <Table>>; }
```

Формат таблицы должен быть таким, чтобы необходимое слово могло быть найдено за счёт применения

простого и эффективного образца. Таблица для пяти итальянских слов может иметь вид:

```
Table { = (('cane') 'dog')
          (('gatto') 'cat')
          (('cavallo') 'horse')
          (('rana') 'frog')
          (('porco') 'pig')
      }
```

При таблице этого формата определение функции **Trans** имеет вид:

```
* Translate a word by table
Trans {
  (e.Word) e.1 ((e.Word) e.Trans) e.2 = e.Trans;
  (e.Word) e.1  =  '***';  }
```

Второе предложение говорит, что если слово не найдено в таблице, то в перевод подставляются три звёздочки.

Упражнение 1.5

В рекурсивной арифметике натуральные числа представляются как 0,0',0", и т.д. Операция сложения определяется соотношениями:

```
x + 0 = x
x + y' = (x + y)'
```

Написать наиболее близкий к этому вариант на РЕФАЛе, используя **'0'** в качестве **0** и **'s'** вместо штрих-символа **'**.

Упражнение 1.6

Определить функцию **<Addb (e.1)(e.2)>**, которая складывает двоичные числа **e.1** и **e.2**.

2. РЕФАЛ В КОМПЬЮТЕРАХ

2.1. КАК ЗАПУСТИТЬ ЕГО

Ранее были приведены определения только отдельных функций в РЕФАЛе. Теперь желательно записать простую программу и запустить её на IBM микрокомпьютере (XT или AT) под управлением MS-DOS. Запуск её в других операционных окружениях аналогичен.

Функция перевода итальянских слов в английские была определена в *Разд. 1.6*. Допустим, требуется переводить и распечатывать итальянские слова, набираемые на терминале. Пусть слова в строке разделяются любым количеством пробелов. Нужно, чтобы наша программа читала строку, переводила каждое слово и выдавала на принтер строку перевода. Затем программа должна запрашивать следующую строку. Если итальянское слово не может быть найдено в таблице, оно должно замещаться комбинацией *******.

Потребуются некоторые средства для обмена информацией с РЕФАЛ-машиной. Некоторые основные функции ввода-вывода должны обеспечиваться системой, так как они очевидно не могут быть определены посредством РЕФАЛ-предложений. Функции, которые не определены в РЕФАЛе, но всё-таки могут применяться в системе, называются *встроенными функциями*. Перечень встроенных функций РЕФАЛ-5, вместе с кратким описанием их действия, можно найти в Reference Section C. Читателю рекомендуется ознакомиться с Reference Section C перед написанием конкретных программ на РЕФАЛе для того, чтобы знать, какие встроенные функции имеются в его распоряжении.

Для нашей задачи имеются две необходимые встроенные функции: **Card** и **Prout**. Во время исполнения **<Card>** РЕФАЛ-машина ожидает то, что пользователь набирает в строке*, завершая символом возврата каретки. Затем она читает строку и замещает вызов функции результатом, который трактуется как строка символов. Если пользователь вводит специальный символ Конец-Файла, которым в MS-DOS является комбинация Control-Z, **<Card>** будет замещена числом 0. Это является сигналом для РЕФАЛ-программы, что не осталось строк, подлежащих считыванию. Напомним, что знаки клавиатуры, слова и числа являются различными символами РЕФАЛа. Так как **Card** обрабатывает строку как последовательность символов, число 0 невозможно спутать с содержимым любой строки.

Поле того, как **<Prout *E*>**, где **E** является некоторым выражением, вычислено, машина выдаёт на принтер **E** и стирает вызов. Имеется также и другая функция вывода, **Print**, которая не уничтожает свой аргумент.

Начнём программирование сверху вниз; т.е., сперва определим общую структуру программы, а затем будем разрабатывать детально. Наша программа транслирует входной текст построчно, поэтому потребуется функция — назовём её **Trans-line** — такая, что если **E** является входной строкой, то вычисление:

<Trans-line E>

порождает её перевод. Далее, необходима контролирующая функция, которая вызывает <**Card**>, проверяет, совпадает ли результат с 0, и либо вызывает **Trans-line**, либо объявляет окончание работы. Назовём эту функцию верхнего уровня **Job**. Поскольку **Job** предназначена для анализа результата **Card** , она должна вызываться следующим образом:

** Имя функции оставлено со времён, когда входная информация считывалась в большинстве случаев с перфокарт*

```
<Job <Card>>
```

Теперь попытаемся определить **Job** двумя предложениями, соответствующими двум разновидностям ее аргумента:

```
Job { 0 = ;
      e.X = <Trans-line e.X>; }
```

Если результатом **<Card>** является 0, поле зрения становится пустым, т.е. пассивным, и это означает конец работы. В противном случае **Trans-line** осуществит перевод строки и поместит её в поле зрения. Это не совсем то, что требуется. Во-первых, желательно, чтобы результат выдавался на принтер, а не просто оставался в поле зрения. Во-вторых, следует сделать запрос на следующую вводимую строку, а это означает, что **<Job <Card>>** должна попасть в поле зрения снова. Напомним, что параллельные (т.е., не вложенные друг в друга) активные подвыражения активизируются слева направо. Поэтому, если нужно, чтобы несколько действий были произведены последовательно, мы объединяем соответствующие подвыражения в требуемом порядке. Приходим к следующему определению:

```
Job { 0 = ;
      e.X = <Prout <Trans-line e.X>>
      <Job <Card>>; }
```

Теперь определим **Trans-line**. Рассмотрим, какие возможные случаи аргумента следует различать. Прежде всего, он может быть пробелом. Следует его проигнорировать и применить **Trans-line** к остатку строки:

```
Trans-line ' 'e.X  =  <Trans-line e.X>;
```

Если перед словами имеется по нескольку пробелов, это предложение исключит их все, шаг за шагом. Если первый набираемый символ не является пробелом, он должен быть буквой, с которой начинается слово. Так как слова разделяются пробелами, мы можем выделить первое слово, используя образец:

```
e.Word ' ' e.Rest
```

Согласно правилам сопоставления слева-направо, пробел в этом образце будет сопоставлен с первым пробелом в аргументе (если таковые имеются). Поэтому, **e.Word** становится первым словом. Оно переводится с использованием функции **Trans**, определённой в *Разд. 1.6*. Английские слова, продуцируемые **Trans**, должны отделяться пробелом от слов, которые будут появляться за счёт рекурсивного вызова **Trans-line**. Таким образом, мы имеем предложение:

```
e.Word' 'e.Rest =
          <Trans (e.Word)<Table>>' '
          <Trans-line e.Rest>;
```

Имеются ещё две возможности. Одна из них, когда строка полностью использована (т.е. пуста), в таком случае вычисление **Trans-line** завершено. Другая - когда вся оставшаяся строка является одним словом. Полное определение выглядит следующим образом:

```
Trans-line {
  ' 'e.X = <Trans-line e.X>;
  e.Word' 'e.Rest =
             <Trans (e.Word)<Table>>' '
             <Trans-line e.Rest>;
     = ;
```

```
  e.Word = <Trans (e.Word)<Table> >' '; }
```

ЗАМЕЧАНИЕ: **`e.Word`** в последнем предложении не может быть пустым, что вытекает из предыдущего предложения.

Теперь определены все функции, необходимые для программы. Но как поместить начальное выражение в поле зрения РЕФАЛ-машины? РЕФАЛ-5 использует соглашение, согласно которому начальным состоянием поля зрения всегда является:

```
<Go>
```

Это значит, что среди функций, определённых в выполнимой программе, всегда должна быть стандартная функция **`Go`** без аргументов:

```
$ENTRY Go { =  the initial Refal expression  }
```

Префикс **`$ENTRY`** объявляет функцию **`Go`** как *входную функцию*, которая может быть вызвана из любого модуля программы (модульная структура программы описана в следующем разделе). В нашем случае начальным выражением должно быть **`<Job <Card>>`**.

ЗАМЕЧАНИЕ: Функция **`Go`** *не может вызывать* сама себя и *не может вызываться* никакой другой функцией. Она служит только для инициации поля зрения.

Теперь соберём все функции в программу и добавим некоторые комментарии. Будем придерживаться правила, согласно которому все строки-комментарии помещаются перед текстом, а прочие комментарии — после текста, к которому они относятся:

```
* Translation from Italian into English
$ENTRY Go { = <Job <Card>>  }

* The top-level control function.
Job {
  0 = ;                   /* end of file */
  e.X = <Prout <Trans-line e.X>>
       <Job <Card>>; }

* Translate one line
Trans-line {
  ' 'e.X = <Trans-line e.X>;
  e.Word' 'e.Rest =
              <Trans (e.Word) <Table>>' '
              <Trans-line e.Rest>;
   = ;
  e.Word = <Trans (e.Word) <Table>>' '; }

* Italian-English dictionary table
Table { = (('cane') 'dog')
          (('gatto') 'cat')
          (('cavallo') 'horse')
          (('rana') 'frog')
          (('porco') 'pig')
        }
* Translate Italian word into English by table
Trans {
  (e.It)e1((e.It)e.Eng)e2 = e.Eng;
```

```
    (e.It)e1 = '***'; /* word not in table */
       }
```

Для того, чтобы запустить эту программу, наберите её в любом текстовом редакторе и сохраните как файл с расширением **.ref** (например, **prog1.ref**). Теперь введите команду:

```
refc prog1
```

Она вызывает РЕФАЛ-компилятор, который переводит программу в промежуточный язык RASL (о котором пользователю не нужно знать ничего, кроме того, что он существует) и размещает её под именем **prog1.rsl**. Если компилятор обнаруживает синтаксические ошибки в программе, он выдаёт соответствующее сообщение и создаёт файл под именем **prog1.lis**, который содержит листинг программы с индикацией обнаруженных ошибок. Если синтаксических ошибок нет, листинг не порождается.

Для запуска программы вызовите РЕФАЛ-машину, вводя:

```
refgo prog1
```

Это приводит к загрузке RASL файла **prog1.rsl** с диска в оперативную память. Затем начинается интерпретация программы, т.е. эмуляция абстрактной РЕФАЛ-машины. Когда **<Card>** становится первым ведущим выражением в поле зрения, РЕФАЛ-машина останавливается и ожидает ввода строки с терминала. Введите:

```
rana cane     rana vacca   porco
```

Система выдаст на принтер

```
frog dog frog *** pig
```

Введите:

```
cavallo
```

и последует ответ:

```
horse
```

Введите символ Конец-Файла (Control-Z), и программа завершит работу.

Когда вы тестируете и отлаживаете программу, весьма полезен РЕФАЛ-трассировщик. Чтобы применить его, введите **reftr** вместо **refgo**:

```
reftr prog1
```

Трассировщик распознает множество команд, управляющих выполнением программы. Можно выдавать на принтер текущее содержимое поле зрения либо только ведущее активное подвыражение. Можно проделать некоторое количество шагов и затем снова распечатать поле зрения либо активное подвыражение. Возможно также запустить машину до момента, когда она завершит вычисление текущего активного подвыражения, а затем распечатать полученное объектное выражение. Лучшим способом управления процессом вычислений является определение ряда образцов — так называемых *контрольных точек*. Тогда РЕФАЛ-машина будет останавливаться и запрашивать дальнейшие команды всякий раз, когда активное подвыражение сопоставлено с одним из образцов.

См. перечень команд трассировщика в Reference Section D .

Использование функции **Go** является основным путём инициализации РЕФАЛ-машины. Есть и другой способ — через сервис РЕФАЛ-программ с очень коротким именем **E** (от **Evaluator**), который поставляется на системной дискете. Эта программа позволяет вводить активные РЕФАЛ-выражения в поле зрения и вычислять их без вызова **refgo** или **reftr** каждый раз. Она имеет две полезных черты. Во-первых, произвольное выражение может быть вычислено, не только стандартное **<Go>**. Во-вторых, всякий раз при вызове **refgo** или **reftr**, программа загружается с диска в оперативную память. Используя **E**, мы делаем это лишь один раз, при этом все последовательные вычисления (в частности, вычисление **<Go>**, если мы хотим запускать программу повторно) выполняются без повторной загрузки программы. Вычислитель будет описан в *Разд. 6.3* (см. также Reference Section A). В этом месте просто приводится иллюстрация, как работает метод.

Добавим строку:

```
$ENTRY Upd {e.X = <Mu e.X>;}
```

в программу **prog1**. Затем введём:

```
refgo e+prog1
```

На терминале появится следующее сообщение:

```
Type expression to evaluate. To end: empty line.
To end session: empty expression
```

Введём:

```
<Table>
```

и нажмём клавишу Return дважды для ввода пустой строки. Система выдаёт на печать:

```
The result is:

  (('cane') 'dog')
  (('gatto') 'cat')
  (('cavallo') 'horse')
  (('rana') 'frog')
  (('porco') 'pig')
```

и повторяет приглашение ввести другое выражение. РЕФАЛ-выражение, которое вы представите, может включать вызовы любых функций, определённых в **prog1**. Оно может также включать вызовы стандартных встроенных функций. Допустим, вы набрали:

```
<+ 2 <* 3 <* 4 5>>>
```

Система выдаст результат **62** .

Может быть вызвана любая комбинация функций. В частности, вы можете вызвать **<Go>**, которое рассматривается как стартовая функция **prog1**. Эффект будет таким же, как если бы вы просто выполнили **prog1**, за исключением следующего:

1. Окончательное значение **Go** будет выдано на печать (в то время как оно теряется при прямом выполнении).
2. После окончания вычисления система пригласит вас представить следующее выражение для вычисления. Вы можете вычислить **Go** снова, без перезагрузки программы с диска.

Упражнение 2.1

Модифицировать **prog1** так, чтобы система выдавала **Please enter a line** перед переходом в ожидание ввода, и **The translation is:** перед выдачей на печать перевода.

2.2. ПРОГРАММНЫЕ МОДУЛИ

Функции, которые однажды уже были определены, не нужно определять повторно. Не требуется и повторная их компиляция. При размещении функциональных определений на языке РЕФАЛ в файлах с расширением **.ref**, их переводы в промежуточный язык RASL размещаются в соответствующих файлах с расширением **rsl**. При написании большой программы мы часто разбиваем её на ряд частей, называемых *модулями*, тестируем и отлаживаем их отдельно, а затем объединяем их в единую программу.

Некоторые функции в модуле предназначены для вызова из функциями из других модулей. Другие функции являются вспомогательными; пользователю совсем не обязательно знать об их существовании. Это могло бы породить конфликт имён. В самом деле, допустим, что два программиста написали свой модуль и использовали одно и то же имя для одной из вспомогательных функций. Как тогда РЕФАЛ-машина распознает, какое из определений использовать?

Эта проблема решена путём определения некоторых функций в каждом модуле в качестве *входных функций*. Только входные функции могут вызываться из других модулей. Чтобы определить функцию как входную, просто добавляется ключевое слово **$ENTRY** перед именем функции, с которого начинается определение. Пусть входная функция **F1** определена в модуле **Mod1**:

```
* Module Mod1
$ENTRY F1 { ... }
```

Для того, чтобы вызвать **F1** из другого модуля **Mod2**, эта функция должна быть объявлена в **Mod2** как *внешняя* функция:

```
* Module Mod2
$EXTRN F1;
     ... <F1 ...> ...
```

Внешнее утверждение отмечает для компьютера, что **F1** определена как входная точка в каком-то другом модуле, а не в текущем. Такой модуль (**Mod1** в нашем примере) должен быть включён в вызов **refgo** или **reftr** :

```
refc Mod2+Mod1
```

В одном внешнем утверждении может быть объявлено несколько внешних функций:

```
$EXTRN F1,F2,F3;
```

Имена внешних функций должны быть одними и теми же для всех модулей, которые используются совместно. Функции, которые не объявлены как входные, могут вызываться только из функций. определённых в том же модуле. Таким образом, разрешаются конфликты имён. Во время написания модуля можно изобретать любые имена для вспомогательных функций, не беспокоясь о функциональных именах в других модулях.

Среди модулей, используемых совместно, один должен включать определение стартовой функции **Go**; он будет считаться *главным* модулем. Если, в дополнение к главному модулю **Mainmod**, желательно

использовать некоторые другие модули: **Mod1**, **Mod2**, и т.д. до **Modn** , вводится:

```
refc Mainmod+Mod1+Mod2+ ... +Modn
```

либо:

```
reftr Mainmod+Mod1+Mod2+ ... +Modn
```

Рассмотрим типичную ситуацию. Функция **Go**, которая выполняет задание, определена в модуле **prog1**, а некоторые из необходимых функций определены в модуле под названием **functions**. Требуется также ряд постоянно используемых функций, которые собраны в модуле **reflib**. Если все три модуля уже имеются в RASL форме, то для запуска **prog1** вводится:

```
refgo prog1+functions+reflib
```

Помимо главного модуля, другие модули также могут включать определение функции **Go**. Оно будет игнорироваться до тех пор, пока модуль не займёт первое место в списке.

2.3. ВВОД-ВЫВОД

Стандартный ввод и вывод, выполняемый посредством **Card** и **Print**, может быть переориентирован на другие файлы с помощью средств, обеспечиваемых операционной системой. Но часто является желательным использование одного или более файлов ввода-вывода в дополнение в операциям с клавиатурой и экраном. Тогда могут быть использованы встроенные функции **Get** и **Put** в комбинации с функцией **Open**.

Файл может использоваться как в режиме чтения (когда вы вводите информацию из него), так и в режиме записи (когда вы выводите в него информацию). Один и тот же файл может быть использован поочерёдно в различных режимах. Для того, чтобы использовать файл, его следует вначале открыть посредством активизации:

```
<Open s.Mode s.Channel e.File-name>
```

где **s.Mode** может быть либо **'r'** для режима чтения, либо **'w'** для режима записи; **e.File-name** является строкой, которая представляет полное имя файла; **s.Channel** является целым числом, которое присваивается открываемому файлу. Каналы используются для того, чтобы сделать процесс написания программы независимым от имён. файлов, с которыми будет взаимодействовать программа. **s.Channel** не должно превышать 19.

После того, как в файл произведена запись, он должен быть *закрыт*; в противном случае будет невозможно его читать либо в текущей программе, либо после её выполнения (файл, который был открыт для записи, но не закрыт, будет показываться в директории как пустой). Однако РЕФАЛ-функций для закрытия файлов не существует; это проделывается автоматически в двух случаях:

1. Когда выполнение РЕФАЛ-программы завершено либо при нормальном останове, либо по команде выхода из трассировщика.
2. Когда канал, открытый для записи, переоткрывается для чтения (либо с этим же файлом, либо с другим).

Чтобы читать из файла, который открыт для чтения, активизируется:

```
<Get s.Channel>
```

Машина считывает одну строку символов из файла, связанного с **s.Channel**. Если она дошла до символа Конец-Файла, результатом будет число 0. **Get** полностью аналогична **Card**.

Чтобы записать строку в файл, открытый для записи, активизируется:

```
<Put s.Channel e.Expression>
```

Затем **e.Expression** будет возвращено в качестве значение и добавлено как строка к текущему содержимому файла, связанного с **s.Channel**. Таким образом, **Put** аналогична **Print**. Функция:

```
<Putout s.Channel e.Expression>
```

является аналогичной **Prout** (возвращает *empty*). Когда **Open** применяется для того, чтобы связать канал с файлом, **s.Channel** не должна быть равным 0. При **s.Channel**, равном 0, функции **Get**, **Put** и **Putout** используют в качестве файла терминал (т.е., становятся эквивалентны **Card**, **Print** и **Prout**); однако их нельзя переназначить из операционной системы).

Допустим, строится функция, которая оперирует с тремя файлами в дополнение к операциям с терминалом. Можно присвоить этим файлам только каналы 1, 2 и 3, не вводя для них специальных имён; в функциях **Get** и **Put** будут указываться только каналы. Когда функциональное определение готово, можно либо компилировать его как отдельный модуль, либо поместить его в библиотеку. От может быть использован теперь с различными главными модулями и с различными файлами. Все, что требуется, - это присваивание каналов файлам с помощью вызова функции **Open** из главного модуля. Это невозможно было бы сделать, если бы мы использовали имена файлов непосредственно в функциях ввода-вывода.

Встроенные функции ввода-вывода обрабатывают только одну строку при каждом вызове и то, что они читают или записывают, всегда должно быть простой строкой символов. Но РЕФАЛ-машина имеет дело с выражениями. Трансформация входных строк в объектные выражения, вообще говоря, оставляется на усмотрение программиста, который создаёт любое специальное приложение к РЕФАЛу. Это обоснованно, поскольку способ, которым пользователь желает представить выражения с помощью символьных строк, может зависеть от специфики приложения. Пользователь должен иметь свободу определения ввода и вывода наиболее удобным способом. Тем не менее, потребность трансформации входных строк в выражения всякий раз, когда мы пишем РЕФАЛ-программу, представляет определённое неудобство. Поэтому был определено некоторое множество стандартных функций ввода-вывода общего назначения, которое поставляется с РЕФАЛ-системой (файл **reflib.ref**, см. Раздел справочника А).

Чтобы ввести РЕФАЛ-выражение, применяется **Input**. Для этой функции было принято представление выражений в основном такое же, как при написании программы на РЕФАЛе. Единственным отличием является менее строгое использование кавычек, которые следует допустить, потому всегда читаются *объектные выражения*, а не РЕФАЛ-программы. Единственными нецифровыми набираемыми символами, которые не могут представлять сами себя, являются круглые скобки, пробелы и одинарные кавычки. Хотя необходима и некоторая осторожность для того, чтобы избегать путаницы между набираемым символом **'-'** как чёрточкой в идентификаторе и символами **'+'** и **'-'** как знаками действительного числа. Таким образом, кавычки и пробелы должны использоваться. Далее, требование, чтобы первая буква идентификатора была заглавной, ослаблено; она может быть также строчной.

Если желательно считывание более чем одного выражения из одного файла, они разделяются пустыми строками. Пустая строка (дополнительный возврат каретки) будет также завершать ввод выражения в терминала. Когда активизируется

```
<Input s.Channel>
```

РЕФАЛ-машина будет читать файл, связанный с **s.Channel** до тех пор, пока не встретит пустую строку либо конец файла. Когда этот вызов активизируется снова, она считывает следующее выражение, и т.д.

Функция **Input** может также быть использована просто вместе с именем файла, который нужно читать:

```
<Input e.File-name>
```

Автоматически будет найден свободный канал, ему будет присвоено имя файла **e.File-name** и последует вызов **Open**. Для чтения с терминала следует использовать **<Input 0>** или **<Input>**.

Input удобна для интерфейса с машиной пользователя, но она может оказаться не лучшим выбором для обмена информацией между оперативной памятью и внешней памятью (магнитными дисками). Нужен более простой код для эффективного информационного обмена. Для этой цели мы готовы пожертвовать удобством чтения файлов (которое достигается в **Input** путём свободного использования пробелов и переносов). Поэтому наша небольшая РЕФАЛ-библиотека **reflib** включает функции **Xxin** и **Xxout** (eXpression eXchange IN и OUT). Можно также обнаружить там функцию **Xxinr** (Xxin в РЕФАЛе). **Xxin** и **Xxinr** действуют одинаково; различие только в том, что первая написана преимущественно на языке С и встроена в систему, в то время как вторая написана на РЕФАЛе с целью формального определения. **Xxin** гораздо более быстрая, чем **Xxinr**.

Функции обмена выделяют символ **#** как исключающий для представления объектов, не являющихся символами. Код выглядит следующим образом:

```
in the view field              on the disk

  s.Identifier              '#' e.Char-string ' '
   s.Number>               '#' e.Digit-string ' '
       (                           '('
       )                           ')'
     '('                           '#('
     ')'                           '#)'
     '#'                           '##'
```

Функции **Xxin** и **Xxout** противоположны по действию. **Xxinr** и **Xxout** могут, как и **Input**, использоваться либо с именами каналов, либо с именами файлов:

```
<Xxinr s.Channel>
<Xxin e.File-name>
<Xxout s.Channel e.Expr>
<Xxout (e.File-name)e.Expr>
```

Эффективная функция **Xxin** может использоваться только с именами файлов:

```
<Xxin e.File-name>
```

Канал 0 соответствует операциям с терминалом. Переносы не разбивают числа и идентификаторы. **Xxout** разрезает текст на строки длины 75.

ЗАМЕЧАНИЕ: Функции **Input**, **Xxin** и **Xxout** используют собственную таблицу свободных и занятых каналов, которая обозначается номером 153443950 (смотри программы) и *ничего не делают* с

системной таблицей, которая показывает, какие каналы действительно задействованы через функцию **Open**. Так, одна из этих функций ввода-вывода может выбрать канал, как свободный, даже если тот уже задействован в вызове **Open**. Поэтому не рекомендуется использовать совместно имена каналов и имена файлов в функциях ввода-вывода в одной и той же программе, поскольку выбор каналов может вступить в конфликт с автоматическим выбором. который осуществляют эти функции. Если всё-таки желательно использовать оба метода, тогда с **Open** надо использовать каналы с малыми номерами: 1, 2, 3, и т.д., поскольку каналы, выбираемые автоматически, будут иметь номера 19, 18, 17, и т.д.

Ниже приведены несколько примеров корректного и некорректного использования функций ввода-вывода. Во всех примерах целью является чтение файла **xfile**, запись его обменного кода в файл **yfile**, затем повторное его чтение и распечатывание.

Простейшим способом является использование только имён. файлов:

```
* Test of I/O functions.Example 1.
* This works
$ENTRY Go { =
   <Xxout ('yfile') <Input 'xfile'>>
   <Prout <Xxin 'yfile'>>; }
$EXTRN Input,Xxin,Xxout;
```

В этой программе **Xxin** закрывает **yfile** автоматически. Ей известно, что этот файл был открыт для записи, потому что **Xxout** проделала это по имени файла, а не по имени канала.

Следующий пример демонстрирует некорректное использование каналов:

```
* Test of I/O functions. Example 2.
* This does NOT work.
$ENTRY Go { =
   <Open 'w' 19 'yfile'>
   <Xxout 19 <Input 'xfile'>>
   <Open 'r' 19 'yfile'>
   <Prout <Xxin 'yfile'>>; }
$EXTRN Input,Xxin,Xxout
```

Ошибка здесь заключается в явном использовании канала 19 вместе с **Input**. Эта функция ввода переоткрывает канал 19 для чтения. Когда **Xxout** начинает работу, канал 19 по умолчанию считается открытым. Заменим канал 19 каналом 1:

```
* Test of I/O functions. Example 3.
* This  works.
$ENTRY Go { =
   <Open 'w' 1 'yfile'>
   <Xxout 1 <Input 'xfile'>>
   <Open 'r' 1 'yfile'>
   <Prout <Xxin 'yfile'>>; }
$EXTRN Input,Xxin,Xxout
```

Тогда это корректная программа. Второе **Open** выражение необходимо для закрытия **yfile**, который был открыт первоначально для записи. Если бы это не было проделано, как в рассматриваемом ниже случае:

```
* Test of I/O functions. Example 4.
* This does NOT work.
```

```
$ENTRY Go { =
   <Open 'w' 1 'yfile'>
   <Xxout 1 <Input 'xfile'>>
   <Prout <Xxin 'yfile'>>; }
$EXTRN Input,Xxin,Xxout
```

то программа не стала бы работать. Когда выполняется **Xxin**, файл **yfile** оказывается пустым, так как он не был закрыт.

ЗАМЕЧАНИЕ: Функции **Print** и **Prout** первоначально предназначались для распечатывания строк набираемых символов. Если их аргументы включают другие РЕФАЛ-объекты, а именно идентификаторы, числа и скобки, эти функции будут выдавать их в легко читаемом представлении. Идентификатор распечатывается большими буквами и заканчивается пробелом; десятичные цифры в макроцифрах также заканчиваются пробелом; скобки выпечатываются как символы **'('** и **')'** . Таким образом, важно знать, включает ли аргумент на самом деле несимвольные объекты или он состоит из строк символов, которые не изменяют вида при распечатывании. Это может оказаться препятствием при отладке; поэтому трассировщик распечатывает все выражения в обменном коде, который обладает единственным обратным отображением (см. Раздел справочника D). Если имеются сомнения в том, что выдано посредством **Prout**, используйте трассировщик, чтобы выяснить это.

Упражнение 2.2

Пусть приведённые ниже символьные строки считываются с помощью **Input**. Записать результатные выражения в том виде, в котором они должны были бы появиться в программе, и в обменном коде.

```
(a) sum-1
(b) sum - 1
(c) sum -1
(d) (95+25)'()'
```

Упражнение 2.3

Функция **Input** работает гораздо медленнее, чем **Xxin**, но её формат более удобен, когда файл написан человеком. Если один и тот же файл нужно читать более одного раза, возможно, имеет смысл написать его для **Input**, а затем уже конвертировать его в файл для **Xxin**. Написать программу, осуществляющую подобное конвертирование. Она должна работать с любым файлом (использовать встроенную функцию **Arg**).

2.4. ПРЕДСТАВЛЕНИЕ РЕФАЛ-ВЫРАЖЕНИЙ

При вводе команды **refgo** или **reftr** запускается программа, которая является *эффективным интерпретатором* РЕФАЛа. Тот факт, что она является интерпретатором, означает, что компьютер будет в основном имитировать шаги РЕФАЛ-машины. Будут поддерживаться две области памяти: поле программы и поле зрения; они будут представлениями поля программы и поля зрения абстрактной РЕФАЛ-машины.

На каждом шаге компьютер будет находить первичное активное подвыражение в поле зрения и применять к нему одно из предложений из поля программы. Это будет приводить к замещению активного подвыражения другим подвыражением. Название "эффективный" для интерпретатора означает, что он будет использовать некоторые адреса памяти для того, чтобы прямо переходить в нужные пункты памяти, не проходя через промежуточные выражения, как это предлагается в

абстрактной РЕФАЛ-машине. Он должен также избегать бесполезного копирования выражений. Более точно, должны удовлетворяться следующие требования :

1. Если некоторый символ локализован в поле зрения, следует иметь возможность непосредственной локализации предыдущего и последующего символов либо скобок.
2. Если структура или активирующая скобка локализованы в поле зрения, следует иметь возможность локализации парной ей скобки одним скачком, без сканирования заключённого в скобках выражения.
3. Если количество вхождений переменной справа не превосходит количества её вхождений слева, то замена вхождений переменной в правой части на их значения должно производиться без реального копирования или сканирования значений. Иными словами, должна иметься возможность перемещать подвыражения в поле зрения без их просмотра или переписывания.
4. В начале каждого шага доступ к первичному активному подвыражению в поле зрения (т.е., к следующему вызову функции, подлежащей вычислению) должен производиться за один скачок, без сканирования поля зрения, как это подразумевается в определении РЕФАЛ-машины.
5. Группа предложений, определяющих данную функцию, должна локализовываться за один скачок, без просмотра всего содержимого поля программы.

Пользователю РЕФАЛа не нужно знать о реальном представлении выражений ничего, кроме факта, что эти пять требований удовлетворяются. Очевидным способом удовлетворить этим требованиям является применение двухсвязных списков, со скобками, хранящими ссылку на парную скобку, наряду со ссылками на следующую и предыдущую. Это на самом деле и реализовано в РЕФАЛ-системе. Для того, чтобы совершать скачок к следующему активному подвыражению, хранится стэк адресов активных подвыражений (стэк вызовов функций). РЕФАЛ-компилятор, под названием **`refc`** , транслирует РЕФАЛ-программу в промежуточный язык RASL и порождает файл, состоящий из подполей, каждое из которых хранит одно из функциональных определений. Доступ к этим подполям во время выполнения программы осуществляется через хэш-таблицу.

Элементы списка, представляющие символы РЕФАЛа, включают коды символьного типа. Таким образом система различает набираемые символы, идентификаторы, макроцифры и действительные числа. Для конвертирования одного типа в другой (например, цифровой строки **`'321'`** в макроцифру **321** или идентификатора **`Sum`** в строку **`'SUM'`** и т.д.), применяются соответствующие встроенные функции, описанные в Разделе справочника С.

Если это краткое описание внутреннего устройства РЕФАЛ-системы не особенно много говорит вам, не беспокойтесь: напоминаем ещё раз, на самом деле вам нет нужды знать о ней, если вы хотите только пользоваться языком.

Упражнение 2.4

Определить функцию **`<Merge s.1 s.2>`**, которая создаёт новый идентификатор, объединяя "тела" (строковые представления) идентификаторов **`s.1`** и **`s.2`**.

2.5. АЛГОРИТМ СОПОСТАВЛЕНИЯ С ОБРАЗЦОМ

Сопоставление с образцом является одной из двух основных операций в РЕФАЛ-машине, поэтому следует рассмотреть его более подробно. Напомним сперва некоторые основные определения.

Выражение, которое не содержит ни активизирующих скобок, ни свободных переменных, считается *объектным выражением*. Выражение, которое может содержать свободные переменные, но не включает активизирующих скобок, является *выражением-образцом*, или просто *образцом*. Операция

сопоставления записывается как **E : P**, где **P** является образцом, а **E** — объектным выражением. Здесь и далее большие буквы курсивом используются в качестве метасимволов для обозначения объектов языка РЕФАЛ.

Операция сопоставления либо проходит успешно, либо терпит неудачу. **E : P** проходит успешно, если есть подстановка ***S*** для свободных переменных в **P**, такая, что когда она выполнена, **P** становится тождественным **E**. Если отождествление успешно, говорим, что **E** может быть *распознано как* (частный случай) **P**, либо используем эквивалентное высказывание, что **P** можно отобразить на **E**. Тогда свободные переменные в **P** принимают значения, предписываемые подстановкой. Если сопоставление неудачно, значения переменных в **P** не определены. Может быть несколько подстановок, преобразующих **P** в **E**. Для того, чтобы обеспечить однозначный результат сопоставления, используется следующее соглашение о том, что сопоставление осуществляется слева направо:

Сопоставление слева-направо. Если существует более чем один способ присвоения значений свободным переменным в образце, при котором сопоставление достигается, то выбирается тот, в котором самая левая e-переменная принимает самое короткое значение. Если это не разрешает неоднозначности, тогда подобный выбор совершается для следующей `e`-переменной, стоящей слева, и т.д.

При таком соглашении сопоставление в РЕФАЛе становится однозначной операцией. Вот и все о результате сопоставления, что пользователь, строго говоря, должен предварительно знать при написании РЕФАЛ-предложений. Можно пропустить окончание этого раздела и всё-таки иметь возможность успешно программировать на РЕФАЛе. Однако он поможет мыслить о сопоставлении как о специфическом алгоритме. Поэтому будет определён точный алгоритм для сопоставления объектного выражения **E** с образцом **P**, как это реализовано в РЕФАЛ-5.

Вхождения символов, скобок и переменных будут называться *элементами* выражений. Пропуски между элементами будут называться *узлами*. Сопоставление **E : P** определяется как процесс *отображения*, или *проектирования*, элементов и узлов образца **P** на элементы и узлы объектного выражения **E**. Графическое представление успешного сопоставления приведено на *Рис. 2.1*. Здесь узлы представлены знаками **o**.

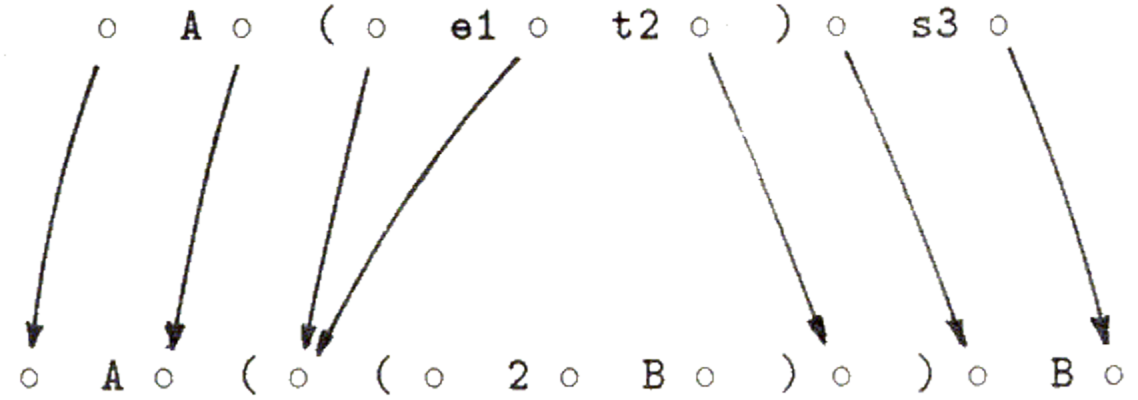


*Рис. 2.1 Сопоставление **E : P** является отображением **P** на **E**. Здесь объектным выражением **E** является **`'A'((2'B'))'B'`**, а образцом **P** является **`'A'(e.1 t.2)s.3`**.*

Следующие очевидные требования должны проверяться на каждой стадии сопоставления:

Общие требования к отображению P на E (сопоставлению E : P)

1. Если узел N_2 расположен в **P** правее узла N_1, то проекция N_2 в **E** может либо совпадать с проекцией N_1, либо располагаться справа от неё (линии проектирования не могут пересекаться).

2. Скобки и символы должны совпадать со своими проекциями.
3. Проекции переменных должны удовлетворять синтаксическим требованиям их значений; т.е., быть символами, термами или произвольными выражениями для `s-`, `t-` и `e-`переменных соответственно. Различные вхождения одной переменной должны иметь одинаковые проекции.

Предполагается, что в начале сопоставления граничные узлы **P** отображаются в граничные узлы **E**. Процесс отображения описывается при помощи следующих шести правил. На каждом шаге отображения правила 1-4 определяют следующий элемент, подлежащий отображению; таким образом, каждый элемент из **P** получает при отображении уникальный номер.

Правила отображения

1. После того, как отображена скобка, следующей подлежит отображению парная ей скобка.
2. Если в результате предыдущих шагов оба конца вхождения некоторой e-переменной уже отображены, но эта переменная ещё не имеет значения (ни одно другое её вхождение не было отображено), то эта переменная отображается следующей. Такие вхождения называются *закрытыми* e-переменными. Две закрытые e-переменные могут появиться одновременно; в этом случае та, что слева, отображается первой.
3. Вхождение переменной, которая уже получила значение, является *повторным*. Скобки, символы, s-переменные, t-переменные и повторные вхождения e-переменных в **P** являются *жёсткими* элементами. Если один из концов жёсткого элемента отображён, проекция второго конца определена однозначно. Если Правила 1 и 2 неприменимы, и имеется несколько жёстких элементов с одним спроектированным концом, то из них выбирается самый левый. Если возможно отобразить этот элемент, не вступая в противоречие с общими требованиями 1-3, приведёнными выше, тогда он отображается, и процесс продолжается дальше. В противном случае объявляется *тупиковая ситуация*.
4. Если Правила 1-3 неприменимы и имеются несколько e-переменных с отображённым левым концом, то выбирается самая левая из них. Она называется *открытой* e-переменной. Первоначально она получает пустое значение, т.е., её правый конец проектируется на тот же узел, что и левый. Другие значения могут присваиваться открытым переменным через удлинение (см. Правило 6).
5. Если все элементы **P** отображены, это значит, что процесс сопоставления успешно завершён.
6. В тупиковой ситуации процесс возвращается назад к последней открытой e-переменной (т.е., к той, что имеет максимальный номер проекции), и её значение *удлиняется*; т.е., проекция её правого конца в **E** подвигается на один терм вправо. После этого процесс возобновляется. Если переменную нельзя удлинить (из-за Общих требований 1-3), удлиняется предшествующая открытая переменная, и т.д. Если не имеется подлежащих удлинению открытых переменных, процесс сопоставления не удался.

На *Рис. 2.1* сопоставление производится следующим образом. Вначале имеется два жёстких элемента с одним отображённым концом: **`'A'`** и **`s.3`**. В соответствии с Правилом 3, отображается **`'A'`**, и этот элемент получает при отображении номер 1. Номера 2 и 3 будут назначены левой и правой скобкам, согласно Правилам 3 и 1. Внутри скобок начинается перемещение справа налево, так как **`t.2`** является жёстким элементом, который может быть отображён, в то время как значение **`e.1`** ещё не может быть определено. На следующем шаге обнаруживается, что **`e.1`** является *закрытой* переменной, чью проекцию не требуется обозревать для того, чтобы присвоить ей значение; что бы ни было между двумя узлами, это годится для присвоения (на самом деле, значение **`e.1`** оказывается пустым). Отображение **`s.3`** завершает сопоставление. Расположение отображающих номеров над элементами образца даёт наглядное представление описанного алгоритма :

```
 1   2   5    4    3   6
'A'  (   e.1  t.2  )   s.3
```

Рассмотрим другой образец, с отображающими номерами над элементами:

```
1   6   7    8    2   9    10   11   4   5    3
(   e.1 '+'  e.2  )   e.3  '+'  e.4  (   e.5  )
```

Здесь **e.1** и **e.3** являются *открытыми* переменными, в то время как **e.5**, **e.2** и **e.4** — *закрытые*. Скобки в РЕФАЛе используются для перепрыгивания из одной части выражения в другую. Первые четыре шага отображения разбивают выражение на три подвыражения. Последний из них переходит непосредственно к **e.5**. В первом подвыражении следует найти первый **'+'**. Начальное значение **e.1** становится пустым. Если следующий (т.е., первый) символ есть **'+'**, переменной **e.2** присваивается окончание подвыражения (т.е., все целиком), и осуществляется переход к номеру 9, который относится к переменной **e.3**. Если первый символ не является **'+'**, значение **e.1** удлиняется и становится равным этому символу; затем второй символ сравнивается с **'+'**, и т.д. Когда (и только когда) обнаружен номер 7 символа **'+'**, продолжается поиск номера 10 для **'+'**.

Если свободная переменная ***V*** из **P** принимает значение ***W***, это запишется как подстановка ***W*** ← ***V***. В отличие от общепринятых обозначений, переменная находится справа. Обоснование: она принадлежит правой части операции сопоставления **E** : **P**, а её значение является подвыражением в левой части.

Пусть указанный выше образец отображается на объектное выражение:

```
(Apples + Peaches + Plums) Cost $45 + 4% (Tax)
```

(Заметим, что это выражение записано без кавычек -- способ, которым его следует записывать для функции **Input**. Как упомянуто выше, это можно делать, когда мы оперируем с объектными выражениями. Если бы это была часть РЕФАЛ-программы, следовало бы заключать в кавычки все набираемые символы.) С использованием введённого обозначения присваиваний, результат сопоставления выглядит как:

```
            Tax <- e.5
         Apples <- e.1
Peaches + Plums <- e.2
       Cost $45 <- e.3
             4% <- e.4
```

Рассмотрим сопоставление:

```
('METASYSTEM INDEX')'XYZ' : (e.1 s.X e.2) e.3 s.X e.4
```

Номерами отображения являются следующие:

```
1    3    4    5    2    6    7    8
(    e.1  s.X  e.2  )    e.3  s.X  e.4
```

После того, как выделены два подвыражения, **e.1** становится пустым, а **s.X** проектируется на **'M'**. Затем **e.3** становится пустым, а **s.X** под номером 7 следует отобразить на начало второго подвыражения. Однако это невозможно, так как **s.X** должно было бы тогда принять значение **'X'**, в то время как оно уже приняло значение **'M'**. Таким образом объявляется тупиковая ситуация. Последней открытой переменной является **e.3**, поэтому она удлиняется и становится равной **'X'**. Но это не помогает, и удлинение **e.3** продолжается до тех пор, пока она не станет равной **'XYZ'** и не сможет

удлиняться далее. Затем удлиняется **e.1**, и цикл над **'XYZ'** повторяется. Окончательно приходим к успешному отображению с помощью подстановок:

```
      'METAS' <- e.1
          'Y' <- s.X
 'STEM_INDEX' <- e.2
          'X' <- e.3
          'Z' <- e.4
```

Следует заметить, что порядок, в котором отображаются жёсткие элементы образца, не влияют на окончательный результат сопоставления (ни на состояние успех/неудача, ни на значения переменных); уже установлен некоторый порядок (а именно, слева направо) , который полностью определяет только алгоритм сопоставления. Но порядок, в котором отображаются *открытые* переменные, имеет значение. В приведённом выше примере, если бы мы выбрали первым отображение **e.3**:

```
1    6    7    8    2    3    4    5
(    e.1  s.X  e.2  )    e.3  s.X  e.4
```

в результате получилось бы совершенно другое присваивание значений переменным:

```
                  <- e.3
              'X' <- s.X
             'YZ' <- e.4
'METASYSTEM_INDE' <- e.1
                  <- e.2
```

В образце:

```
e.1'+'e.2(('**')e.1 t.3)
```

на первый взгляд переменная **e.1** является открытой. Но это не так. Одно из вхождений **e1** является закрытым, а другое повторным:

```
9    10  11   2   3   5   6   4   8    7    1
e.1  '+' e.2  (   (  '*' '*'  )   e.1  t.3  )
```

Упражнение 2.5

Назначить отображающие номера и определить тип (закрытое, открытое, повторное) для каждого вхождения e-переменной в следующих образцах:

```
(a) e.A (t.2)(Sunday)(s.Day s.Night)
(b) (e.Word) e.1 ((e.Word) e.Translation) e.2
(c) (e.1'+'e.2'+'e.3)e.1 '+' e.2(e.3)
```

Упражнение 2.6

Определить результаты следующих сопоставлений:

```
(a) 'diffident' : e.A t.1 t.1 e.B
(b) 'diffident' : e.1 s.X e.2 s.X e.3
(c) '(Texas)' : (e.State)
(d) A B C D : e.1 e.2 e.3 D
(e) A(A B)(C)((C))D : e.1 e.X e.X e.2
```

Следует отметить, что на практике при программировании на РЕФАЛе редко используются очень сложные образцы.

3. ОСНОВНЫЕ ПРИЁМЫ ПРОГРАММИРОВАНИЯ

Средства языка, которые были введены до сих пор, составляют уровень, который известен как *базисный* РЕФАЛ. Расширение этого языка, которое предоставит некоторые дополнительные средства программирования, будет описано в следующей главе. Начнём с основных приёмов программирования на базисном РЕФАЛе.

3.1. КРУГЛЫЕ СКОБКИ КАК УКАЗАТЕЛИ

Чтобы проиллюстрировать процесс создания программы на РЕФАЛе, предположим, что мы хотим написать программу, которая удаляет все лишние пробелы в данной строке символов, т.е. замещает каждую группу подряд стоящих пробелов единственным пробелом. Назовём функцию, выполняющую эту работу, **Blanks**. Мы приходим к самому очевидному решению, рассуждая следующим образом: нам не нужны соседние пробелы; поэтому, как только обнаруживается пара соседних пробелов, следует удалить один из них и повторять это до тех пор, пока имеются такие пары. В результате получаем предложение:

```
<Blanks e1' ' e2>  =  <Blanks e1' ' e2>
```

(для удобочитаемости пробелы замещены знаками `' '`).

ЗАМЕЧАНИЕ: При рассмотрении отдельного РЕФАЛ-предложения удобно записывать его левую часть полностью, т.е. как вызов функции, который состоит из имени функции и угловых скобок вызова функции.

Если это предложение оказывается неприменимым, желаемый результат достигнут, и работа завершается при помощи предложения:

```
<Blanks e1> = e1
```

Таким образом, определение функции имеет вид:

```
Blanks {
  e1' ' e2 = <Blanks e1' ' e2>;
  e1 = e1; }
```

Проанализируем алгоритмическую эффективность нашей программы. Какие действия будет выполнять РЕФАЛ-интерпретатор, вычисляя эту функцию?

Переменная **e1** в первом предложении является *открытой*. Первоначально она принимает пустое значение, а затем удлиняется до тех пор, пока не будет найдена первая комбинация `' '` (если таковая имеется). Переменная **e2** является *закрытой*; поэтому оставшаяся часть аргумента не просматривается. Так как мы очевидно должны просматривать аргумент при поиске первых соседних пробелов, ни одного лишнего действия пока ещё не совершено.

Теперь наступает очередь замещения. Умный интерпретатор определит, сравнивая левую и правую части предложения, что для выполнения замещения нужно только исключить один из пробелов. Более ограниченный интерпретатор построит замещение из фрагментов поля зрения, на которые была спроектирована левая часть. В зависимости от степени интеллекта, интерпретатор может использовать те же пробел и скобки вызова функции, которые стоят в левой части, или отбрасывать их и вставлять новые; но он, конечно, не будет просматривать или копировать значения переменных **e1** и **e2**, так как это привело бы к нарушению требования 2 к эффективности интерпретации. За счёт манипулирования

адресами в компьютерной памяти значения **e1** и **e2** будут просто перемещены из одного места в другое. Эта операция потребует постоянного времени, которое не зависит от размера обрабатываемых выражений. При анализе алгоритмов обычно желательно знать, как время выполнения алгоритма зависит от размера обрабатываемых им объектов. Такая зависимость имеет решающее значение, в то время как постоянные кванты времени выполнения составляющих операторов, которые зависят от конкретной реализации алгоритма или от скорости компьютера, имеют второстепенное значение. Следовательно, в нашем случае работа РЕФАЛ-интерпретатора является безусловно эффективной.

На следующем шаге работы РЕФАЛ-машины весь аргумент **Blanks**, включая проекцию **e1**, будет просмотрен снова при поиске пары смежных пробелов. И это является очевидной потерей времени, так как проекция **e1** не может содержать такую пару. Это является недостатком нашей программы, и можно скорректировать её, вынося **e1** за пределы вычислительных скобок в правой части. Один пробел, разумеется, должен остаться слева внутри скобок. Определение приобретает вид:

```
Blanks {
  e1' ' 'e2 = e1 <Blanks ' 'e2>;
  e1 = e1; }
```

Теперь алгоритм, скрывающийся за нашим определением, полностью эффективен.

Когда вычисляется функция **Blanks**, левая скобка вызова функции используется как указатель, который отделяет обработанную часть строки от части, ещё не подвергавшейся обработке. Рассмотрим пример, где скобки вызова функции не могут использоваться подобным образом. Допустим, желательно определить корректирующую функцию **Correct**, уничтожающую в заданной строке символов всякий символ, за которым следует знак уничтожения **'#'**. Если в строке имеется несколько знаков уничтожения, функция ликвидирует соответствующее число предшествующих им символов. Так, если набрана строка **'Crocodyle'**, а затем замечена ошибка, можно продолжить:

```
Crocodyle###iles were everywe#here
```

и если **Correct** применяется к этой строке, результатом будет:

```
Crocodiles were everywhere
```

Прямое определение, которое хочется записать, не заботясь об эффективности, имеет вид:

```
Correct {
  e1 sA'#' e2  =  <Correct e1 e2>;
  e1  =  e1; }
```

Согласно этому алгоритму аргумент будет просматриваться с начала столько раз, сколько в нем встречается символов уничтожения. Для улучшения этой ситуации нельзя просто вынести строку **e1** за пределы скобок вызова функции, так как её правый конец может ещё нуждаться в коррекции. Что можно было бы сделать, если бы программирование велось на машинном языке? После первого шага можно было бы поместить указатель в ту позицию строки, где производится действие, т.е. где надлежит уничтожить символ:

```
Crocodyl^##iles were everywe#here
```

Указатель, который представлен символом **^**, является адресом в компьютере, который позволяет добраться до нужной позиции за один прыжок. Тогда на следующем шаге можно было бы не просматривать **Crocodyl**, но, просто совершив прыжок по указателю, увидеть, что следующим

символом снова является знак уничтожения, и уничтожить предшествующий символ **'l'**. На каждом следующем шаге можно было бы начинать просмотр снова с прыжка по указателю.

То же самое можно сделать и на РЕФАЛе. Структурные скобки будут служить указателями. В самом деле, согласно требованию эффективности 1, с каждой круглой скобкой должен быть связан адрес сопряжённой ей скобки, так что возможен мгновенный переход от одной скобки к другой. По отношению к функции **Correct** это означает, что следует заключать в скобки начало строки, которое уже просмотрено. Первоначальным аргументом должно быть:

```
() Crocodyle###iles were everywe#here
```

После первого использования знака уничтожения аргумент приобретёт вид:

```
(Crocodyl) ##iles were everywe#here
```

и т.д.. Правая скобка здесь по существу является указателем.

Если желательно сохранить формат функции **Correct** таким, каким он был определён ранее, т.е. просто **<Correct_e.String>**, то следует ввести вспомогательную функцию, скажем **Cor**, в формате:

```
<Cor (e.Scanned-already) e.Not-yet-scanned>
```

Эффективно будет работать следующее определение:

```
Correct {
  e1 = <Cor () e1>; }

Cor {
*1.A delete sign after the pointer;
*  step back and delete
  (e1 sX) '#'e2 = <Cor (e1) e2>;
*2.Look forward for the next delete sign;
*  delete and move pointer
  (e1) e2 sX '#'e3 = <Cor (e1 e2) e3>;
*3.No signes to delete
  (e1) e2 = e2; }
```

Упражнение 3.1

Проанализировать, что происходит при вычислении

```
<Correct 'c##Crocodile'>
```

Переопределить **Cor** таким образом, чтобы выдавалось предупреждение `'Extra delete signs!'` в подобных ситуациях.

3.2. ФОРМАТЫ ФУНКЦИЙ

С формальной точки зрения все РЕФАЛ-функции являются функциями одного аргумента. Однако очень часто этот единственный аргумент состоит из частей, которые по существу являются подаргументами (но обычно называются просто *аргументами*). Когда определяется функция, на самом деле представляющая собой функцию нескольких аргументов, производится объединение этих аргументов каким-либо образом, чтобы образовать единственный формальный аргумент. Это задаёт *формат* функций.

Программируя на РЕФАЛе, можно выбирать любые форматы. В принципе можно разделять подаргументы некоторыми символами, например, запятыми:

```
<F e1','e2>
<G e1','e2','e3>
```

Одним из недостатков этого метода является то, что, как известно, аргументы не должны включать разделители — в данном случае запятые. Но существует ещё более серьёзное возражение против таких форматов: они включают открытые e-переменные, которые подразумевают, что значения аргументов будут просматриваться даже тогда, когда это не требуется по сути алгоритма. Вот почему в форматах следует использовать круглые скобки.

Чтобы разделить два подвыражения, требуется только одна пара круглых скобок; для трёх подвыражений потребуется две пары, и т.д. Но ничто не мешает заключать каждый подаргумент в круглые скобки. Таким образом, имеется очевидная свобода в выборе форматов:

```
<F (e1)e2>
<F e1(e2)>
<F (e1)(e2)>
<F (e1)(e2)e3>
<F (e1)e2(e3)>
```

и т.д. Можно варьировать форматы, включая мнемонические символы или строки, например:

```
<F Graph (e.Gr) Start (e.St) Weights (eW)>
```

Настоятельно рекомендуется, чтобы форматы функций, когда они являются нетривиальными, снабжались в программе комментариями. Также желательно указывать форму и тип значения функции. Хотя такие комментарии не являются необходимыми (и обычно опускаются для простых функций), они значительно повышают читабельность программы и неоценимы при отладке. Конкретная форма комментариев оставляется на усмотрение пользователя. В этой книге используется следующая форма:

```
* < Function-name Argument >
*     ==   Result-1
*     ==   Result-2
 ... etc.
```

В то время как единственный знак равенства в РЕФАЛе символизирует один шаг РЕФАЛ-машины, сдвоенный знак равенства говорит о том, что должно произойти после некоторого точно не установленного количества шагов машины, т.е. когда процесс вычислений завершён. Часто результат может иметь одну из возможных форм. Если форма результата более или менее очевидна, эта часть опускается.

Комментарии к определению функции **Cor** из *Разд. 3.1* могли бы иметь такой вид:

```
* <Cor (e.Scanned-already) e.Not-yet-scanned>
*     ==  e.Corrected
```

Имя переменной, как всегда в программировании, может давать представление о том, чем она является. В РЕФАЛе имена переменных, используемые в различных предложениях, совершенно независимы; и, разумеется, имена переменных в комментариях не играют никакой роли. Но полезной является практика выбора имён переменных таким образом, чтобы они были идентичными или, по крайней мере, похожими в различных предложениях и так, чтобы они походили на имена переменных в форматах, даже если это односимвольное имя. Сопоставление предложения с форматом помогает читать

программу.

В следующем упражнении приводится пример функции с более сложным форматом.

Упражнение 3.2

Определить функцию **<Cut_e1__e2_>**, которая рассекает выражения **e1** и **e2** на части так, что они оканчиваются одинаковыми термами, имеющими одинаковые порядковые номера в соответствующих выражениях. Эти части должны быть заключены в круглые скобки и выданы на печать попарно. Оставшиеся части выражений, которые не сопоставляются, должны быть отброшены. Например, если два выражения имеют вид:

```
'abcc++'()'y'('!')'yxrtr'
'xyzz+'('==')'yy'('!')'**x'(())'pq'
```

то вывод на печать должен выглядеть следующим образом:

```
(abcc+)(xyzz+)
(+()y)((==)yy)
((!))((!))
```

В РЕФАЛе нет других типов данных, кроме трёх синтаксических категорий: символ, терм, выражение. Вследствие этого язык становится очень простым для программирования и формального анализа, все ещё позволяя легко имитировать в рамках его простого синтаксиса многое из того, что достигается введением типов данных. Предположим, например, что нужно оперировать с рациональными числами, которые представлены тремя символами: знак, числитель, знаменатель. Мы вводим имя **Rat** для такой структуры данных и используем её в следующей форме:

```
(Rat s.Sign s.Numerator s.Denominator)
```

Нет необходимости связывать специальный тип с переменными этого типа данных. Для произвольного рационального числа 'X' мы просто запишем:

```
(Rat eX)
```

Произвольное отрицательное рациональное число может быть записано как:

```
(Rat '-'eX)
```

Для каждого типа данных легко можно определить функции доступа, которые предусматривают выделение структурных компонент, например:

```
Numerator { (Rat sS sN sD) = sN}
```

Однако очень часто использование функций доступа не нужно; прямое использование образцов делает определение и более кратким, и более ясным. Например, умножение рациональных чисел можно определить следующим образом:

```
Mul{
  (Rat s.S1 s.N1 s.D1)(Rat s.S2 s.N2 s.D2) =
         (Rat <Mul-signs s.S1 s.S2>
              <* s.N1 s.N2> <* s.D1 s.D2>);
  }
```

где функция **Mul-signs** должна быть определена соответствующим образом. (Здесь, как и выше,

оставлен открытым вопрос о том, как представить рациональное число 0) .

3.3. НЕЯВНЫЕ И ЯВНЫЕ РЕКУРСИИ

Вернёмся к функции **Blanks**, определённой в *Разд. 3.1*. Теперь желательно модифицировать её так, чтобы исключались только те повторяющиеся пробелы, которые не заключены в кавычки — обычная ситуация в языках программирования. В этом случае невозможно использовать образец `e1' ' 'e2` с открытой e-переменной, но следует обозревать символы один за другим. Прежде всего, переопределим старую функцию **Blanks** следующим образом:

```
Blanks {
  ' ' 'e1 = <Blanks ' 'e1>;
  sA e1 = sA <Blanks e1>;
* End of job
  = ; };
```

Здесь открывающая скобка вызова функции ('указатель') продвигается посимвольно, в то время как результат накапливается в поле зрения. Теперь можно модифицировать это определение, добавив два предложения на случай, когда следующим символом является кавычка:

```
Blanks {
*1.Delete one of two adjacent blanks
  ' ' 'e1 = <Blanks ' 'e1>;
*2.If a quote, jump to the next quote
  ''e1'' e2 = ''e1'' <Blanks e2>;
*3.No pair for a quote: an error
  '' e1 =
    <Prout 'ERROR: no pair for quote in 'e1>;
*4.Regular symbol
  sA e1 = sA <Blanks e1>;
*5.End of job
  = ; };
```

Как упомянуто в Главе 1, изолированная кавычка представляется в РЕФАЛе (как и во многих других языках программирования) двойной кавычкой, и точно так же представляется кавычка внутри строки. Заметим, что **Blanks** корректно работает, если встречается кавычка — как изолированная, так и внутри строки.

Где бы ни встретились в алгоритме открытые переменные, они должны применяться с целью эффективности его выполнения. Если даже алгоритм с открытой переменной может быть по существу таким же, как в случае посимвольной обработки, ряд шагов РЕФАЛ-машины может стать короче. Из-за накладных расходов реализации шага программа с открытыми переменными будет работать быстрее. В последнем определении **Blanks** оставлены открытые переменные, а именно **e1** в предложении 2, для того, чтобы совершить прыжок прямо к закрывающей кавычке.

Следует упомянуть, однако, что когда имеется более чем одна открытая переменная, универсальный алгоритм сопоставления, описанный в Главе 2, может и не являться наиболее эффективным решением проблемы.

Рассмотрим предложение:

```
<Sub-a-z e1 'a'eX'z' e2> = (e1) 'a'eX'z' (e2)
```

Оно выделяет подстроку, которая начинается с **'a'** и заканчивается **'z'**. Если аргумент в самом деле

содержит такую подстроку, алгоритм будет работать вполне эффективно. Первое вхождение **'a'** будет обнаружено, а затем и первое вхождение **'z'** за ним. Но предположим, что аргумент не содержит ни одного символа **'z'**, однако в нем есть множество символов **'a'**, например:

```
'abababababababababababababababababababaabab'
```

Тогда машина выделяет первое **'a'**, терпит неудачу в поиске **'z'**, удлиняет **e1**, вновь терпит неудачу в поиске **'z'**, вновь удлиняет **e1**, и т.д.: наблюдается квадратичная зависимость количества операций от длины выражения вместо линейной.

Во избежание подобной неэффективности, следует встроить обработку в две функции, наподобие:

```
Sub-a-z {
  e1'a'e2 = <Find-z (e1'a')e2>;
/* Find 'a' and pass the argument to Find-z */
  e2 = <Prout 'No substring a-z'>; }

Find-z {
  (e1'a') eX'z'e2 = (e1) 'A'e1'z' (e2);
  (e1) e2 = <Prout 'No substring a-z'>;
      }
```

Вообще говоря, всегда есть надёжное средство, чтобы представить линейный просмотр как процесс сопоставления с одной открытой переменной. При наличии нескольких открытых переменных, если речь идёт об эффективности, необходим специальный просмотр в каждом отдельном случае. Переопределение с целью избежать возможной неэффективности, как в приведённом выше примере, обычно является довольно лёгким.

Открытые и повторные e-переменные, так же как и повторные t-переменные, позволяют скрывать за образцом рекурсию. Если по какой-либо причине желательно все циклы рекурсии сделать явными (и мы действительно хотим этого в ходе функциональной трансформации), можно переписать каждое определение таким образом, чтобы оно не использовало переменные указанных типов. Например, функция **Sub-a-z** не выглядит рекурсивной: она либо вызывает **Find-z** (которая никогда не вызывает **Sub-a-z**), либо встроенную функцию **Prout**. Но на самом деле она является рекурсивной из-за наличия открытой переменной в первом предложении. Она может быть переопределена следующим образом:

```
Sub-a-z {
  e1 = <Sub-a-z-1 ()e1>; }

Sub-a-z-1 {
*1.'a' is found. Control is passed to Find-z.
  (e1) 'a'e2 = <Find-z (e1'a') e2>;
*2.Recursion: jump over any symbol distinct
*   from 'a'.
  (e1) sX e2 = <Find-a-z-1 (e1 sX) e2>;
*3.The string is exhausted without finding 'a'
  (e1) = <Prout 'No substring a-z'>; }
```

Можно было бы заметить, что используется некая система в переписывании определений этой функции. Она состоит в заключении открытых переменных в скобки и во введении вспомогательных функций, которые имитируют удлинение открытых переменных. Этот процесс исключения неявной рекурсии можно легко запрограммировать и выполнять автоматически.

Упражнение 3.3

Функцию проверки равенства произвольных выражений (не обязательно являющихся строками) можно определить следующим образом:

```
Equal {
  (e1)(e1) = T;
  (e1)(e2) = F; }
```

Переопределить её так, чтобы исключить неявную рекурсию.

Упражнение 3.4

Исключить неявную рекурсию в следующем определении:

```
F {
  e1 sX e2 sX e3 = sX e2 sX;
  e1  = <Prout 'No repeated symbols'>; }
```

Сравнить оба определения по эффективности.

Упражнение 3.5

Конечное множество символов может быть представлено строками, включающими те и только те символы, которые входят в это множество. Определить функцию **<Isect (e.Set1)e.Set2>**, которая вычисляет пересечение двух множеств. Определить её как с использованием открытых e-переменных, так и без него.

3.4. ДУБЛИРОВАНИЕ ПЕРЕМЕННЫХ

Если количество вхождений свободной переменной в правую часть предложения превышает количество её вхождений в левую часть, её значение должно быть продублировано интерпретатором, когда он применяется к этому предложению. Это следует принимать во внимание при программировании, так как бесполезное дублирование переменных может приводить к существенным потерям в эффективности. В более сложных реализациях РЕФАЛа, чем простой интерпретатор, излишнего дублирования объектных выражений можно избежать, используя указатели для некоторых частей исходных выражений. Но, программируя для интерпретатора и мысля о РЕФАЛ-программах в терминах простого пошагового выполнения, следует принимать на себя полную ответственность за избежание бесполезного дублирования.

Эти рассмотрения имеют прямое отношение к способу, которым определяется ветвление в РЕФАЛе. Не представило бы труда определить семантику условных if-then-else выражений в РЕФАЛе, а затем использовать её, как она применяется, скажем, в Паскале. Можно было бы принять решение об использовании **T** и **F** в качестве значений истинности, и определить функцию **If** в формате:

```
<If e.Condition
 Then (e.If-True)
 Else (e.If-False)>
```

Когда используется **If**, активное выражение, вычисляющее значение истинности условия, должно быть подставлено вместо **e.Condition** . Затем **If** можно было бы просто определить как:

```
If {
  T Then (e1) Else (e2) = e1;
  F Then (e1) Else (e2) = e2;  }
```

Логические связки можно было бы применять подобным же образом. Конъюнкция, например, могла бы быть определена посредством функции **And**:

```
And {
  T sX = sX;
  F sX = F; }
```

Однако, такой подход приводит к излишним дублированиям. Это будет продемонстрировано на примере процедуры, которая упорядочивает и сцепляет строки.

Предположим, имеется некоторое отношение предшествования (порядка), заданное на строках символов посредством предиката

```
<Pre (e1)(e2)>
```

который принимает значение **T**, если строка **e1** предшествует строке **e2**, и **F** в противном случае. Можно определить функцию **Order**, которая сцепляет две строки в правильном порядке, следующим образом:

```
Order {
  (e1)e2  = <If (<Pre (e1)(e2)>)
            Then (e1 e2)
            Else (e2 e1)>; }
```

Пока, при такой интерпретирующей реализации РЕФАЛа, как наша, эта программа не является наилучшим решением. Каждая из переменных **e1** и **e2** появляется один раз в левой части и три раза в правой части. Это означает, что в ходе выполнения будет создано по две копии каждой строки; если первоначальное выражение сохраняется для вывода, то одна из копий будет использована при вычислении предиката **Pre**, а другая будет всегда пропадать, так как машина использует только одно из двух выражений для двух значений истинности.

Исключим сперва вопиющую неэффективность создания двух копий (в then- и else-фрагментах), когда нам известно, что только одна из них будет использована на самом деле. Чтобы это проделать, разъединим фрагменты, помещая их в различные предложения. **Order** вызывает предикат **Pre** и передаёт результат, вместе с исходными строками, на вход вспомогательной функции **Order1**, которая имеет два предложения для двух фрагментов и осуществляет выбор одного из них (более элегантный способ определения **Order** без вспомогательной функции будет продемонстрирован в следующей главе, когда будет введено расширение базисного РЕФАЛа):

```
Order { (e1)e2 = <Order1 <Pre (e1)(e2)>(e1)e2> }

Order1 {
  T (e1)e2 = e1 e2;
  F (e1)e2 = e2 e1; };
```

Вместо определения ветвления в алгоритме посредством функции **If**, или какой-либо другой специальной функции, используем тип ветвления, встроенный в наш язык: выбор предложения в зависимости от аргумента. Функция **Pre** добавляет к аргументу **Order1** индикатор, **T** либо **F**, посылая **Order1** сигнал, в котором она нуждается для того, чтобы сделать выбор. Этот вид ветвления в

РЕФАЛе является и более естественным, и более эффективным. Он также не ограничивает количество альтернатив двумя, но действует как устройство, известное в других языках программирования как *переключатель*, или как *case-выражение*.

Наше определение **Order** потребует создания только одной копии переменных **e1** и **e2**. Должны ли мы пытаться исключить и это копирование?

Это зависит от ожидаемого размера подобных выражений (и, конечно, от того, насколько часто предполагается вычислять эту функцию). Допустим, **e1** и **e2** являются английскими словами, а **Pre** основывается на лексикографическом порядке. Чтобы определить **Pre**, мы должны сперва задать отношение предшествования в алфавите **Pre-alph** для букв. В базисном РЕФАЛе это можно проделать следующим образом:

```
Pre-alph {
*1.This relation is reflexive
  s1 s1 = T;
*2.If the letters are different, see whether the
*  first is before the second in the alphabet
  s1 s2 = <Before s1 s2 In <Alphabet>>; }

Before {
  s1 s2 In eA s1 eB s2 eC = T;
  eZ = F; }

Alphabet { = 'abcdefghijklmnopqrstuvwxyz'; }
```

Теперь не представляет труда определить лексикографический предикат **Pre**:

```
Pre {
  ()(e2) = T;
  (e1)() = F;
  (s1 eX)(s1 eY) = <Pre (eX)(eY)>;
  (s1 eX)(s2 eY) = <Pre-alph s1 s2>;
   }
```

Так как длины английских слов невелики, их копирование не приводит к существенной потере эффективности. Это оправдано благодаря простому и естественному определению предиката **Pre**. Однако в некоторых случаях может быть желательно избегать копирования. Например, если строки состоят не из английских букв, а из некоторых выражений, которые могут быть очень велики, хотелось бы переопределить **Pre** так, чтобы не производилось ни одного лишнего копирования.

Причиной, почему следует копировать **e1** и **e2** в определении **Order**, является то, что **Pre** ликвидирует свои аргументы и замещает их единственным символом: значением истинности. Таким образом, может быть сформулировано следующее общее правило:

Для того, чтобы избежать неоправданного дублирования аргументов, следует избегать неоправданного уничтожения аргументов.

Будем рассматривать такие функции, которые не уничтожают свои аргументы, а просто добавляют к ним **T** или **F**, как сохраняющие предикаты. Если вместо **Pre** используется сохраняющий предикат **Prec**, функция **Order** может быть переопределена как

```
Order { (e1)(e2) = <Order1 <Prec (e1)(e2)>; }
```

Допустим теперь, что формат **Prec** определён таким комментарием:

```
* <Prec (e1)(e2)>  ==  T(e1)(e2)
*                  ==  F(e1)(e2)
```

Для того, чтобы определить **Prec**, модифицируем определение **Pre** в двух пунктах. Во-первых, обобщаем его до обработки термов вместо простых символов, и заменяем **Pre-alph** на **Pre-term**. Во-вторых, новый предикат должен стать сохраняющим. Этого можно достичь, не изменяя формата аргумента. Изменения очевидны для предложений 1, 2 и 4. Но в случае, который соответствует предложению 3, следует хранить первый терм, который является общим для обоих аргументов, для того, чтобы добавить его к ним обоим после того, как предшествование установлено. Решением является:

```
Prec {
  ()(e2) = T()(e2);
  (e1)() = F(e1)();
  (t1 eX)(t1 eY) = <Add-c t1 <Prec (eX)(eY)>>;
  (t1 eX)(t2 eY) =
           <Pre-term t1 t2>(t1 eX)(t2 eY); }

Add-c { t1 sT(eX)(eY) = sT(t1 eX)(t1 eY); }
```

Упражнение 3.6

Решение, предложенное выше, не самое лучшее. Запустите

```
<Prec ('but')('butter')>
```

с **Pre-alph** как **Pre-term** чтобы увидеть, как он сработает. Предполагая, что длина общего начала двух строк равна *c*, а верхняя граница количества шагов для вычисления **Pre-term** равна *t*, определить количество шагов, необходимое для вычисления **Prec**. Переопределить его так, чтобы он работал быстрее. Указание: добавить один или более ящиков к формату **Prec** для накопления общей части. Определить количество шагов, которое потребуется при этом определении.

Как упоминалось выше, не всегда необходимо избегать дублирования. Иногда оно является частью алгоритма. Если это и не так, дублирование часто не приводит к существенной потере эффективности. Например, если алгоритм включает просмотр выражения по одному символу за шаг РЕФАЛ-машины, дополнительное дублирование этого выражения ненамного увеличит время выполнения. Дублирование символов или коротких выражений никогда не было проблемой. Дублирование громоздких выражений на верхнем уровне алгоритма также обычно не составит труда, так как время, в основном, уходит на внутренние циклы. Наконец, более изощрённые реализации РЕФАЛа, которые разрабатываются в настоящее время, используют указатели для избежания ненужного копирования.

Пока, программируя для РЕФАЛ-интерпретатора, следует думать о свободных переменных не как об указателях на значения, которые где-то размещены, но скорее как о самих значениях. Когда элементы левой части переставляются или размножаются, в правой части предложения этот процесс будет повторен во время выполнения. Недозволенного копирования переменных следует решительно избегать.

Упражнение 3.7

Определить следующие сохраняющие предикаты:

(a) **<String eX>**, истинен тогда и только тогда. когда **eX** является строкой (бесскобочной).

(b) **<Pal eX>**, истинен тогда и только тогда, когда **eX** является палиндромом.

3.5. ВСТРАИВАНИЕ АЛГОРИТМОВ В ФУНКЦИИ

Алгоритм на РЕФАЛе является набором функций. Для того, чтобы определить функцию, рассматриваются различные варианты её аргумента и записываются соответствующие предложения. Таким вот образом функциональные определения встраиваются в предложения. А как некоторый алгоритм встраивается в функции? Далее следует краткое изложение обычных обоснований для введения новой функции, которая рассматривается как алгоритм.

1. ПОСЛЕДОВАТЕЛЬНАЯ ОБРАБОТКА

Когда операция, которую мы хотим выполнить над объектом, может быть определена как последовательное выполнение нескольких операций, мы определяем каждую из операций соответствующей функцией, например:

```
F { eX = <F3 <F2 <F1 eX>>> }
```

Здесь результат одной функции становится аргументом для другой.

2. ПАРАЛЛЕЛЬНАЯ ОБРАБОТКА

Когда различные части объекта должны подвергнуться обработке по отдельности, мы определяем функции, которые применяются к соответствующим частям, например:

```
F { (eX)(eY)eZ = <F1 eX> <F2 eY> <F3 eZ> }
```

3. РАЗБИЕНИЕ ВЫРАЖЕНИЯ НА ЧАСТИ

В результате выполнения предыдущих шагов мы можем получить структурированный объект, различные части которого подразумевают различные применения. Например, функция деления нацело **</ s.N1 s.N2>** продуцирует комбинацию **(s.Quotient)s.Remainder**. Если нам нужен только остаток от деления **s.N1** на **s.N2**, мы вызываем:

```
<Rem </ s.N1 s.N2>>
```

где **Rem** определяется предложением:

```
Rem { (sQ) eR> = eR }
```

4. ВЕТВЛЕНИЕ ПО ЗНАЧЕНИЮ ФУНКЦИИ

Часто возникает потребность вычислить некоторую функцию **F**, а затем, в зависимости от результата вычисления, продолжать вычисления тем или иным путём. Тогда мы вводим вспомогательную функцию **F1**, и производим вызов:

```
<F1 <F eX>>
```

Функция ветвления **F1** анализирует результат **F** и производит требуемый выбор. Мы уже видели многие примеры этого.

5. ИЗМЕНЕНИЕ ФОРМАТА

Часто требуются дополнительные подаргументы функции для того, чтобы определить рекурсивный процесс. В этом случае определяется вспомогательная функция с требуемым форматом. В приведённом выше примере было видно, что мы определили функцию **Cor** с форматом:

```
<Cor (e.Scanned-already) e.Not-yet-scanned>
```

для того, чтобы определить функцию **Correct**:

```
Correct {
  e1 = <Cor () e1>; }
```

Взаимодействие между функциями в РЕФАЛ-системе может производиться либо через поле зрения, либо посредством прямых вызовов. В примере последовательной обработки, приведённом выше:

```
F {eX = <F3 <F2 <F1 eX>>> }
```

мы использовали взаимодействие через поле зрения. Каждая из трёх функций, **F1** и т.д., ничего не знает о других. Она оставляет своё значение в поле зрения, а следующая функция принимает его. Альтернативой этому является определение **F** через систему прямых вызовов. **F** сама просто вызывает **F1**:

```
F { eX = <F1 eX> }
```

Определение **F1** теперь должно быть модифицировано. В нерекурсивных предложениях, где прежнее определение **F1** заканчивало работу с выражением в качестве результата, новое определение должно оканчиваться **<F2_.>**. Определение **F2** должно быть модифицировано таким же образом.

Взаимодействие через поле зрения имеет то преимущество, что функциональные определения независимы от их применения, так что они могут быть использованы также и в других контекстах. Итак, по этому методу мы можем надстраивать результат функции справа в поле зрения, например, по схеме, которая уже использовалась (см. функции **Chpm** , **Blanks** , и т.д.):

```
F {... =  E   <F ...> }
```

где **E** есть некоторое выражение. Эта схема может варьироваться:

```
F {... = <F ...> E }
```

или

```
F {... = E1(<F ...>)E2 }
```

и т.д. При прямом вызове функций-преемников это невозможно. С другой стороны, прямые вызовы удобны, когда функция имеет различных преемников в различных предложениях.

3.6. РЕКУРСИЯ И ИТЕРАЦИЯ

В программировании термин *рекурсия* имеет специальное значение и часто противопоставляется *итерации*. Рекурсия и итерация являются двумя основными способами определения циклических процессов. Сравним их, взяв в качестве примера функцию факториал и используя в качестве языка программирования Паскаль. Итеративным определением является:

```
function fact(n:integer):integer;
var f:integer;
begin
   f := 1;
   while n > 0 do
   begin f := n * f;
         n := n - 1
   end;
   fact := f
end
```

Здесь заголовок сообщает нам, что **fact** является функцией целочисленной переменной; **n** является формальным параметром. Ещё одна целочисленная переменная, **f**, используется при вычислении. Вначале она принимает значение 1. Затем включается проверка, превосходит ли **n** значение 0. До тех пор, пока это не выполняется, вычисляются новые значения **f** и **n**, а управление возвращается на проверку условия. Это простой цикл с двумя переменными, изменяющими своё значение в ходе вычислений.

Сравним это с рекурсивным определением той же самой функции:

```
function fact(n:integer):integer;
begin
   if n = 0 then fact := 1
            else fact := n * fact(n-1)
end
```

Когда выполняется эта программа, аргумент **n** сперва сравнивается с 0. Если он не является 0 (и подразумевается, что он неотрицателен), то **n** должно быть умножено на значение той же самой функции **fact** для меньшего аргумента **n-1**. Это означает, что вычисление вычисления первоначального функционального вызова должно быть приостановлено и **fact(n-1)** должно быть вычислено посредством той же программы. Текущее значение **n** и **fact** (хотя последнее и не будет использоваться на самом деле) должны быть сохранены в памяти для продолжения вычислений **fact(n)** после того, как будет вычислено **fact(n-1)**. Эта процедура откладывания будет повторена **n** раз, пока текущее значение **n** не станет равным 0. Затем отложенные вызовы возобновляются и выполняются в обратном порядке так, что наконец мы приходим к ответу. Говорят, что отложенные вызовы функции в процессе рекурсивного вычисления образуют *стэк* (они размещаются один над другим, а используются в обратной последовательности). Таким образом, перед тем, как числа действительно начинают перемножаться, следует запомнить 2*n* чисел. К тому же следует организовать правильное возобновление отложенных вызовов функции. Очевидно, что рекурсивное вычисление более сложно и требует больше времени и компьютерной памяти, чем итеративное.

РЕФАЛ-функция является рекурсивной, если она вызывает себя самое непосредственно или в результате последовательности вызовов других функций. Но рекурсия в этом смысле не обязательно означает рекурсивность в терминах Паскаля. Это отличие проистекает из отличий в семантике функционального вызова. В Паскале, когда функция сама себя вызывает, она откладывает выполнение программы, чтобы вернуться к нему позднее. В РЕФАЛе же правая часть предложения замещает вызов функции. Если функция вызывает сама себя с другим аргументом:

```
F {... = <F ...> }
```

это, в терминах Паскаля, является итерацией, но не рекурсией: вызов функции воспроизводит сам себя,

изменяется только аргумент. Когда управляющая структура имеет вид:

```
F1 { ... = <F1 ...<F2 ...>...> }
```

либо

```
F1 ... = <F2 ...> <F1 ...>
```

это снова является присущим 'Паскаль-итерации', но не 'Паскаль-рекурсии'. Вызов **F2** вычисляется первым, и снова имеется единственный функциональный вызов **F1**. Не происходит никакого накопления вызовов в поле зрения.

Настоящая 'Паскаль-рекурсия' возникает за счёт структур следующего вида:

```
F1 {... = <F2 ...<F1 ...>...> }
```

либо

```
F1 {... = <F1 ...> <F2 ...> }
```

В РЕФАЛе не существует прерываемых, или отложенных, вызовов функций, так как работа РЕФАЛ-машины состоит просто из последовательности замещений. Но функциональный вызов, как единое целое, может ожидать своей очереди в поле зрения. Вот что происходит с вызовами **F2**: они накапливаются в поле зрения, подобно отложенным вызовам Паскаль-функции **fact**.

Ранее уже представлялся случай продемонстрировать рекурсивное определение факториала в терминах РЕФАЛа:

```
Fact {
  0 = 1;
  sN = <* sN <Fact <- sN 1>>>; }
```

Когда вычисляется **<Fact 10>**, первые 20 шагов порождают структуру:

```
<* 10 <* 9 ... <* 1 <Fact 0>> ... >>
```

затем работает первое предложение из определения **Fact**, и десять последующих шагов умножения приводят к результату. Это весьма похоже на случай Паскаль-функции, но отсроченные вызовы обозначены *****, а не **Fact**.

Можно переопределить **Fact** в виде итерации, полностью аналогично Паскаль-определению. Функция, которая будет выполнять работу, должна будет включать все переменные, используемые в цикле, а именно переменную цикла **s.n**, которая первоначально является такой же, как аргумент, и накапливаемый результат **s.f**, который первоначально равен 1. Первым шагом поэтому будет преобразование формата:

```
Fact { s.n = <Loop s.n 1>; };
```

Здесь нет ничего, кроме присваивания начальных значений переменным цикла. Сам цикл определяется следующим образом:

```
Loop {
*1.Termination condition satisfied
  0 s.f = s.f;
*2.Otherwise
```

```
    s.n s.f = <Loop <- s.n 1> <* s.n s.f>>; }
```

Функциональные определения такого типа, каким задаётся **Loop**, известны под названием *остаточная рекурсия*. В функциях этого вида расширение функционального вызова включает не более одного рекурсивного вызова функции, и, коль скоро он имел место, он должен быть последней операцией в расширении. Из-за этого остаточно-рекурсивные функции допускают реализацию, которая является итеративной, т.е. исключает накопление отсроченных вызовов функций. Трансляторы для Паскаля и для других универсальных языков не используют эту возможность и транслируют остаточно-рекурсивные функциональные определения таким же образом, как и общие рекурсивные определения. Это также справедливо для ранних реализаций Лиспа. В самой последней реализации Лиспа и её версиях (например, Scheme) позаботились о реализации остаточной рекурсии в форме итерации. В РЕФАЛе нет нужды уделять специальное внимание остаточной рекурсии: подобные определения исполняются итеративно в силу самой семантики языка.

Итеративные (остаточно-рекурсивные) определения используют память экономно, в то время как 'по-настоящему рекурсивные' определения могут породить громадные промежуточные структуры в поле зрения. Но с другой стороны рекурсивные определения, как правило, короче и более прозрачны.

Рассмотрим характер рекурсии на примере очень простой функции обработки строк, которая трансформирует каждое **A** в **B**:

```
Fab {
  A e1 = B <Fab e1>;
  s2 e1 = s2 <Fab e1>;
   = ; }
```

Будучи написанным и исполняемым в РЕФАЛе, это определение является итеративным. В самом деле, расширение вызова функции включает не более одного функционального вызова; символы накапливаются в поле зрения как части окончательного результата, а накопления функциональных вызовов не происходит. И все же само определение рекурсивно по форме, а не итеративно. Если записать полностью аналогичное определение в Лиспе:

```
(define (fab x)
  (cond ((null x) nil)
        ((equal (car x)(quote a))
          (cons (quote b) (fab (cdr x))))
         (T (cons (car x)(fab (cdr x))))
  ))
```

то оно определённо является рекурсивным, но не остаточно-рекурсивным: отложенные вызовы функции-конструктора **cons** накапливаются в памяти до тех пор, пока значение аргумента функции **fab** не становится равным **nil** (пустым). Различие проистекает из разных способов трактовки основных операций над строками и выражениями, т.е. сопоставления с образцом и подстановки. В РЕФАЛе эти операции являются частью встроенного кибернетического механизма, который трактует их как физические объекты. Обрабатываемые символы (либо громоздкие выражения в иных случаях) накапливаются в соответствующих местах поля зрения по мере поступления, и нет необходимости откладывать какие-либо вызовы функций. При таком способе ясность и краткость рекурсивного определения сочетаются с эффективностью итеративного определения. Если необходимо использовать специальный конструктор **Cons** для конкатенации (сцепления) термов в левой части выражения, как в случае Лиспа, наше выражение могло бы быть 'по-лисповски рекурсивным':

```
Fab {
  A e1 = <Cons B <Fab e1>>;
  s2 e1 = <Cons s2 <Fab e1>>;
   = ; }
```

что приводит к накоплению отложенных вызовов функции.

Легко переписать первое определение **Fab** в классической остаточно-рекурсивной (итеративной) форме:

```
Fab {e1 = <Fab1 ()e1>; }
Fab1 {
  (e1) A e2 = <Fab1 (e1 B) e2>;
  (e1) s3 e2 = <Fab1 (e1 s3) e2>;
  (e1)  = e1; }
```

Но нельзя то же самое проделать в Лиспе (до тех пор, пока его структуры данных не будут расширены за счёт включения специальных функций для обработки строк). Как видно из первых двух примеров задания **Fab1**, последний обрабатываемый символ должен быть добавлен *справа* к текущему результату трансформации **e1**. В Лиспе же **cons** добавляет терм к списку только слева: либо необходимо использовать функцию **append** (которая совершенно неэффективна), либо конструировать строку в обратном порядке, а затем реверсировать её, приводя к первоначальному виду (что удваивает время выполнения).

Упражнение 3.8

Функция факториал может быть вычислена перемножением чисел в порядке возрастания. Определить РЕФАЛ-функцию, которая использует этот алгоритм.

Упражнение 3.9

Определить функцию **Reverse**, которая переписывает строку в обратном порядке (реверсирует), как рекурсивным, так и итеративным способами. Проанализировать, как они ведут себя в смысле времени выполнения. Какое определение предпочтительнее?

3.7. РАБОТА С ВЛОЖЕННЫМИ СКОБКАМИ

В РЕФАЛе отношение к скобкам весьма серьёзно. Ни при каких обстоятельствах нельзя трактовать их как символы (разумеется, это не распространяется на символы **'('** и **')'**, которые ничем не отличаются от остальных символов). Выражение представляет собой дерево, в котором символы являются листьями, а заключённые в скобки термы — поддеревьями. Если мы захотим, например, заменить **'+'** на **'-'** на всех скобочных уровнях, мы сможем проделать это только при точном определении процесса расстановки вложенных скобок. Простейшее решение таково:

```
* Change every '+' to '-', depth-first order.
Chpm {
  '+'e1 = '-'<Chpm e1>;
  sX e1 = sX <Chpm e1>;
  (e1)e2 = (<Chpm e1>) <Chpm e2>;
   = ;  }
```

Этот способ обработки деревьев известен как метод "внутри-слева". Всякий раз, когда встречается поддерево, т.е., мы натыкаемся на левую скобку, сперва производится переход вниз (внутрь), как

предписывает первый вызов **Chpm** в предложении 3, и только после того как полностью обработано это поддерево, производится второй вызов функции.

Вот другой способ подробного определения **Chpm**:

```
* Change every '+' to '-' using open e-variables
Chpm {
  e1 '+' e2 = <Chpm-d e1> '-' <Chpm e2>;
  e1 = <Chpm-d e1>;  }

* Go down with Chpm
Chpm-d {
  e1 (e2) e3 = e1 (<Chpm e2>) <Chpm-d e3>;
  e1 = e1;  }
```

Теперь порядок, в котором обрабатывается аргумент, иной.

Когда длинное выражение просто выдаётся на печать построчно, его может быть трудно прочесть. Желательно написать процедуру ("красивой печати"), которая распечатывала бы выражение в более читабельной форме. Точнее говоря, желательно, чтобы все скобки были напечатаны в отдельных строках с отступом, пропорциональным структурному уровню терма. Поэтому правые скобки следовало бы печатать в той же колонке, что и сопряжённые им левые скобки.

Следующая далее программа решает эту проблему. Идея её состоит в том, чтобы хранить строку пробелов как отступ и удлинять её при входе в новый скобочный уровень.

```
* Pretty print-out of expressions
$ENTRY Pprout { eX = <Pprt (' ') eX>;  }
    /* Initial offset 1 to see transfers */

* <Pprt (e.Blanks-offset) e.Expression>
Pprt {
  (eB) e1(e2)e3 = <Pr-ne (eB) e1>
                  <Prout eB'('>
                  <Pprt (eB' ') e2>
                  <Prout eB')'>
                  <Pprt (eB) e3>;
  (eB) e1 = <Pr-ne (eB) e1>;  }

* Print if non-empty
Pr-ne {
  (eB) = ;
  (eB) e1 = <Prout eB e1>;  }
```

Если на вход **Pprout** поступает следующее выражение:

```
'000'('111111')'00000'(('222')'111111')'0000'
```

то оно будет распечатано таким образом:

```
000
(
 111111
)
00000
(
```

```
 (
  222
 )
 111111
)
0000
```

Упражнение 3.10

Функция **Pprout**, определённая выше, использует целую строку для каждой скобки. Распечатка может быть несколько сжата, если левая скобка не вызывает появления новой строки:

```
000
(111111
)
00000
((222
 )
 111111
)
0000
```

Переопределить **Pprout** так, чтобы распечатывать выражения указанным образом.

Функция **Chpm**, определённая выше, может применяться независимо к отдельным ветвям дерева. Такие функции особенно легко определять с помощью рекурсии. Однако часто деревья обрабатываются таким образом, что информация передаётся от одной ветви к другой. Например, мы могли бы захотеть обойти дерево при поиске "внутри-слева" и сохранить только те листья, которые появляются только первый раз. Для этой цели нам необходим указатель, который продвигается по дереву, т.е., проходит все структурные уровни выражения.

Некоторый опыт работы с указателями в РЕФАЛе уже приобретён. Когда обрабатывается строка, положение указателя может быть представлено парой скобок; используется

```
( 'ABCDEF' ) 'GHIJKLM'
```

вместо

```
'ABCDEF' ^ 'GHIJKLM'
```

Если бы требовалось применять **Chpm** только к строкам, можно было бы определить её как использующую указатель вместо накопления результата в поле зрения:

```
* Change every '+' to '-' in a string
Chpm { e1 = <Chpm1 ()e1>; }
    /* set pointer at the beginning */

Chpm1 {
 (e1) '+'e2 = <Chpm1 (e1'-') e2>;
    /* move pointer and replace +*/
 (e1) sX e2 = <Chpm1 (e1 sX) e2>;
    /* move pointer */
 (e1) = e1;
    /* end of work */
     }
```

В такой форме наше определение можно модифицировать, чтобы организовать использование в трансформации обеих частей строки. Предположим, например, что **'+'** не просто замещается на **'-'** , но замещается на всю предшествующую строку, таким образом дублируя её (вообще-то глупая вещь, но достаточно хорошая в качестве иллюстрации). Тогда первое предложение для **Chpm1** должно быть переписано как:

```
(e1) '+'e2 = <Chpm1 (e1 e1) e2>;
```

Мы могли бы не делать этого, если бы результат накапливался в поле зрения.

Но что делать, если мы обрабатываем выражение и хотим использовать указатель *внутри* скобок? Рассмотрим, например, такое *размеченное выражение*, т.е. выражение с указателем:

```
'AB'('C'('DEF' ^'GH')'KLM'('V')'XY')
```

Часть выражения, которая предшествует указателю, сама не является выражением и не может быть заключена в скобки. То же справедливо и для части, следующей за указателем. Назовём эти части *мультискобками*, левой и правой соответственно. Можно принять соглашение о представлении мультискобок с помощью выражений. Имеются два естественных представления: с последовательными либо вложенными форматными скобками. Обсудим подробно последовательное представление, которое более наглядно и легче для употребления.

В левой мультискобке некоторое количество непарных скобок разделяет выражение на *n* сегментов. В правой мультискобке существует в точности то же количество *n* сегментов, разделённых правыми скобками. Для того, чтобы представить мультискобку выражением, каждый сегмент заключается в пару скобок. Для левой мультискобки:

$$E_1(E_2(\ldots\ (E_n \Longrightarrow (E_1)(E_2)(\ldots)(E_n)$$

Для правой мультискобки:

$$E_1)E_2)\ldots)E_n \Longrightarrow (E_1)(E_2)(\ldots)(E_n)$$

Размеченное выражение, упомянутое выше, принимает вид:

```
('AB')('C')('DEF') ^('GH')('KLM'('V')'XY')()
```

Теперь можно заменить указатель ещё одной парой скобок, и это будет окончательным представлением размеченных выражений:

```
01    11   11     1 0 1    11     2   2    111
(('AB')('C')('DEF') ) ('GH')('KLM'('V')'XY')()
```

Пара скобок, представляющих указатель, помечена уровнем 0; все другие скобки помечаются, начиная с 1 и игнорируя указательные скобки. Скобки, имеющие уровень 1, являются форматными для мультискобки; все остальные являются первоначальными скобками исходных выражений. Для прочтения мультискобочного представления исходного размеченного выражения, производятся следующие действия:

```
                01          1
1.Игнорировать (( слева, и ) справа.

          101
2.Читать ))(  как указатель.
```

```
        11
3.Читать )( слева/справа от указателя
как непарную левую/правую скобку
в левой/правой мультискобке.
```

Теперь остаётся только сделать последний шаг для удобства применения мультискобочного формата. Примем **e.ML** и **e.MR** в качестве стандартных переменных для заключающих левой и правой мультискобок, и введём следующие обозначения:

```
[  stands for (e.ML (
]  stands for )e.MR
[. stands for ((
.] stands for )
^  stands for ))(
```

Эта таблица должна сопровождаться толкованием, что внутри квадратных скобок каждая 'непарная' круглая скобка, т.е., та, для которой её пара расположена по другую сторону указателя, заменена инвертированной парой **)(** .

При такой системе обозначений размеченные мультискобочные выражения представляются наиболее естественным образом. Размеченное выражение, рассмотренное выше, приобретает вид:

```
[.'AB'('C'('DEF' ^'GH')'KLM'('V')'XY').]
```

Единственным дополнением к простому введению указателя в нужной позиции является заключение всего выражения в квадратные скобки с точками.

Квадратные скобки без точек используются при записи предложений в мультискобочном формате. Предложение, согласно которому функция **F** перемещает указатель на одну позицию вправо, имеет вид:

```
[ e1 ^ sX e2 ] = <F [ e1 sX ^ e2 ]>;
```

Предложение для входа в левую скобку имеет вид:

```
[ e1 ^ (e2)e3 ] = <F [ e1 ( ^ e2)e3 ]>;
```

Вот каким образом это соответствует действительному предложению:

```
      [   e1  ^ (e2)e3   ]  =
   (e.ML( e1 ))( (e2)e3 )e.MR =

              [   e1 (   ^ e2 )  e3   ]
           (e.ML( e1 )( ))( e2 )( e3 )e.MR
```

Чтобы транслировать предложение в мультискобочном формате в обычное предложение, используется приведённая выше таблица и производится замещение непарных скобок на инвертированные пары **)(**. Повторяющиеся квадратные скобки употребляются для специального случая, когда мультискобочные переменные **e.ML** и **e.MR** являются пустыми; это применяется для вызова и закрытия функций, использующих мультискобочный формат. Предложение, которое запускает мультискобочную обработку некоторого выражения **e1**, помещая указатель перед ним, имеет вид:

```
e1 = <F [. ^ e1 .]>;
```

Стоит упомянуть о следующем правиле: когда **e.ML** и **e.MR** составляют выражение, количество термов

в них обеих должно быть одинаковым. Однако, в процессе формирования нового выражения (как в следующем далее случае функции **Pair**) **e.ML** и **e.MR** могут и не удовлетворять такому отношению.

Эта система обозначений реализована не как расширение базисного РЕФАЛа. Она используется как *макро-нотация*. Программы, её использующие, должны транслироваться, вручную либо автоматически, в регулярный РЕФАЛ так, чтобы, когда программист реально использует РЕФАЛ-программу, он смог видеть транслированные предложения такими, какими они действительно используются в РЕФАЛ-машине. Обоснование такого подхода кроется главным образом в том, что следует принимать во внимание отладку. В некоторых случаях нам также может захотеться работать непосредственно с представлениями мультискобок.

Полностью функцию **Chpm** можно определить, при использовании мультискобочной техники, следующим образом:

```
Chpm { e1 = <Chpm1 [. ^ e1 .]>; }
     /* change to MB format;
        set pointer at the beginning */
Chpm1 {
  [ e1 ^ '+' e2 ] = <Chpm1 [ e1'-' ^ e2 ]>;
  [ e1 ^ sX e2  ] = <Chpm1 [ e1 sX ^ e2 ]>;
  [ e1 ^ (e2)e3 ] = <Chpm1 [ e1( ^ e2)e3 ]>;
  [ e1(e2 ^ )e3 ] = <Chpm1 [ e1(e2) ^ e3 ]>;
  [. e1 ^ .] = e1;  }
```

Этот текст должен быть странслирован в регулярный РЕФАЛ. РЕФАЛ-система включает функцию **Mbprep** (мультискобочный препроцессор), написанную на РЕФАЛе. Она обнаруживает те предложения в программе, которые используют макро-нотацию, и транслирует их. Когда **Mbprep.rsl** находится в Вашей текущей директории, запустите

```
refgo mbprep  source-file out-file
```

Предложения, использующие мультискобочную макро-нотацию, гораздо легче читать, чем действительные предложения, которые они замещают. Хорошей практикой является переписывание таких предложений в терминах макро-нотации и сохранение их в окончательной программе в форме комментариев. Просто сделайте копию каждого такого предложения и добавьте звёздочку в начале каждой строки. Затем вызовите **Mbprep**.

Продемонстрируем результат на примере функции **Pair**. Базовые функции чтения РЕФАЛ-5 порождают РЕФАЛ-выражения, состоящие только из набираемых символов. Если мы хотим конвертировать символы **'('** и **')'** в таких выражениях в "настоящие" скобки РЕФАЛа, можно использовать функцию, определённую следующим образом:

```
*** Function Pair pairs symbol-parentheses
$ENTRY Pair{
*  eX = <Pair1 [. ^ eX .]>
  eX = <Pair1 (( ))( eX )> };

Pair1 {
*  [ e1 ^ '('e2 ] = <Pair1 [ e1( ^ e2 ]>;
  (e.ML( e1 ))( '('e2 )e.MR =
              <Pair1 (e.ML( e1)( ))( e2 )e.MR>;
*  [ e1(e2 ^ ')'e3 ] =
*          <Pair1 [ e1(e2) ^ e3 ]>;
```

```
  (e.ML( e1)(e2 ))( ')'e3 )e.MR =
          <Pair1 (e.ML( e1(e2) ))( e3 )e.MR>;
*  [. e1 ^ ')'e3 ] = error message;
  (( e1 ))( ')'e3 )e.MR =
    <Prout "*** ERROR: Unpaired ')' found:">
    <Prout '*** 'e1')'>;
*  [ e1 ^ s2 e3 ] = <Pair1 [ e1 s2 ^ e3 ]>;
  (e.ML( e1 ))( s2 e3 )e.MR =
          <Pair1 (e.ML( e1 s2 ))( e3 )e.MR>;
*  [. e1 ^ .]  = e1;
  (( e1 ))( ) = e1;
*  [ e1 ^ .] = error message;
  (e.ML( e1 ))( ) =
    <Prout "*** ERROR: Unbalanced '(' found:">
    <Pr-lmb e.ML>;  }

*  PRint Left MultiBracket
Pr-lmb {
  (e1)e2 = <Prout '*** 'e1'('> <Pr-lmb e2>;
     = ;  }
```

Когда программа встречает правую скобку, которая не имеет пары, предшествующее выражение уже не имеет мультискобок, так что оно просто выдаётся на печать, чтобы локализовать ошибку. Когда обработка доходит до конца выражения и имеется непарная левая скобка, просмотренная часть выражения все ещё в мультискобочном формате, поэтому вводится **`Pr-lmb`** для того, чтобы выдать её на печать.

Интересной особенностью последовательного мультискобочного формата является то, что при обработке дерева весь путь от указателя до корня дерева представлен в виде последовательности термов и потому легко доступен. Прибегнем к представлению деревьев в форме РЕФАЛ-выражений, используя следующие правила:

- каждый узел представлен символом;
- для представления поддерева, состоящего из единственного конечного узла (листа) *N* после узла указывается период: *N*'.';
- для представления нелистового поддерева, которое начинается с *N*, после *N* помещается в скобках список поддеревьев, которые начинаются с потомков *N*: $N(T_1T_2...T_n)$.

Так, дерево на *Рис. 3.1* записывается в форме:

```
T(A(B.C.)D(E.F(G.H.I.)J.)K.)
```

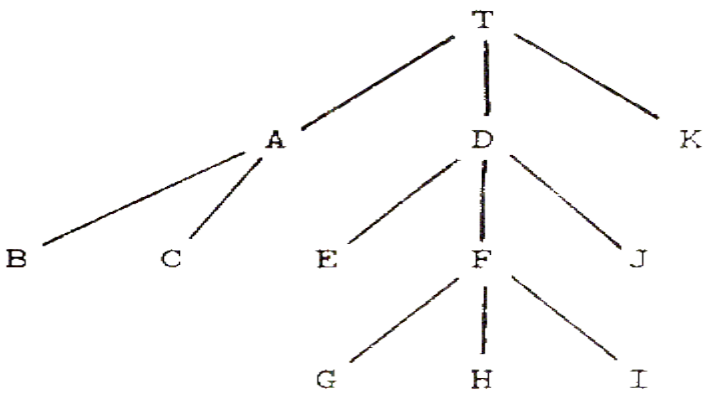


Рисунок 3.1 Дерево объектов

Если только что пройден **H**, размеченное дерево в мультискобочной макро-нотации будет иметь вид:

```
[.T(A(B.C.)D(E.F(G.H ^ .I.)J.)K.).]
```

что соответствует:

```
( (T)(A(B.C.)D)(E.F)(G.H) ) (.I.)(J.)(K.)()
```

(Заметим, что если бы этот текст был частью программы, все данные-точки стояли бы в кавычках, отличаясь таким образом от точек, используемых при квадратных скобках, которые, конечно, в кавычки не заключаются.)

Здесь можно отследить пути от текущего пункта во всех четырёх направлениях: вперёд, назад, вверх или вниз. Если восстанавливается путь от указателя (0- уровень скобок) назад, видно, что последним пройденным объектом был **H**; также можно далее увидеть на предшествующей ветви лист **G.**, являющийся братом **H**. Для того, чтобы подняться вверх по дереву, продвигаемся на один мультискобочный терм влево и видим, что предком **H** является **F**. Уровнем выше мы видим **D** как более отдалённого предка, а затем **T**. В РЕФАЛ-выражении все эти объекты находятся в одинаковой позиции: последний объект внутри терма первого уровня. Остальные объекты, такие как **A**, находятся вне пути. Их форма сохранена: взятие в мультискобки оставляет их в стороне.

Из-за этого свойства мультискобочного формата можно работать с путём от корня дерева до текущего пункта, используя открытые переменные. Предположим, например, для таких деревьев, как описано выше, мы хотим определить функцию, которая определяла бы, находится ли объект **sX** на пути к указанному узлу. Пусть форматом функции является:

```
<In-path sX e.Tree>
```

Для выполнения этой работы вводится следующее определение:

```
In-path {
  sX (e1(e.Sibl sX)e2) e3 = True;
  eZ = False; }
```

Здесь **e.Sibl** является списком уже просмотренных братьев узла **sX**.

ЗАМЕЧАНИЕ: Указанный метод является общим методом записи функций, которые используют мультискобки. Сперва описывается ситуация в естественной форме, с использованием примера или схемы, и определяется необходимая трансформация в этих терминах. Далее можно следовать по одному из двух путей; какой-либо из них, возможно, окажется более полезным, чем другой. Можно прямо выразить трансформацию в мультискобочном формате и затем использовать **mbprep**. Либо сперва транслировать трансформационную схему в мультискобочный формат, а затем записать соответствующие РЕФАЛ-предложения.

Упражнение 3.11

Иногда необходимо прекратить мультискобочную обработку и выдать выражение в том виде, который оно имеет в текущий момент. Тогда мы должны провести обратную трансформацию мультискобочного представления в регулярное РЕФАЛ-выражение, которое стоит за ним. Записать функцию **Mback**, осуществляющую это.

Упражнение 3.12

(a) Пусть дана функция **<Subtree sX>**, которая конвертирует узел **sX** в поддерево, начинающееся с корня **sX**. Предполагается, что функция **Subtree** будет использоваться рекурсивно, т.е. нелистовые узлы, возникающие в результате **Subtree**, ещё потребуют повторного применения этой функции. Определить функцию **Tree**, которая будет строить полное дерево по данному корневому узлу.

(b) Если **Subtree** требует большого времени для вычисления, может быть полезным просмотр всего существующего дерева перед применением этой функции. Если текущий узел уже перед этим возник, и его поддерево было определено, тогда оно может быть скопировано, чтобы не получать его повторно. Записать модификацию **Tree**, которая реализует этот алгоритм.

4. РАСШИРЕННЫЙ РЕФАЛ

В этой главе вводится некоторое расширение базисного РЕФАЛа, которое делает программирование более лёгким, а программы — более наглядными. Средствами расширения являются:

- where-конструкции, или условия;
- with-конструкции, или блоки;
- функции закапывания-выкапывания.

4.1. УСЛОВИЯ

Рассмотрим функцию **Pre-alph**, которая устанавливает отношение предшествования между символами в смысле обычного алфавитного порядка. В Главе 3 она была определена следующим образом:

```
Pre-alph {
*1.This relation is reflexive
  s.1 s.1 = T;
*2.If the letters are different, see whether the
*  first is before the second in the alphabet
  s.1 s.2 = <Before s.1 s.2 In <Alphabet>>; }

Before {
  s.1 s.2 In e.A s.1 e.B s.2 e.C = T;
  e.Z = F; }

Alphabet { = 'abcdefghijklmnopqrstuvwxyz'; }
```

Это определение записано средствами базисного РЕФАЛа. В полном РЕФАЛе функцию **Pre-alph** можно определить, не вводя вспомогательную функцию **Before**:

```
Pre-alph {
  s.1 s.1 = T;
  s.1 s.2, <Alphabet>: e.A s.1 e.B s.2 e.C
         = T;
  e.1 = F;  }
```

Здесь было использована *where-конструкция*, которая накладывает дополнительное условие на применимость предложения. Этим условием служит:

```
<Alphabet>: e.A s.1 e.B s.2 e.C
```

Она следует за левой частью предложения, отделяясь от него специальным *where-with знаком* РЕФАЛа. В языке РЕФАЛ-5 этот знак может быть представлен либо амперсендом &, либо запятой **,**. Он используется для выделения как where-конструкций, так и with-конструкций (блоков). Как вскоре будет видно, with-конструкцию легко отличить от where-конструкции с помощью фигурных скобок. Однако некоторые программисты, возможно, предпочтут использование запятой в качестве where-знака и амперсенда в качестве with-знака (либо наоборот). В этой книге в обоих случаях используется запятая.

Условие является сопоставляемой парой, где образцом (правым операндом) может служить любое выражение-образец, а аргументом (левым операндом) может являться любое РЕФАЛ-выражение; единственным ограничением для аргумента является то, что он должен включать только те переменные, которые имеют определённые значения на момент проверки условия (*связанные* переменные). Для

условия, которое непосредственно следует за левой частью предложения, это требование означает, что в его аргументе могут использоваться только те переменные, которые появляются в левой части предложения. Выражение-образец в правой части условия может включать как связанные переменные, так и *свободные* переменные, которые ещё не были определены и получают значения в процессе сопоставления.

В предложении могут встретиться несколько последовательных where-конструкций. Условия, введённые таким образом, будут вычисляться (проверяться) в заданной последовательности.

Пусть задано условие *E* : *P*. Оно вычисляется следующим образом. Во-первых, РЕФАЛ-машина порождает новое поле зрения, помещает *E* в это поле и заменяет переменные из *E* их значениями. Затем машина работает, как обычно, над этим полем зрения до тех пор, пока его содержимое пассивно. В процессе вычислений могут порождаться дополнительные поля зрения. Когда процесс завершается, результат сопоставляется с образцом *P* . При этом сопоставлении те переменные в *P* , которые являются уже связанными, заменяются на свои значения, в то время как свободные переменные принимают значения в процессе сопоставления. Если сопоставление является успешным, вычисляется следующее условие; если больше нет условий, имеет место замена левой части предложения на правую его часть. Если сопоставление терпит неудачу, объявляется тупиковая ситуация в предшествующем сопоставлении, (которое могло относиться как к условию, так и к левой части сопоставления с образцом). В этом случае открытые переменные в образце будут удлиняться до тех пор, пока это возможно. Если больше не остаётся переменных, подлежащих удлинению, тупиковая ситуация снова переносится на предшествующее сопоставление; если это происходит при сопоставлении в левой части, предложение неприменимо, как и в базисном РЕФАЛе, и испытывается следующее предложение. После того как проверка условия завершается успехом либо неудачей, временное поле зрения, созданное для **E**, уничтожается, и РЕФАЛ-машина возвращается к полю зрения, из которого вызывалось *E*.

Таким образом, каждый образец в последовательности условий может присваивать значения новым переменным, которые становятся связанными в последующих условиях. В конце концов все эти переменные получат значения и смогут быть использованы в правой части предложения.

В случае функции **Pre-alph** при проверке условия вычисляется значение функции **<Alphabet>**. В последующем процессе сопоставления **s.1** и **s.2** заменяются их значениями, т.е. символами, подлежащими сравнению. Сопоставление завершится успешно тогда и только тогда, когда **s.1** предшествует **s.2** в алфавите.

Интерпретация условий, по существу, занимает столько же, сколько и трансляция условий в базисный РЕФАЛ. Для каждого условия создаётся специальная вспомогательная функция, которая выполняет сопоставление, требуемое по условию. Имена этих функций образуются из имени вызывающей функции добавлением $1, $2, и т.д. Предположим, имеется вызов ***<F X>***, такой, что на следующем шаге РЕФАЛ-машины применяется одно из предложений ***F***, которое включает условие:

```
F {...
   L, E : P  =  R;
   ...
  }
```

Тогда система вставит вызов вспомогательной функции ***F***$*n* между окончанием аргумента *X* и закрывающей вычислительной скобкой:

*<**F**X<**F**$n**E**>>*

Согласно общему правилу, теперь будут вычислены активные подвыражения из **E** (если таковые

имеются) ; затем управление перейдёт к ***F***$*n*. Однако, эта функция является специальной; выполняется только сопоставительная часть шага, а именно, вычисленное *E* сопоставляется с **P**. Затем управление возвращается к ***F***. Если сопоставление было успешным, имеет место подстановка выражения **R** , в противном случае производится попытка применить следующее предложение. В любом случае вызов ***F***$*n*, который был ранее вставлен, удаляется.

Об этом механизме полезно знать, когда используется трассировщик. Отследим вычисление вызова

```
<Pre-alph 'ba'>
```

После первого шага в поле зрения появится:

```
<Pre-alph 'ba'<Pre-alph$1 <Alphabet>>>
```

Теперь **<Alphabet>** проработает с результатом:

```
<Pre-alph 'ba'<Pre-alph$1 'ab...z'>>
```

Если в этот момент есть обращение к трассировщику для печати активного выражения, мы увидим

```
<Pre-alph$1 'ab...z'>
```

как если бы **Pre-alph$1** являлась настоящей функцией. Однако, замещению результатом подлежит вызов **Pre-alph**, а не **Pre-alph$1**, так что на следующем шаге мы имеем:

```
F
```

Упражнение 4.1.

Определить функцию, которая ищет первое вхождение операции сложения-вычитания на верхнем уровне, т.е., знак **'+'** или **'-'**, и заключает предшествующее подвыражение в скобки.

Упражнение 4.2.

В некоторых сообщениях о системе РЕФАЛ-5 упоминается только целочисленная арифметика, а предикат **Compare** не встраивается. Предикаты для сравнения целых чисел могут быть определены в РЕФАЛе посредством вычитания. Определить **Less** и **Lesseq** ($\prec$ и $\preceq$) для целых чисел (см. Reference Section C , форматы).

4.2. БЛОКИ

With-конструкции позволяют использовать *блоки* (напомним, что блоком является список предложений в фигурных скобках) прямо в рамках функционального определения без введения вспомогательной функции для этой цели.

В качестве примера with-конструкции, напомним функцию упорядочения пар из Главы 3 :

```
Order {
 (e.1)e.2 = <Order1 <Pre (e.1)(e.2)> (e.1)e.2>;}

Order1 {
  T (e.1)e.2 = e.1 e.2;
  F (e.1)e.2 = e.2 e.1;
       };
```

Вспомогательная функция становится ненужной, если используется полный РЕФАЛ:

```
Order {
  (e.1)e.2, <Pre (e.1)(e.2)>:
          { T = (e.1)(e.2);
            F = (e.2)(e.1);
          }; }
```

With-конструкция начинается таким же образом, как и where-конструкция: с where-with-знака, затем следует общее РЕФАЛ-выражение, использующее только связанные переменные (аргумент), затем двоеточие - знак операции сопоставления. Однако вместо образца в with-конструкции используется блок. Фигурные скобки, объединяющие группу предложений в блок, имеют двоякое значение.

Во-первых, они дают возможность сопоставлять аргумент с несколькими образцами. В нашем примере **<Pre (e.1)(e.2)>** вычисляется однажды, после чего РЕФАЛ-машина пытается сопоставить его с последовательными левыми частями в блоке, замещение выполняется как обычно.

Второй функцией фигурных скобок в with-конструкции является завершение процесса сопоставления в предшествующей левой части и в условиях. Когда управление переходит к левой фигурной скобке, нет пути назад к удлинению е-переменных. Какие бы значения ни были присвоены переменным вне блока, они должны такими и остаться. В случае, когда ни одно из предложений неприменимо, последует аварийное завершение, как если бы это было функциональное определение.

На самом деле блок является функциональным определением, но эта функция не имеет присвоенного имени и вызывается по *аргументу*, (по выражению, следующему за where-with-знаком) в качестве ее аргумента. Можно видеть, что блок в определении функции **Order** в точности тот же, что и во вспомогательной функции **Order1**. With-констукция позволяет определить безымянную функцию именно в том месте, где она нужна. Определение функции **Order** в базисном РЕФАЛе можно рассматривать как семантику with-конструкции во втором определении.

Рассмотрим следующее определение:

```
F {
  e.1'+'(e.2)e.3, <P e.2>: T = (e.2);
  e.1 = <Prout 'No such term'>;
 };
```

Эта функция находит первый заключенный в скобки терм после **'+'**, такой, что выражение внутри удовлетворяет предикату **P** . Если **P** определён как:

```
P { s.1'/'s.2 = T;
  e.X = F;  }
```

то вычисление функции:

```
<F  A-B+(C*D)+(C/D)>
```

приводит к результату **(C/D)** . Если заключить условие и правую часть в фигурные скобки, так, что они образуют блок-предложение:

```
F { e.1'+'(e.2)e.3>, <P e.2>: { T = (e.2) };
   e.1 = <Prout 'No such term'>;  }
```

тогда алгоритм следует читать следующим образом: найти первый заключенный в скобки терм,

следующий за **'+'**, а затем проверить, удовлетворяет ли заключённое в скобки выражение предикату **P**. Если теперь вычисляется тот же вызов с измененным определением, результатом будет аварийный останов `Распознавание невозможно'. На самом деле первым термом после **'+'** является **(C*D)**, и **P** не удовлетворяется. Не будет попытки удлинять **e.1**: можно только *войти* в фигурные скобки, окружающие блок, но никогда нельзя выйти из них.

Вспомним функцию **Sub-a-z**, которая выделяет подстроку, начинающуюся на **'a'** и оканчивающуюся на **'z'**. В Главе 2, чтобы эффективно задать её, потребовалось ввести вспомогательную функцию. Теперь эта функция может быть включена как блок:

```
Sub-a-z {
  e.1'a'e.2, e.2 :
        { e.3'z'e.4 = (e.1)'a'e.3'z'(e.4);
           e.3 = <Prout 'No substring a-z'>;
        };
  e.1 = <Prout "No 'a' found">;
       }
```

После того как обнаружено первое **'a'**, производится вход в блок, и если после этого нет ни одного **'z'**, **e.1** не будет удлиняться; вместо этого будет сформулирован отрицательный результат.

Блоки, как и условия, реализуются через вызовы вспомогательных $-функций, которые вставляются между аргументом главной (вызывающей) функции и правой функциональной скобкой. Вспомогательные функции для блоков даже более близки к обычным функциям, чем аналогичные функции для условий. Подобно обычным функциям, они вычисляются посредством попытки сопоставить аргумент с последовательными предложениями в блоке; после успешного сопоставления соответствующая правая часть используется для замещения. Однако вызов, который замещается результатом, вовсе не вызов $-функции, а вызов главной функции, определение которой включало эту with-конструкцию.

Отследим вызов функции **Order**, рассмотренной выше:

```
1.   <Order ('abc')'de'>
2.   <Order ('abc')'de'<Order$1 <Pre ('abc')('de')>>>
```

После вычисления **Pre**, которое производится за четыре шага, имеем:

```
6.   <Order ('abc')'de'<Order$1 T>>
7.   ('abc')('de')
```

Упражнение 4.3

Переопределить следующую функцию, чтобы сделать ее определение более эффективным :

```
s123 {
  e.A 1 e.B 2 e.C 3 e.D = T;
  e.A = F;  }
```

Упражнение 4.4

Инвертированная пара в последовательности чисел есть такая пара смежных чисел, что предшествующее число больше последующего.

(a) Написать программу, которая отыскивает первую инвертированную пару в данной числовой

последовательности, для которой сумма членов пары превосходит 100.

(b) Записать программу, которая определяется, является ли сумма членов первой инвертированной пары превосходящей число 100.

Использовать функцию **Less** из Упражнения 4.2.

4.3. ФУНКЦИИ ЗАКАПЫВАНИЯ-ВЫКАПЫВАНИЯ

РЕФАЛ является строго функциональным языком. Но в качестве уступки более традиционным способам программирования, в нем ещё остаётся ряд встроенных функций, которые позволяют пользователю употреблять присваивания. Это не рекомендуется делать и, на самом деле, в этом нет нужды — использовать присваивания для манипуляции объектами. Но иногда желательно поддерживать некоторые глобальные параметры или таблицы, не передавая их как часть аргумента от одного функционального вызова к другому. Тогда можно "закопать" такие параметры под присвоенными именами, активизируя выражения следующего вида:

(*) **<Br *N* '=' *E* >**

Когда такой параметр потребуется, он "выкапывается" посредством

(**) **<Dg *N*>**

Здесь ***N*** означает строку символов, которая не включает символ **'='**, а **E** — любое выражение.

Когда вычисляется **(*)**, оно исчезает из поля зрения, но на его месте остаётся выражение **E** в реальной связной структуре, представляющей поле зрения. Размещение его связывается со строкой ***N*** и сохраняется в таблице. Поэтому вычисление **Br** занимает постоянный малый квант времени, не зависящий от величины **E**. Когда вычисляется **(**)** , оно снова замещается закопанным выражением, без просмотра последнего, только путём восстановления связей с его концами. Имеется также функция копирования **Cp**, которая имеет такой же формат и значение, как и **Dg** , но с её помощью закапываемое выражение копируется, а не извлекается. При многократном использовании **Br** и **Dg** порождают стэк закопанных значений для каждого имени. Функция **Br** закапывает каждое следующее значение для данного имени, не затрагивая предыдущих значений. Функция **Dg** извлекает последнее закопанное значение. Таким образом, эти функции работают как *стэки* значений для каждого используемого имени. Однако, если стэк для данного имени пуст, результатом **Dg** является пустое значение, т.е. он не является неопределённым, как это должно было бы быть для обычного стэка. Это отклонение от строгой семантики стэков оказалось практически полезным.

Предположим, создаётся программа, которая имеет дело с некоторыми объектами и время от времени порождает новые объекты того же вида. Им присваиваются последовательные номера, начиная с 1. Функция, которая порождает новые объекты, должна иметь доступ к первому свободному индексу. Он может быть сохранен, будучи закопан под некоторым именем, скажем, **'fr-ind'** . Тогда при запуске программы, например, как фрагмента правой части **Go**, выполняется:

```
<Br 'fr-ind=' 1>
```

Как только понадобится породить новый индекс, используется:

```
<Next 'fr-ind'>
```

в качестве индекса. Функция **Next** получает число, закопанное под её аргументом-именем, и

увеличивает его на 1:

```
Next {
  e.Name, <Dg e.Name>: e.Value  =
          e.Value
          <Br e.Name'=' <+ e.Value 1>>; }
```

В Разделе 2.1 была написана программа, транслирующая строки итальянских слов в строки слов английских. Функция **Table** была применена для хранения трансляционной таблицы (словаря):

```
* Italian-English dictionary table
Table { = (('cane') 'dog')
          (('gatto') 'cat')
          (('cavallo') 'horse')
          (('rana') 'frog')
          (('porco') 'pig')   }
```

Вместо хранения этой таблицы в качестве правой части предложения, можно было бы хранить ее как выражение, закапываемое под одним и тем же именем, скажем, **'dict'**.

Выражения, закапываемые при помощи **Br**, хранятся в форме двухсвязных списков, как и все выражения в поле зрения. В отличие от них, правые части предложений сохраняются в форме кода в языке RASL. Последняя форма гораздо более компактна, чем первая. Однако, закопанное выражение готово к применению, в то время как правая часть предложения подобна той, которой используется для **Table**, будет прочитана РЕФАЛ-машиной посимвольно в процессе создания соответствующего двухсвязного списка в поле зрения. Поэтому, когда существенны пространственные ограничения, предпочтительнее сохранять таблицу в правой части предложения; но если более важно время выполнения, таблицу следовало бы закапывать.

Модифицируем наш итало-английский транслятор так, чтобы таблица сохранялась за счёт закапывания. Первым делом следует модифицировать функцию **Go** таким образом, чтобы производить закапывание в начале работы:

```
* This program translates from Italian into
* English and keeps the dictionary buried.
$ENTRY Go { =  <Br 'dict='
          (('cane') 'dog')
          (('gatto') 'cat')
          (('cavallo') 'horse')
          (('rana') 'frog')
          (('porco') 'pig')
                >
                <Job <Card>>    }
```

Теперь следует соответствующим образом изменить использование таблицы. Самый простой способ сделать это - переопределить только одну функцию, **Table**, так, чтобы она копировала словарную таблицу:

```
Table { = <Cp 'dict'>; }
```

Однако, этот способ ничем не лучше хранения таблицы в предложении, так как перед использованием таблицы РЕФАЛ-машина должна будет сделать вторую её копию — с неизбежными потерями по времени и памяти. Для того, чтобы использовать закопанную таблицу эффективно, следует определить процесс так, чтобы именно та копия, которая была закопана, и использовалась в дальнейшем, а затем

снова закапывалась без изменений.

Таким образом, вызов **<Table>**, который встречается в **Trans-line**, заменяется на **<Dg 'dict'>** :

```
* Translate one line
Trans-line {
  ' 'e.X = <Trans-line e.X>;
  e.Word' 'e.Rest =
        <Trans (e.Word) <Dg 'dict'>>' '
        <Trans-line e.Rest>;
   =  ;
  e.Word = <Trans (e.Word) <Dg'dict'>>' ';
         }
```

Полная словарная таблица будет теперь аргументом **Trans**. Модифицируем **Trans** так, чтобы она опять закапывала таблицу после использования:

```
* Translate Italian word into English by table
Trans {
   (e.It) e.1 ((e.It)e.Eng) e.2 =
        e.Eng <Br 'dict=' e.1 ((e.It)e.Eng) e.2>;
   (e.It) e.1 =
        ***<Br 'dict=' e.1>;
              /* the word not in table */
     }
```

Одним из преимуществ сохранения закопанных таблиц является то, что они могут быть легко обновлены во время выполнения программы.

Упражнение 4.5

Модифицировать итало-английский транслятор таким образом, чтобы словарь постоянно сохранялся на диске как файл **DICT**. В начале работы программа предлагает пользователю сделать (возможные) дополнения к словарной таблице. Пользователь вводит дополнения, программа обновляет файл и затем работает, как описано выше.

5. РАЗРАБОТКА ПРОГРАММ

В этой главе приводится ряд примеров разработки программ на РЕФАЛе.

5.1. МИССИОНЕРЫ И КАННИБАЛЫ

РЕФАЛ является таким языком, который ориентирован на конструирование определённых *объектов*, представленных РЕФАЛ-выражениями, и на манипулирование ими. Структура управления РЕФАЛ-программы весьма проста и незамысловата: сопоставление в образцом и подстановка. Именно структура объектов, которыми манипулирует программа, несёт бремя хранения всей полезной информации. Удачное программирование на РЕФАЛе — это прежде всего удачный выбор основных структур данных. Представление проблемы, которая должна быть решена, в терминах РЕФАЛ-выражений является первой и важнейшей частью программирования на РЕФАЛе. Это представление должно, во-первых, быть наглядным для человека, во-вторых, допускать выражение основных операций над данными в терминах трансформации объектов в соответствии с некоторыми простыми образцами. Если эти два требования удовлетворяются, у вас достаточно хорошие шансы написать РЕФАЛ-программу, которая будет компактной, читаемой, и в то же время легко отлаживаемой.

Рассмотрим хорошо известную задачу "Миссионеры и каннибалы". Три миссионера и три каннибала приходят на берег реки и видят лодку. Они хотят переплыть реку. Лодка, однако, не может вместить более двух человек. Дальнейшим ограничением является следующее. Ни в один момент времени число каннибалов на любом берегу реки (включая причалившую лодку) не должно превосходить число миссионеров (так как в этом случае миссионеров одолеют и съедят). Каким образом можно переплыть реку (если это вообще возможно)?

Общий метод решения задачи таков:

1. Определим множество возможных ситуаций, или *состояний*, подразумеваемых в задаче (*пространство состояний*). Одно из состояний должно быть определено как заключительное. И должен существовать критерий, согласно которому состояние является допустимым решением задачи (достижение цели).
2. Определим множество допустимых *перемещений*, которые можно совершать в направлении решения проблемы. Перемещение является операцией, которая трансформирует одно (текущее) состояние в другое (следующее) состояние. Как правило, не всякое перемещение допустимо в каждом состоянии, но для каждого состояния будет определено подмножество *допустимых* перемещений.
3. Эти два множества определяют дерево достижимых состояний, корнем которого является начальное состояние. Теперь следует произвести поиск по этому графу, чтобы отыскать путь, который ведёт к решению (если он вообще существует).

Как будут представлены ситуации в нашей задаче? Подвыражения в РЕФАЛе представляют подсистемы в полной системе. Наша система очевидным образом включает две подсистемы людей: находящихся на левом и на правом берегу реки. Таким образом, можно использовать образец:

```
(e.Left-bank)(e.Right-bank)
```

Простейшие объекты могут быть представлены РЕФАЛ-символами. Будем использовать для обозначения миссионера `'M'`, а для каннибала — `'C'`. Ситуация, когда на левом берегу три миссионера и один каннибал, а на правом берегу два каннибала, может быть тогда представлена следующим образом:

```
('MMMC')('CC')
```

Так как порядок, в котором люди перечисляются, не играет роли, три миссионера и каннибал могут быть также представлены как **'CMMM'**, **'MCMM'**, и т.д. Нужно ли оставлять такую свободу в представлении или лучше фиксировать определённый порядок? Последний вариант, очевидно, предпочтительнее. Следует продумать заранее использование РЕФАЛ-представлений в образцах. При произвольном порядке символов для определения, находятся ли миссионеры на данном берегу, будет использоваться образец **e1'M'e2**. Этот образец включает открытую e-переменную, которая означает просмотр выражения. Если принять соглашение о перечислении миссионеров перед каннибалами, то же самое может быть определено по более простому образцу **'M'e1**. Чтобы определить, имеется ли в группе хотя бы один каннибал, применяется **e1'C'**, а для определения, присутствуют ли в ней оба вида людей, применяется **'M'e1'C'**. Здесь используется тот факт, что люди делятся только на два вида.

Теперь следует позаботиться о лодке. Один из способов - включить её в качестве символа **'B'** в описание соответствующего берега, скажем, так:

```
('MMMC')('BCC')
```

Однако, это будет не слушком удобно. Чтобы абстрагироваться от размещения лодки, будут применяться два образца:

```
('B'e1)(e2) = (e1)(e2);
(e1)('B'e2) = (e1)(e2);
```

Лодка не будет приравниваться к людям; её расположение является независимым измерением. Более хорошая идея — добавлять один символ (**'L'** или **'R'**) - для того, чтобы отмечать, находится ли лодка на левом или на правом берегу. Допустим, что вся компания приходит на левый берег реки. Тогда представлением начальной ситуации (состояния) служит:

```
L(e.Left-bank)(e.Right-bank)
```

ЗАМЕЧАНИЕ: Для представления миссионеров и каннибалов были использованы символы (обратите внимание на кавычки), а для положения лодки - идентификаторы **L** и **R**. Это не имеет большого значения. Обычно программа выглядит лучше, если элементы, которые могут образовать предложение, представлены символами, в то время как символы, используемые отдельно, представлены идентификаторами.

Обратимся теперь ко множеству возможных перемещений. Перемещением является переплывание реки на лодке. Так как лодка может вместить не более двух человек и не может переплывать сама, то имеется пять возможных перемещений, которые будут представлены последовательными номерами в следующей таблице:

Перемещение	В лодку садятся:
1	два миссионера
2	два каннибала
3	миссионер и каннибал
4	миссионер
5	каннибал

Не все перемещения физически возможны в каждом состоянии. Например, если лодка находится на левом берегу и только один миссионер на этом берегу, тогда перемещение 1 невозможно. Но даже если перемещение возможно, следует проверить, является ли оно допустимым, т.е. не приводит ли к

убийству. Так как перемещение является трансформацией текущего состояния, то оно определяется как:

```
* Move
* <Move s.Move e.State> == s.New-State
Move { sM eS = <Admiss <Try sM eS>>; }
```

Функция **Try** будет трансформировать состояние. Если перемещение невозможно в данном состоянии, то новое, фиктивное, состояние будет представляться символом **Imposs**. **Admiss** будет проверять, что результирующее состояние удовлетворяет необходимым требованиям. Если это так, функция оставит состояние неизменным. Если же требования не удовлетворяются, она заменит состояние на **Imposs**. Вот как выглядит простое и прямолинейное определение **Try**, которое использует средства сопоставления с образцом языка РЕФАЛ и изобилует комментариями:

```
* Try a move
Try {
* Boat on left bank
* MM crossing
  1 L('MM'e1)(e2) = R(e1)('MM'e2);
* CC crossing
  2 L(e1'CC')(e2) = R(e1)(e2'CC');
* MC crossing
  3 L(e1'MC'e2)(e3) = R(e1 e2)('M'e3'C');
* M crossing
  4 L('M'e1)(e2) = R(e1)('M'e2);
* C crossing
  5 L(e1'C')(e2) = R(e1)(e2'C');
* Boat on right bank
* MM crossing
  1 R(e1)('MM'e2) = L('MM'e1)(e2);
* CC crossing
  2 R(e1)(e2'CC') = L(e1'CC')(e2);
* MC crossing
  3 R(e1)('M'e2'C') = L('M'e1'C')(e2);
* M crossing
  4 R(e1)('M'e2) = L('M'e1)(e2);
* C crossing
  5 R(e1)(e2'C') = L(e1'C')(e2);
* Otherwise move impossible
  s.Mv eS = Imposs;
     }
```

Функция приемлемости проверяет, что условие "непоедания" выполняется на каждом берегу:

```
* Admissibility of the move
Admiss {
  s.Side(eL)(eR),
         <No-eat eL>: T, <No-eat eR>: T =
               s.Side(eL)(eR);
  eS = Imposs;
       }

* No eating missionaries
No-eat {
*1. Both missionaries and cannibals are present
  'M'e1'C' = <Compare e1>;
```

```
  * Otherwise OK
    e1 = T; }

  * Both M and C on the bank.Compare the numbers.
  Compare {
    'C'e1 = F;
    'M'e1'C' = <Compare e1>;
    e1 = T; }
```

Дерево достижимых состояний содержит узлы-состояния и ребра-перемещения. Перемещение и состояние, в которое оно ведёт, могут быть легко объединены в единый элемент, который будет называться *связью*:

```
(s.Move'='e.State)
```

где знак равенства вставлен просто для удобочитаемости. Чтобы подогнать начальное состояние под этот формат, будем рассматривать его как следующее из фиктивного предшествующего перемещения со специальным кодом . Дерево состояний начинается следующим образом:

$$(0 = S_{init})(1 = S_1)(1 = S_{11})...$$
$$(2 = S_{12})...$$
$$...$$
$$(5 = S_{15})...$$
$$(2 = S_2)(1 = S_{21})...$$
$$(2 = S_{22})...$$
$$...$$

Примем в качестве метода поиска по дереву метод "внутри-слева" и простой фиксированный порядок опробования перемещений: 1, 2, и т.д. до 5. Тогда на самом деле нет надобности работать со всей структурой дерева. Можно хранить только одну цепочку связей, которая образует путь из корня в текущий узел.

Назовём поисковую функцию **Search**. Она вызывается в начале программы с начальной цепочкой в качестве аргумента:

```
$ENTRY Go { =
       <Search (0'='L('MMMCCC')())>; };
```

Теперь желательно рассмотреть все случаи, которые могут встретиться в ходе поиска.

Задача считается решённой, когда состояние в последней связи является целевым. Имеем предложение:

```
e1 (s.M'='R()('MMMCCC')) =
      <Path e1 (s.M'='R()('MMMCCC'))>;
```

В этом случае остаётся только подходящим образом напечатать решение, что является обязанностью функции **Path**.

Это предложение будет испытываться первым. Пока оно не срабатывает, продолжается поиск и удлинение нашего пути за счёт добавления новой связи, возникающей в результате перемещения 1 :

```
e1 (sM'='eS) =
    <Search e1 (sM'='eS) (1'='<Move 1 eS>)>;
```

Теперь, если этот путь записать, видно, что последним состоянием в пути может оказаться **Imposs**. Поэтому, перед удлинением пути следует проверять, является ли последнее состояние допустимым или состоянием **Imposs**, и в последнем случае предпринять соответствующее действие. Как всегда в РЕФАЛе, предложение (или предложения) для специального случая, когда **eS** является **Imposs**, должно предшествовать предложению(-ям) для общего случая.

Если результатом последнего перемещения является **Imposs**, испытывается следующее перемещение, т.е., последняя связь замещается следующим образом:

```
e1 (s.Mp'='eS)(s.M'='Imposs),
                  <+ s.M 1>: s.Mn =
        <Search e1 (s.Mp'='eS)
                   (s.Mn'='<Move s.Mn eS>)>;
```

Это подразумевает, однако, что **s.M** меньше, чем 5. Если оно совпадает с 5, то это означает, что в нашем распоряжении больше нет перемещений, и следует совершить *возврат*, т.е., удалить последнюю связь и предпринять следующее (возможное) перемещение в предыдущей связи. Для того, чтобы совершить следующее перемещение в предыдущей связи в соответствии с общим правилом, состояние в этой связи заменяется на **Imposs**:

```
e1 (s.Mp'='eS) (5'='Imposs) =
          <Search e1 (s.Mp'='Imposs)>;
```

По написанному РЕФАЛ-предложению, т.е. правилу замещения, можно произвести анализ, чтобы увидеть, каковы возможные исключения из него. Мы уже использовали этот метод выше и должны применить его опять. Если предшествующая связь является начальной связью, т.е. **s.Mp** является равным 0, то нет возможности для возврата и следует объявить, что задача не имеет решения:

```
(0'='eS) (5'='Imposs) =
     <Prout 'The problem has no solution'>;
```

Имеется еще одна ситуация, подлежащая рассмотрению. Когда достигается состояние, которое уже было пройдено прежде, т.е. может быть обнаружено в одной из связей построенного пути, нежелательно просматривать его продолжения, поскольку они уже были просмотрены в качестве продолжений предыдущего вхождения этого состояния. Таким образом, повторяющееся состояние замещается на **Imposs**:

```
e1 (s.Mp'='eS) e2 (s.M'='eS) =
   <Search e1 (s.Mp'='eS) e2 (s.M'='Imposs)>;
```

Теперь следует собрать все шесть предложений в правильном порядке. Совместно с другими функциональными определениями (предоставляем читателю изобразить, как работает функция **Path**), образуем следующую программу:

```
* Missionaries and Cannibals
$ENTRY Go { =
    <Search (0'='L('MMMCCC')())>; };

Search {
*1. The goal found
 e1 (s.M'='R()('MMMCCC')) =
```

```
          <Path e1 (s.M'='R()('MMMCCC'))>;
*2. Impossible state. No backup. No solution.
 (0'='eS) (5'='Imposs) =
       <Prout 'The problem has no solution'>;
*3. Impossible state. No next move. Back up.
 e1 (s.Mp'='eS) (5'='Imposs) =
              <Search e1 (s.Mp'='Imposs)>;
*4. Impossible state. Do next move.
 e1 (s.Mp'='eS)(s.M'='Imposs),
                   <+ s.M 1>: s.Mn =
         <Search e1 (s.Mp'='eS)
                   (s.Mn'='<Move s.Mn eS>)>;
*5. Repeated state. Equate to impossible.
 e1 (s.Mp'='eS) e2 (s.M'='eS) =
     <Search e1 (s.Mp'='eS) e2 (s.M'='Imposs)>;
*6. Down the tree.
 e1 (s.M'='eS) =
      <Search e1 (s.M'='eS) (1'='<Move 1 eS>)>;
     }

* Move
Move { eS = <Admiss <Try eS>>; }

* Try a move
Try {
* Boat on left bank
* MM crossing
  1 L('MM'e1)(e2) = R(e1)('MM'e2);
* CC crossing
  2 L(e1'CC')(e2) = R(e1)(e2'CC');
* MC crossing
  3 L(e1'MC'e2)(e3) = R(e1 e2)('M'e3'C');
* M crossing
  4 L('M'e1)(e2) = R(e1)('M'e2);
* C crossing
  5 L(e1'C')(e2) = R(e1)(e2'C');
* Boat on right bank
* MM crossing
  1 R(e1)('MM'e2) = L('MM'e1)(e2);
* CC crossing
  2 R(e1)(e2'CC') = L(e1'CC')(e2);
* MC crossing
  3 R(e1)('M'e2'C') = L('M'e1'C')(e2);
* M crossing
  4 R(e1)('M'e2) = L('M'e1)(e2);
* C crossing
  5 R(e1)(e2'C') = L(e1'C')(e2);
* Otherwise move impossible
  s.M eS = Imposs;
     }

* Admissibility of the move
Admiss {
 s.Side(eL)(eR), <Noeat eL>:T, <Noeat eR>:T =
               s.Side(eL)(eR);
```

```
 eS = Imposs;
      }

* No eating missionaries
Noeat {
*1. Both missionaries and cannibals are present
 'M'e1'C'= <Compare e1>;
* Otherwise OK
  e1 = T; }

* Both M and C on the bank.Compare the numbers.
Compare {
  'C'e1 = F;
  'M'e1'C'= <Compare e1>;
  e1 = T; }

* Print the path leading to the goal
Path {
  (0'='eS) e2 =
   <Prout 'The initial state:'eS>
   <Path e2>;
  (s.M'='s.Side eS) e2,
       <Look-up s.M In <Table>>: e.Who =
       <Prout e.Who ' crossing to  's.Side
             ' bank:  state ' s.Side eS>
       <Path e2>;
   = ;
     }

Look-up { sM In e1 sM(e.Who) e2 = e.Who }
Table { =
    1('MM') 2('CC') 3('MC') 4('M ') 5('C ') }
```

Назовём файл этой программы **mmmccc.ref**. Транслируем его посредством **refc** и запускаем:

```
refgo mmmccc -nt
```

(указанные флажки приводят к выдаче на печать номера шага РЕФАЛ-машины и затраченного на шаг времени). Вывод имеет вид:

```
The initial state:  L (MMMCCC)()
CC crossing to  R  bank:  state R (MMMC)(CC)
C  crossing to  L  bank:  state L (MMMCC)(C)
CC crossing to  R  bank:  state R (MMM)(CCC)
C  crossing to  L  bank:  state L (MMMC)(CC)
MM crossing to  R  bank:  state R (MC)(MMCC)
MC crossing to  L  bank:  state L (MMCC)(MC)
MM crossing to  R  bank:  state R (CC)(MMMC)
C  crossing to  L  bank:  state L (CCC)(MMM)
CC crossing to  R  bank:  state R (C)(MMMCC)
M  crossing to  L  bank:  state L (MC)(MMCC)
MC crossing to  R  bank:  state R ()(MMMCCC)
```

Упражнение 5.1 [*]

Усилия, которые нужно затратить на поиск решения (как и само решение) зависят от порядка, в котором испытываются возможные перемещения. До этого был использован один и тот же порядок перемещений, независимо от того, на каком берегу находится лодка. Однако, эти порядки могут отличаться. Модифицировать программу, изменяя порядок перемещений таким образом, чтобы она выполнялась быстрее или медленнее, чем данная программа.

Сноска:

1. К упражнениям в этой главе ответы не приводятся.

5.2. АЛГОРИТМЫ СОРТИРОВКИ

Пусть задан предикат **`Order`**, который вводит отношение (полного) порядка на некотором множестве РЕФАЛ-термов; так, **`<Order t1 t2>`** принимает значение **T**, если пара **`t1 t2`** упорядочена должным образом, и значение **`F`** в противном случае. Отношение порядка рефлексивно, антисимметрично и транзитивно.

Имея такое отношение, можно задаться целью упорядочить данный список термов таким образом, чтобы в любом случае, когда **`t1`** предшествует **`t2`** в списке, **`<Order t1 t2>`** являлся истинным. В программировании такая процедура упорядочивания обычно называется *сортировкой*. Термы, подлежащие упорядочиванию, могут, например, быть заключёнными в скобки строками, для которых в качестве **`Order`** выступает лексикографическое отношение, либо числами с **`Order`**, которое является отношением "меньше или равно".

Наиболее очевидным способом упорядочивания последовательности термов является перестановка соседних термов до тех пор, пока они не удовлетворят отношению порядка:

```
* Definition of sorting
Sort {
  e1 tX tY e2, <Order tX tY>: F =
              <Sort e1 tY tX e2>;
  e1 = e1; }
```

Эта функция является хорошим пояснением к проблеме, однако в качестве ее решения чрезвычайно неэффективна. В самом деле, предположим, выполнен шаг РЕФАЛ-машины, при котором поменялись местами **`tX`** и **`tY`**. Теперь все соседние пары в **`e1 tY`**, за исключением, быть может, последней (образованной за счёт последнего терма в **`e1`** и **`tY`**), должны удовлетворять отношению порядка. Тем не менее, на следующем шаге РЕФАЛ-машина станет проверять все эти пары перед тем, как она дойдёт до следующей неупорядоченной пары.

Улучшение напрашивается само собой: заключить уже готовую часть в круглые скобки и сравнивать последний терм готовой (упорядоченной) части с первым термом ещё неупорядочивавшейся части:

```
Sort {
   =  ;
  sX e2 = <Sort1 (sX) e2>; }

Sort1 {
```

```
  (e1 tX) tY e2, <Order tX tY>:
              { T = <Sort1 (e1 tX tY) e2>;
                F = <Sort1 (e1) tY tX e2>;
              };
  (e1) = e1; }
```

Этот алгоритм значительно лучше, но его можно ещё улучшить. После перестановки **sX** и **sY** готовая часть **e1** укорачивается на один терм, и это будет продолжаться до тех пор, пока **sY** не окажется неупорядоченным по отношению к последнему терму **e1**. Однако, когда **sY** займёт своё место, вся первоначальная часть **e1** будет упорядочена; поэтому можно улучшить алгоритм за счёт восстановления исходного положения круглых скобок. Конечно, это не может быть проделано, если не зафиксировать каким-либо образом их положение в начале процесса. Так мы приходим к алгоритму, который известен как сортировка вставками: сохранение правильного порядка в начальной части списка и вставка каждого следующего терма в соответствующее место путём перемещения его справа налево:

```
* Insertion sort
Sort {e1 = <Sort1 ()e1>; };

Sort1 {
  (e1) tY e2 =  <Sort1 (<Insert e1 tY>) e2>;
  (e1) = e1;  }

* <Insert e1 tY> assumes that e1 is ordered
* and inserts tY into it
Insert {
  e1 tX tY, <Order tX tY>:
          {T = e1 tX tY;
           F = <Insert e1 tY> tX;
          };
  e1 = e1;
      };
```

При использовании этого алгоритма среднее количество шагов, необходимых для того, чтобы отсортировать ***n*** термов, равно приблизительно $n^2 / 2$. Имеются алгоритмы, такие как сортировка слиянием, которые проделывают то же за количество шагов, пропорциональное $n \log n$. Идея сортировки слиянием состоит в делении и выборе. Для того, чтобы отсортировать список, его делят на две приблизительно равные части. Затем эти части сортируются каждая по отдельности и сливаются в один отсортированный список. Во время слияния двух упорядоченных списков ни один из них не нужно просматривать повторно. Просто сравниваются первые элементы обоих списков и выбирается наименьший (точнее, не превосходящий другого). Таким образом, общее количество операций при слиянии будет пропорционально ***n***. Для того, чтобы отсортировать половины первоначального списка, их делят на четверти, и т.д. Так как общее количество термов на всех уровнях деления всегда равно ***n***, то время, требуемое для всех слияний, на каждом уровне пропорционально ***n***. Число уровней деления, необходимое для того, чтобы прийти к одноэлементному списку, равно $\log n$. Отсюда общее время сортировки составляет $n \log n$.

В реализации этой идеи лучше идти от противного: сперва сформировать двухтермовые упорядоченные списки, затем слить их в четырехтермовые списки, и т.д.:

```
* Merge-sort
Sort { e1 = <Check <Merge <Pairs e1>>>; };
```

```
* Form ordered pairs from a list of terms
Pairs {
  t1 t2 e3, <Order t1 t2>:
           {T = (t1 t2) <Pairs e3>;
            F = (t2 t1) <Pairs e3>; };
  t1 = (t1);
      /* the odd term makes a separate list */
     = ;   }

Merge {
  (e1)(e2)e.Rest =
        (<Merge2 (e1)e2>) <Merge e.Rest>;
  (e1) = (e1);
    =  ;  }

* merge two lists
Merge2 {
  (t1 eX) t2 eY, <Order t1 t2>:
              {T = t1 <Merge2 (eX)t2 eY>;
               F = t2 <Merge2 (t1 eX)eY>; };
  (e1)e2 = e1 e2;  /* One of e1,e2 is empty */
      }

* Check whether there is one list or more
Check {
  (e1) = e1;
  e1 = <Check <Merge e1>>; }
```

Имеются и другие алгоритмы сортировки. Рассмотрим *быструю сортировку*. Выбирается первый терм в списке **t1** и производится попытка разместить его на окончательное место. С этой целью оставшийся список делится на те термы, которые должны предшествовать **t1** (назовём эту часть **e.Left**), и на те, которые должны за ним следовать(назовём часть **e.Right**). Тогда место **t1** находится между этими двумя списками. Теперь остаётся применить эту процедуру рекурсивно к двум подспискам и произвести сцепление:

```
<Sort e.Left> t1 <Sort e.Right>
```

Ниже следует программа:

```
* Quick-sort
Sort {
   = ;
  t1 e2, <Partit ()t1()e2>: (eL)t1(eR)
        = <Sort eL> t1 <Sort eR>;

* Partition list e.List by element sM.
* <Partit (e.Left)sM(e.Right)e.Remaining-list>
*       == (e.Left1)sM(e.Right1)
Partit {
  (eL)sM(eR) = (eL)sM(eR)
  (eL)sM(eR) sX e2, <Order sX sM>:
            {T = <Partit (eL sX)sM(eR) e2>;
             F = <Partit (eL)sM(eR sX) e2>;
            };
```

```
    }
```

Для разбиения на части требуется время, пропорциональное длине списка. Суммарное время для разбиений на одном рекурсивном уровне приблизительно равно ***n***. Так как **`e.Left`** и **`e.Right`** являются, в среднем, одинаковыми по длине, потребуется приблизительно **log *n*** уровней рекурсии. Отсюда полное время сортировки пропорционально ***n* log *n***.

5.3. ПУТИ НА ГРАФЕ

Ориентированный граф называется *простым*, если в нем существует не более чем одно ребро с данными начальной и конечной вершинами. *Путь* в таком графе определяется как последовательность вершин, в которой каждая следующая вершина достижима по ребру из предыдущей. Путь является *ациклическим* , если ни одна из его вершин не повторяется. Мы хотим определить функцию, которая для данного графа ***G*** и некоторой его вершины ***V*** будет перечислять все ациклические пути в ***G***, начинающиеся с ***V***.

Как всегда, первым, что следует выполнить, является представление объектов. Представим граф в формате

```
s.Vertex(e.List-of-next) ...
```

где **`e.List-of-next`** содержит все те вершины, которые могут быть достигнуты из **`s.Vertex`** по ребру (может быть, пустому). Например, граф на *Рис. 5.1* представлен следующим образом:

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

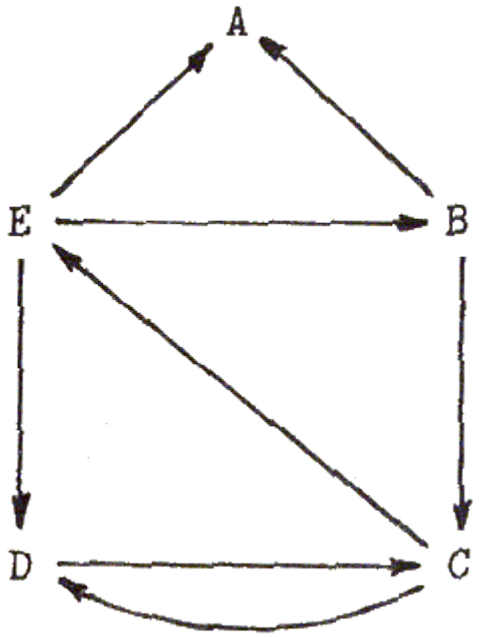


Рисунок 5.1 Ориентированный граф

Самым кратким решением проблемы является:

```
Paths {
  eP(e1 sV e2)(e3 sV(e.Next)e4) =
       <Paths eP sV (e.Next)(e3 e4)>
       <Paths eP (e2)(e3 sV(e.Next)e4)>;
  eP(e1)(e2) = (eP);
      }
```

где начальный вызов должен иметь вид:

```
<Paths (V)(G)>
```

Для графа на *Рис. 5.1* и начальной вершины **E** наша функция выдаёт результат:

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

Поясним, как мы приходим к следующему решению. Как это часто случается, прямое рекурсивное определение допустимо не только для сформулированной задачи, но и для ее обобщения. В нашем случае производится два обобщения: во-первых, вместо единственной начальной вершины ***V*** рассматривается список вершин V^{list}, и требуется найти множество путей, которые могут начинаться с любой вершины, принадлежащей V^{list}. Во-вторых, допускается произвольное начало ***P*** для вычисляемых путей. Таким образом, вызов

```
<Paths P(V^list)(G)>
```

должен приводить к получению множества путей, каждый из которых начинается с ***P***, затем проходит через некоторую вершину из V^{list} и продолжается по графу ***G***.

Идея алгоритма состоит в удалении из графа записей о тех вершинах, которые уже включены в какой-либо путь. При этом способе гарантируется, что в конструируемых путях не будет повторных вхождений вершин. В левой части предложения 1 находится образец, отыскивающий первую вершину **sV** , для которой ещё возможно продолжение, так как имеется соответствующая запись в графе. Тогда искомое множество будет состоять из двух подмножеств, для которых записаны два параллельных рекурсивных вызова **`Paths`**: в первом **`sV`** добавляется к **`eP`**, **`e.Next`** становится множеством V^{list} , и запись для **sV** исключается; во втором производится поиск других вершин во множестве V^{list} без изменения графа ***G***. Если возможных продолжений не осталось , выдаётся путь ***P***.

Тем не менее, с точки зрения эффективности этот алгоритм имеет изъяны, поскольку для каждого рекурсивного шага процедуры **`Paths`** создаётся дополнительная копия графа (или того, что от него осталось). Этот недостаток исправлен в следующей программе:

```
$ENTRY Go { =
      <Prout 'Type in the graph'>
      <Br 'graph='<Input>>
      <Prout 'Initial vertices, please'>
      <Prout <Paths (<Input>)>>; }
$EXTRN Input;

Paths {
  ePs1eQ(s1 e2) = <Paths ePs1eQ(e2)>;
  eP(s1 e2) = <Paths ePs1 (<Next s1>)>
            <Paths eP(e2)>;
  eP() = (eP); }

Next {
  sV, <Dg 'graph'>: e1 sV(eN) e2 = eN
     <Br 'graph=' e1 sV(eN) e2>;  }
```

Граф извлекается из аргумента **`Paths`** и закапывается под именем **`'graph'`**. Каждый раз, когда нужно

построить продолжение для некоторой вершины, применяется функция **Next**, которая копирует только подходящий фрагмент описания графа. Поэтому на каждом рекурсивном шаге процедуры **Paths** нужно проверить, что вершина-кандидат не встречается в данном пути (предложение 1).

Упражнение 5.2

Как вы могли бы уже заметить, в обеих версиях **Paths** пустой путь является элементом выходного множества. Хотя это естественное следствие нашего рекурсивного определения, оно может оказаться нежелательным. Модифицировать программу (программы) таким образом, чтобы исключить пустой путь.

5.4.ТРАНСЛЯЦИЯ АРИФМЕТИЧЕСКИХ ВЫРАЖЕНИЙ

Чтобы привести один из примеров программы-транслятора на РЕФАЛе, рассмотрим трансляцию арифметических выражений в язык ассемблера для одноадресного компьютера. Элементарными операндами в арифметических выражениях будут служить переменные (представленные идентификаторами) и целые числа. Допустимы пять арифметических операций: **+** , **-** , ***** , **/** , **^** (последний символ - для возведения в степень) с обычными правилами старшинства; могут также использоваться скобки.

Одноадресная вычислительная машина имеет накопитель и память. Компьютерный код состоит из операторов **LOAD**, **STORE**, **ADD**, **SUB**, **MUL**, **DIV**, **POWER** и **MINUS**, разделяемых точками с запятой. Оператор **LOAD x**, где **x** является адресом ячейки памяти, помещает значение, хранящееся в **x**, в накопитель; **STORE x** размещает содержимое накопителя в ячейке памяти с адресом **x**. Бинарные операторы выполняются над содержимым накопителя и некоторой ячейки памяти, а результат помещается в накопитель, например, **SUB x** вычитает содержимое **x** из накопителя и помещает разность обратно в накопитель. Оператор **MINUS** изменяет знак содержимого накопителя.

Мы будем транслировать арифметические выражения в язык ассемблера для описанной машины. Это значит, что вместо действительных машинных адресов будут использоваться символические адреса. Идентификаторы переменных будут использоваться в качестве их символических адресов. Вместо размещения констант по некоторому адресу и последующего использования этого адреса будут использоваться сами константы. Результат трансляции арифметического выражения представляет собой программу, которая вычисляет его значение и помещает его в накопитель. Например, выражение:

```
(x1 + 25)*factor
```

будет транслировано в:

```
LOAD x1;
ADD 25;
MUL factor;
```

Когда вычисляется арифметическое выражение, часто требуются дополнительные размещения в памяти для сохранения промежуточных результатов вычисления. Символическими адресами для таких размещений будут служить $1, $2, $3 ... и т.д. Результат трансляции выражения:

```
(a + 318)*(b - c)
```

будет выглядеть как:

```
LOAD b;
SUB c;
```

```
STORE $1;
LOAD a;
ADD 318;
MUL $1;
```

Присвоим нашей программе-транслятору имя **TRAREX**. Далее приводится полный текст программы, сопровождаемый комментариями к её составлению.

```
* Program TRAREX,
* Translation of arithmetic expressions
$ENTRY Go { =  <Open 'r' 1 <Arg 1>>
               <Open 'w' 2 <Arg 2>>
               <Write <Translate <Input 1>>>; }

$EXTRN Input;

Translate {
   = ;
  e1 = <Code-gen (1) <Parse e1> <DG 'compl'>>
         }
* Last symbol from a given list in expression
* <Last (e.Symb-List) e.Expr ()>  ==  e1 sX(e2)
*    where sX is the symbol looked for,
*    and e1 sX e2 is e.Expr,
* or                               ==  (e.Expr)
*    if there are no such at top level
Last {
  (eA sX eB) e1 sX(e2) = e1 sX(e2);
  (eA) e1 tX(e2) = <Last (eA) e1(tX e2)>;
  (eA) (e2) = (e2);  }

* <Parse e.Arithm-expr>  ==  e.Tree
* e.Tree  == s.Operation (e.Tree-1) e.Tree-2
*         == s.Operand
*         ==  empty
Parse {
 e.Expr, <Last ('+-') e.Expr()>: e1 s.Op(e2) =
                  s.Op(<Parse e1>) <Parse e2>;
 e.Expr, <Last ('*/') e.Expr()>: e1 s.Op(e2)
       = s.Op(<Parse e1>) <Parse e2>;
  e1'^'e2 = '^'(<Parse e1>) <Parse e2>;
  s.Symb, <Type s.Symb>: 'F's.Symb = s.Symb;
  s.Symb, <Type s.Symb>: 'N's.Symb = s.Symb;
  (e.Expr) = <Parse e.Expr>;
   = ;
  e.Exp =
     <Prout 'Syntax error.Cannot parse 'e.Exp>
     <BR 'compl=' Fail>;
      };
* Code generation.
* <Code-gen (sN) e.Parsing-tree>
*                    ==  assembler code
* sN is the number of the first location
* free for intermediate results.
Code-gen {
```

```
  e1 Fail  = ;
  (sN) '-' ()e2 =
           <Code-gen (sN) e2>
           'MINUS ;';
  (sN) s.Op(e1)s2 =
           <Code-gen (sN) e1>
           <Code-op s.Op> <Outform s2>';';
  (sN) '+'(s1)e2 =
           <Code-gen (sN) e2>
           <Code-op '+'> <Outform s1>';';
  (sN) '*'(s1)e2 =
           <Code-gen (sN) e2>
           <Code-op '*'> <Outform s1>';';
  (sN) s.Op(e1)e2 =
           <Code-gen (sN) e2>
           'STORE $'<Symb sN>';'
           <Code-gen (<Add (sN)1>) e1>
           <Code-op s.Op> '$'<Symb sN>';';
  (sN) s.Symb =
           'LOAD '<Outform s.Symb> ';';
  (sN) eX = (Synt-err)';';
       };

* Convert operands into character strings
Outform {
     sS, <Type sS>:
        {'F'sS = <Explode sS>;
         'N'sS = <Symb sS>; };
        }
* Assembler codes of operations
Code-op {
  '+' = 'ADD ';
  '-' = 'SUB ';
  '*' = 'MUL ';
  '/' = 'DIV ';
  '^' = 'POWER ';
       };
* Write assembler code in a column
* (on screen and on disk)
Write {
  (Synt-err)';' e2 = <Prout 'Syntax error'>;
  e1';' e2 = <Prout <Put 2 e1>> <Write e2>;
  = ;  }
```

Эта программа читает выражение из одного файла и помещает результат трансляции в другой файл. Она также выводит результат трансляции на терминал. Программе требуется одна внешняя функция, **`Input`**, которая определена в модуле **`reflib`**, поставляемом с пакетом РЕФАЛ-системы; предполагается, что **`reflib.rsl`** размещён в текущей директории. Для того, чтобы вызвать **`trarex`**, следует ввести:

```
refgo trarex+reflib  input-file output-file
```

Стартовая функция **`Go`** использует встроенную функцию **`Arg`**, чтобы сделать параметры, вводимые вместе с вызовом программы, доступными для РЕФАЛ-машины. **`<Arg 1>`** предназначен для *входного*

файла, а **<Arg 2>** — для *выходного файла*. Функция **Open** назначает эти файлы для чтения и записи по каналам 1 и 2 соответственно.

Трансляция с одного языка программирования на другой включает следующие три этапа:

1. Лексический анализ. На этом этапе символы входного текста преобразуется в минимальные лексические единицы исходного языка.
2. Синтаксический анализ, или грамматический разбор. Здесь текст на исходном языке преобразуется в древовидные структуры, которые выражают смысл текста в форме, удобной для третьего этапа.
3. Генерация кода. По дереву синтаксического анализа порождается код на выходном языке.

В нашем случае лексическими единицами являются идентификаторы, целые числа, знаки операций и круглые скобки. Функция **Input** полностью осуществляет необходимый лексический анализ. Она не только сворачивает строки символов, представляющие идентификаторы и числа, в неделимые символы-атомы, но и преобразует символы-скобки в круглые РЕФАЛ-скобки, которые надлежащим образом объединяются в пары. Такое преобразование уже является началом формирования синтаксического дерева; оно делает более лёгким следующий этап синтаксического анализа с помощью функции **Parse**.

ЗАМЕЧАНИЕ: Символ - на клавиатуре используется и как дефис, и как знак минус. Первая его функция используется в идентификаторах. Поэтому **x-1** будет восприниматься как идентификатор. При использовании для вычитания знак - окружается пробелами.

Входом для **Parse** служит результат функции **Input**. Её выходом является синтаксическое дерево, которое должно использоваться функцией генерации кода **Code-gen** (см. определение функции **Translate**). Следует выбрать точный формат этого дерева так, чтобы сделать генерацию кода более лёгкой.

Главным критерием для проектирования представления древовидного объекта в РЕФАЛе является то, что все варианты объекта, которые желательно различать в программах, должны быть различимы за счёт применения простых образцов. Использование открытых e-переменных в образцах следовало бы ограничить, за исключением случаев, когда это совершенно необходимо; таких, например, как случай, когда ведётся поиск элемента списка. Круглых скобок, не являющихся необходимыми, следует избегать. Часто в форматы объектов включается ряд символов для того, чтобы провести различие между вариантами и повысить читабельность. В нашем случае дерево имеет простую структуру. В процессе генерации кода нам нужно различать элементарные операнды и поддеревья. Используется формат, где элементарные операнды всегда являются символами, в то время как поддеревья никогда ими не являются. Таким образом, нет нужды в дополнительных средствах различения.

Случай элементарного операнда (листа дерева) задаётся следующей формулой:

```
e.Tree == s.Operand
```

а случай бинарной операции — формулой:

```
e.Tree == s.Operation (e.Tree-1) e.Tree-2
```

Следует также разрешить одну унарную операцию в арифметических выражениях: изменение знака, которое представлено тем же символом'-', что и бинарная операция вычитания. Когда проводится синтаксический анализ выражения, можно трактовать унарный минус таким же образом, что и бинарный; единственная разница между ними в том, что в случае унарного минуса первый операнд является пустым. Поэтому добавляется ещё один вариант для типа данных **e.Tree**:

```
e.Tree  == - () e.Tree
```

и он рассматривается как отдельный случай на этапе генерации кода.

Проектируя структуры данных в РЕФАЛе, следует принимать в расчёт как интересы функций, производящих эти структуры, так и интересы функций, их использующих. Часто случается, что структура, первоначально изобретённая в интересах одной из этих партий, требует модификаций для того, чтобы лучше удовлетворить потребности другой партии. Иногда вам придётся произвести целый ряд таких подгонок перед тем как прийти к окончательным структурам данных, по мере того как вы вдаётесь в детали алгоритма. Это может потребовать некоторых усилий, но выигрышем для вас является лучшее представление и, почти наверняка, лучшее понимание алгоритма. Усилия эти окупятся во время тестирования и отладки.

При синтаксическом анализе используется способность РЕФАЛ-машины совершать скачок от левой или правой скобки к парной ей скобке и просматривать выражения как слева направо, так и справа налево. Для левоассоциативных операций, а именно, для **+**, **-**, *****, и **/**, самая верхняя операция в дереве синтаксического разбора является последней на уровне главных скобок, например:

```
a - b - c = (a - b) - c
```

В этом выражении невозможно использовать открытые e-переменные для локализации нужного знака минус, потому что порядок слева-направо встроен в процедуру удлинения. Но можно производить просмотр терм за термом справа налево, пока не обнаружится первый минус. Это производится при помощи функции **Last**, которая обнаруживает последнее вхождение символа из данного перечня ("найти" в РЕФАЛе означает поставить скобки в соответствующих местах). (Форма входа и выхода функции **Last** подробно описана в комментариях к программе.)

Операция возведения в степень ^ является правоассоциативной, поэтому при ее поиске в грамматическом разборе сверху-вниз можно использовать открытые переменные.

Все это ясно можно видеть в определении функции **Parse**. Каждое предложение в нем соответствует синтаксической категории исходного языка. Сперва выражение разделяется на части аддитивными операциями верхнего уровня **+** и **-**; затем мультипликатиными операциями ***** и /. В третьем предложении уделено внимание операции **^**. Важен порядок этих предложений; он отражает отношение старшинства между операциями.

Но что делать с унарной операцией изменения знака? Она связывает сильнее, чем аддитивные или мультиприкативные операции, но слабее, чем операция возведения в степень. Одно из решений — добавить соответствующее предложение между вторым и третьим предложениями. Но тогда мы должны быть уверены, что первое предложение не будет работать с пустым значением переменной **e1**. Этого можно добиться, заменяя **e1** на **tX e1**. Однако мы применяем в программе другое решение. Просто разрешено первому (так же, как и второму) аргументу бинарных операций быть пустым, таким образом расширяется определение дерева грамматического разбора. Когда дело доходит до генерации кода, то разрешено только в одном случае листу дерева быть пустым — в случае унарного **-**.

Однако, такая простота не даётся даром. Выражение типа **()-x** будет считаться законным и эквивалентным **-x**. (О лучшем решении см. в упражнении далее.)

Остальные пять предложений **Parse** относятся к идентификаторам, числам, скобкам, пустым выражениям и ошибочным ситуациям.

Контроль ошибок является важной частью трансляции. В нашей программе лексические ошибки

(включая непарные скобки) отлавливаются стандартной функцией **Input**. В процессе грамматического разбора в поле зрения может появиться много параллельных вызовов функции **Parse**. Если хотя бы при одном из таких вызовов встречается синтаксическая ошибка, не имеет смысла генерировать код. Программа должна выдать сообщение об ошибке и либо прекратить процесс, либо продолжить грамматический разбор с целью обнаружения других возможных ошибок, но в любом случае прерывая генерацию кода. Вторая возможность предпочтительнее, и в нашей программе выбрана эта опция. Чтобы передать сообщение процедуре **Code-gen**, **Parse** закапывает символ **Fail** под именем **'compl'** (сокращение от 'completion' - "завершение"). При вызове **<Code-gen> 'compl'** выкапывается и добавляется к правой части аргумента. Если ничего не было закопано (ошибок нет), добавление будет пустым. Первое предложение функции **Code-gen** проверяет, присутствует ли **Fail**, и если это так, заканчивает вычисление.

Функция **Code-gen** порождает ассемблерный код по дереву грамматического разбора арифметического выражения. Она имеет ещё один аргумент — число, отмечающее первое свободное размещение для промежуточных результатов вычислений. Вначале оно равно 1.

Когда наша учебная машина выполняет операцию, первый операнд находится в накопителе, а второй по определённому адресу. Однако хороший генератор кода должен был бы использовать коммутативность сложения и умножения и переставлять порядок операндов в случае, когда это может улучшить код. Например, при порядке операндов, заданном в выражении

```
a * (b - c)
```

результат трансляции имеет вид:

```
LOAD b;
SUB c;
STORE $1;
LOAD a;
MUL $1;
```

Вот, несомненно, лучший вариант трансляции:

```
LOAD b;
SUB c;
MUL a;
```

Это результат трансляции, который будет получен при выполнении нашей программы. В определении **Code-gen** имеются специальные предложения, отслеживающие ситуации, в которых инвертированный порядок более предпочтителен.

Некоторые ошибки не могут быть обнаружены вплоть до этапа генерации кода, например, пустой операнд (кроме случая пустого операнда для **-**). Последнее предложение **Code-gen** обнаружит такую ошибку, а функция вывода **Write** выдаст на печать сообщение и остановится. Что касается других предложений **Code-gen**, легче уяснить, как они работают, из текста на РЕФАЛе , чем из записываемых пояснений.

Упражнение 5.3

Модифицировать **TRAREX** таким образом, чтобы она работала в интерактивном режиме, запрашивая у пользователя ввод выражения, транслируя его и снова запрашивая, пока пользователь не наберёт **END** вместо выражения.

Упражнение 5.4

Как упомянуто выше, **TRAREX** воспринимает `()-x` как `-x`. Модифицировать `Parse` так, чтобы пустое подвыражение всегда рассматривалось как ошибка.

6. МЕТАСИСТЕМНЫЙ ПЕРЕХОД

При написании программы мы имеем дело с некоторым объектным полем — множеством объектов, к которым мы обращаемся, когда используем функции. Это множество включает области определения и области значения всех функций, т.е. множества их допустимых аргументов и значений. Будем считать, что это множество объектов основного уровня, или уровня 0, в иерархии объектов. Уровнем 1 в этой иерархии является РЕФАЛ-машина с загруженными в неё функциональными определениями. Взаимодействием между этими двумя уровнями является процесс вычислений, при котором объекты уровня 0 составляют материал, "плоть" процесса, в то время как РЕФАЛ-машина управляет созданием и преобразованием этого материала. Будем называть систему *S*1 уровня 1, которая осуществляет управление совокупностью систем *S*0 уровня 0, метасистемой по отношению к уровню 0.

Эта двухуровневая иерархия расширяется, когда создаётся следующий уровень управления, который имеет дело с РЕФАЛ-машиной и с функциональными определениями как с управляемыми объектами. Назовём такой шаг метасистемным переходом.

6.1. МЕТАФУНКЦИЯ `Mu`

Читатель мог бы уже заметить, что в варианте нашего языке, определённом до настоящего момента, нет способа применения таких функций, для которых имя не задаётся явно, а является значением некоторой переменной либо результатом вычислений. Согласно синтаксису РЕФАЛа, после открывающей вычислительной скобки обязательно должно следовать символическое имя функции (идентификатор). Это требование, которое, разумеется, введено с целью эффективной реализации, не позволяет использовать такие выражения как:

```
<s.F e.X>
```

которое можно было бы записать для того, чтобы вызвать функцию, чьё имя определяется динамически как значение переменной **`s.F`**. Однако РЕФАЛ-5 позволяет достичь того же эффекта посредством использования выражения:

```
<Mu s.F e.X>
```

Здесь **`Mu`** является встроенной функцией, или, скорее, *метафункцией*, осуществляющей вызов функции с таким именем, которое задаётся символом, непосредственно следующим за **`Mu`**. Остальная часть выражения становится аргументом этой функции. Таким образом, **`Mu`** работает, как будто определена следующим предложением:

```
Mu { s.F e.X = <s.F e.X> }
```

если оно синтаксически допустимо.

Применяя **`Mu`**, можно определять различного рода *интерпретаторы*, которые используют выражения в качестве программ, специфицирующих, какие из функций низшего уровня (функции-исполнители) должны быть вызваны. Вот простой пример интерпретатора, который последовательно применяет заданный список операторов к заданному аргументу :

```
Use-seq
 (s.F e.1) On e.X = <Use-seq (e.1) On <Mu s.F e.X>>;
 () On e.X = e.X;
```

Модульная структура программ в РЕФАЛе-5 привносит некоторое усложнение в простую идею функции **Mu**. Предположим, функция **Mu** уже применена для того, чтобы определить некоторую функцию **Callmu** в одном модуле, а затем **Callmu** используется как внешняя функция в другом модуле. Может случиться так, что некоторое имя функции используется в обоих модулях для определения различных локальных функций. Какую их этих функций следует тогда вызывать посредством **Mu**? Рассуждая подробнее, рассмотрим следующие два модуля — **Mod1**:

```
* This module is the main module Mod1.
* It uses the function Callmu defined in Mod2.
* It also defines function F
* which has a different definition in Mod2.
$ENTRY Go { =  <Prout 'Mu: ' <Mu F 'ABC'>>
      <Prout 'Callmu: ' <Callmu F 'ABC'>> }
$e.XTRN Callmu;
F { e.1 = 'Mod1' }
```

и **Mod2**:

```
* This auxiliary module is Mod2. It defines
* a simple interpreting function Callmu,
* which uses Mu. It also defines function F
* which is defined differently in Mod1.
$ENTRY Callmu { s.F e.X = <Mu s.F e.X>; }
F { e.1 = 'Mod2' }
```

Можно определить две версии метафункции **Mu**: Mu-Статическую (**MuS**) и Mu-Динамическую (**MuD**). Согласно статическому определению функциональное значение символического имени определяется модулем, где *записана* функция, использующая **MuS**; в нашем случае этим модулем является **Mod2**. Общее правило таково: когда бы ни появился вызов **MuS**, он может активизировать только функцию, *видимую* из этой точки (т.е., либо локально определённую, либо входящую в список **$e.XTERN**), и вызываемая функция будет функцией, определённой в этой точке. При динамическом определении функциональное значение символического имени определяется во время выполнения, т.е. в главном модуле: в нашем случае это **Mod1**. Пусть выполняется:

```
refgo Mod1+Mod2
```

Если **Mu** определена статически (как **MuS**), программа выдаст на принтер:

```
Mu: Mod1
Callmu: Mod2
```

Если же **Mu** определена динамически (**MuD**), будет напечатано:

```
Mu: Mod1
Callmu: Mod1
```

Следует привести некоторые соображения как за, так и против указанных способов определения **Mu**. Динамическое определение выглядит более естественным и соответствует общему принципу: если функция не видима из главного модуля, она не может быть активизирована. С другой стороны, с системной точки зрения хорошо иметь возможность вызвать любую функцию программы, если это нужно. **MuD** не допускает этого, в то время как **MuS** позволяет это проделать. В самом деле, пусть используются модули с именами **Mod1**, **Mod2**, **Mod3**, и т.д. В **Mod1** определяется:

```
$ENTRY Mu-Mod1 { e.1 = <Mu e.1> }
```

в **Mod2**:

```
$ENTRY Mu-Mod2 { e.1 = <Mu e.1> }
```

и т.д. Тогда, в какой бы момент мы ни пожелали вызвать функции, определённые в некотором модуле **Mod*i*** , мы вызываем **Mu-Mod*i*** (не забывайте включать эту функцию в список **$e.XTERN**). Динамическая функция **MuD** может быть имитирована применением статической функции **Mu**. Следует только определить:

```
$ENTRY MuD { e.1 = <Mu e.1> }
```

в главном модуле и затем использовать **MuD** в любом модуле (снова объявив её как внешнюю за пределами главного модуля).

С учётом этого, встроенная функция **Mu** в нашей РЕФАЛ-системе определяется статически.

Упражнение 6.1

Предположим, определены пятьдесят функций с именами **Fun1**, **Fun2** ... и т.д. до **Fun50**. Определить функцию **Fun-n** так, чтобы <**Fun-n s.N e.X**> вызывала **s.N**-ю функцию из этой последовательности с аргументом **e.X**. Пример:

<Fun-n 12 'ABC'> *превращается в* **<Fun12 'ABC'>**

6.2. МЕТАКОД

Преобразование программ — это одна из важнейших областей применения РЕФАЛа; как программы, подлежащие преобразованиям (уровень 1, *объектные программы*), так и программы-преобразователи (уровень 2) обычно будут записываться на РЕФАЛе — это делает возможной самоприменимость программ-преобразователей.

Для того, чтобы записывать РЕФАЛ-программы, которые оперируют с РЕФАЛ-программами, мы должны представлять объектные программы с помощью некоторых объектных выражений, так как свободные переменные и функциональные скобки, которые используются в объектных программах, не могут подвергаться преобразованию в РЕФАЛе. В самом деле, предположим, что имеется некоторая программа и нужно преобразовать всякий вызов функции **F1** из этой программы в вызов **F2**. Предложение, подобное:

```
Subst21 { e.1 <F1 e.X> e.2  =
          e.1 <F2 e.X> <Subst21 e.2>; }
```

не сработает. Согласно синтаксису РЕФАЛа, активные подвыражения не могут использоваться в левой части. Но даже если бы мы расширили РЕФАЛ, допустив такое использование, активное подвыражение **<F2 e.1>** в правой части воспринималось бы как *процесс*, а не как объект; вычисление этого вызова начиналось бы перед следующим далее вычислением вызова **Subst21**, даже если бы это являлось нежелательным. Аналогично, мы не можем использовать свободную переменную в качестве объекта преобразования, поскольку она будет воспринята РЕФАЛ-машиной как сигнал к выполнению подстановки.

Отображение всех РЕФАЛ-объектов на множество объектных выражений будет рассматриваться как

некоторый *метакод*. Такое отображение, разумеется, должно быть инъективным; т.е. все образы ("метакоды") различных РЕФАЛ-объектов должны быть различными, так что существует единственное обратное преобразование. Преобразование-метакодирование приводит к понижению метасистемного уровня объекта — это переход от управляющего устройства к обрабатываемому объекту. Поэтому, когда метакодирование применяется к выражению **E**, говорим, что оно *погружается* в метакод; когда применяется обратное преобразование, говорим, что **E** *подымается* из метакода. Когда реально создаются РЕФАЛ-программы, работающие с РЕФАЛ-программами, следует выбрать некоторое конкретное метакодирование. Но когда ведутся рассуждения о погружаемых и подымаемых выражениях, часто хочется использовать систему обозначений, позволяющую оставлять такие преобразования неопределёнными. Поэтому погружение в метакод будет обозначаться символом "стрелка вниз" $\downarrow$; для подъёма из метакода выбирается символ "стрелка вверх" $\uparrow$. Когда область действия этих операций распространяется до конца текущего подвыражения, перед ним просто ставится $\downarrow$ либо $\uparrow$. При необходимости ограничить область их действия используются скобки. Например, конкатенация (сцепление) погруженного выражения **E**1 с незакодированным **E**2 изображается как {$\downarrow$**E**1}**E**2 , в то время как ($\downarrow$**E**1)**E**2 означает то же, что и ({$\downarrow$**E**1})**E**2. Очевидно, $\uparrow\downarrow$ **E** = **E**; и если $\uparrow$ **E** существует, то $\downarrow\uparrow$ **E** = **E**.

Целью метакодирования является отображение свободных переменных и вычислительных скобок в объектные выражения. Прекрасно было бы, если б при метакодировании можно было оставить объектные выражения неизменными. К сожалению, это невозможно из-за требования однозначности обратного преобразования. В самом деле, предположим, что имеется такой метакод. Тогда $\downarrow\downarrow$**e.1** должно быть равным $\downarrow$**e.1**, так как $\downarrow$**e.1** является объектным выражением. Отсюда вытекает, что два различных объекта, **e.1** и $\downarrow$**e.1**, имеют одинаковые метакоды.

Тем не менее, можно минимизировать различия между объектным выражением и его метакодом. Метакод, используемый в РЕФАЛ-системе, выделяет один символ, а именно звёздочку **'*'**, который изменяется при метакодировании. Все другие символы и объектные скобки (круглые) представляют сами себя. Наш метакод определён следующей таблицей:

Выражение E	**его метакод** $\downarrow$**E**
s.*I*	'*S'.*I*
t.*I*	'*T'.*I*
e.*I*	'*E'.*I*
<F **E**>	'*'((F) $\downarrow$**E**)
(**E**)	($\downarrow$**E**)
E1 **E**2	{$\downarrow$**E**1} $\downarrow$**E**2
'*'	'*V'
S (но не '*')	S

Таким образом $\downarrow$**e.X** является **'*E'.X** , $\downarrow$**s.1** является **'*S'.1**, $\downarrow$**'a*b'** представляет собой **'a*Vb'** , $\downarrow$**<F e.Arg>** является **'*'((F)'*E'.Arg)** и т.д.

Когда метакодирование применяется в некоторому объектному выражению, его результат может быть вычислен посредством вызова РЕФАЛ-функции **Dn**, которая заменяет всякий символ **'*'** на **'*V'** :

```
Dn {
  '*'e.1  =  '*V' <Dn e.1>;
  s.2 e.1 = s.2 <Dn e.1>;
  (e.2)e.1 = (<Dn e.2>) <Dn e.1>;
   = ; }
```

Функция **Dn** также реализована в РЕФАЛ-системе как встроенная функция, которая производит погружение своего аргумента в метакод за один шаг.

Для произвольного объектного выражения **E**0,

<Dn *E*0**> ==** $\downarrow$*E*0

Время, необходимое для погружения или подъёма объектного выражения пропорционально его длине. Часто является желательным отложить реальное метакодирование некоторого выражения до того момента, когда действительно потребуется применить его погруженную форму, поскольку вполне может случиться, что все выражение, или его часть, будут позднее подыматься к своему оригиналу. Поэтому следует избегать неоправданных двусторонних преобразований выражения, если существует способ пометить выражение, подлежащее метакодированию, но на самом деле не изменяющееся. Мы будем использовать выражение

'*!'(*E*0**)**

для представления отложенного метакода выражения **E0**. Это не нарушает однозначности обратного преобразования, так как комбинация **'*!'** не могла бы возникнуть никаким иным путём. Таким образом, обратное преобразование будет просто трансформировать **'*!'(E0)** обратно в **E0**. Следующее правило помогает управлять отложенным метакодированием: для всякого (не только объектного) выражения,

'*!'(*E***)** является эквивалентным **<Dn** *E***>**

В самом деле, даже если **E** включает свободные переменные и активные подвыражения, подразумеваемая последовательность действий в обеих частях этой эквивалентности одна и та же: первые свободные переменные должны быть замещены объектными выражениями, затем процесс вычисления должен начаться и закончиться, а затем результат должен быть погружен в метакод.

Обратной функцией к **Dn** является **Up**. Если ее аргументом является результат метакодирования объектного выражения **E0**, то **Up** возвращает значение **E0**:

<Up <Dn *E*0**>> ==** *E*0

Но область определения **Up** можно расширить так, чтобы она включала метакоды любого *основного (ground-*) выражения Egr (т.е. такого, которое может содержать активные подвыражения, но не содержит свободных переменных). Выберем ground-выражение, например, **<F 'abc'>**, и погрузим его в объектное выражение **'*'((F)'abc')**. Теперь, если применить к нему **Up**, оно будет подыматься назад к исходному активному выражению:

<Up '*'((F)'abc')> == <F 'abc'>

которое немедленно запускает процесс вычисления. Вообще, для любого ground-выражения **E**gr,

<Up ↓*E*gr**> ==** *E*gr

Функцию **Up** можно определить в РЕФАЛе следующим образом:

```
Up {
  '*V'e.1 = '*'<Up e.1>;
  '*'((s.F) e.1)e.2 = <Mu s.F <Up e.1>> <Up e.2>;
  '*!'(e.2)e.1 = e.2 <Up e.1>;
  s.2 e.1 = s.2 <Up e.1>;
  (e.2)e.1 = (<Up e.2>)<Up e.1>;
   =  ; }
```

Из определения **Up** легко видеть, что вычисление вызова **<Up** ↓*E*gr**>** является имитацией вычисления *E*gr. Эта функция конвертирует "замороженные" вызовы функций из *E*gr в их активную форму и проделывает это в точности в том же порядке изнутри-наружу, как это проделала бы РЕФАЛ-машина.

Упражнение 6.2

Если функция **Up** встречает первый метакод свободной переменной, например, **'*E'.X**, она оставит его без изменения, а это было бы ошибкой. Если её аргумент не является метакодом ground-выражения, эта функция должна быть не определена. В самом деле, подъём выражения **'*E'.X** должен приводить к свободной переменной **e.X**, которая будет помещена в поле зрения; однако, это не допускается определением РЕФАЛ-машины. Модифицировать приведённое выше определение **Up** таким образом, чтобы в случае, когда её аргумент не принадлежит области определения, выдавалось сообщение об ошибке и происходил аварийный останов.

На самом деле, рассмотренное выше определение функции **Up** представляет чисто академический интерес, так как РЕФАЛ-система содержит встроенную функцию **Up**, которая эквивалентна нашему РЕФАЛ-определению для аргумента ↓*E*gr, но работает быстрее. Она помещает *E*gr в поле зрения РЕФАЛ-машины всего за один шаг. Вдобавок её область определения расширена за счёт включения метакодов произвольных РЕФАЛ-выражений, а не только ground-выражений. Это будет обсуждено в *Разд. 6.4*.

Применяя функцию **Up**, мы имеем те же альтернативы, в терминах модульной структуры, что и для функции **Mu**. Наша встроенная функция **Up** является статической. Пользователь должен принимать во внимание, что функции, предназначенные для активизации с помощью **Up**, являются *видимыми* из модуля, в котором встречается вызов **Up**. Если функция, использующая **Up**, определена во вспомогательном модуле, нужно заменить её на **UpD** и определить **UpD** в главном модуле:

```
$ENTRY UpD { e.X = <Up e.X> }
```

совершенно аналогичным образом, как в трактовке функции **Mu** .

ЗАМЕЧАНИЕ: Следует иметь в виду различие между ↓ и ↑, с одной стороны, и между **Dn** и **Up**, с другой стороны. Первые просто являются метасимволами, используемыми для обозначения некоторых РЕФАЛ-объектов; сами они не являются ни РЕФАЛ-объектами, ни частью РЕФАЛ-системы. Вторые же являются законными именами функций.

Упражнение 6.3

Развернуть в РЕФАЛ-выражения (если это возможно):

```
(a) ↓<Add (35)16>
(b) ↓<Up (↓<F (e.1)e.2>)'*!'(e.3)>
(c) ↑'*'((Comp) 'A*VB')
(d) ↑'*!'('A*B')
```

Упражнение 6.4

Вычислить:

```
(a)<Dn <Add (35)16>>
(b)<Up '*!'('A*B')>
```

6.3. ВЫЧИСЛИТЕЛЬ

В РЕФАЛ-системе используется специальная функция **Go** для того, чтобы поместить начальное выражение в поле зрения РЕФАЛ-машины. Когда выражение вычислено, РЕФАЛ-машина останавливается и передаёт управление операционной системе компьютера. Это простейший способ взаимодействия между пользователем и РЕФАЛ-машиной. Иногда предпочтительнее был бы другой способ. Хотелось бы, чтобы РЕФАЛ-выражение подавалось на вход машины, получался результат его вычисления, и управление возвращалось бы пользователю без выхода из РЕФАЛ-системы, так что следующее выражение могло бы снова подвергнуться вычислению, и так сколько угодно раз.

Если пользователь искушён в написании выражений и захочет, чтобы вычисление производилось в метакоде, тогда решение этой проблемы является весьма простым. **Go** определяется как вызов интерпретирующей функции **Job**, которая предлагает пользователю ввести выражение, применяет к нему **Up**, и когда вычисления закончены, распечатывает результат и запрашивает следующее выражение.

Однако для пользователя может оказаться затруднительным записывать нужные выражения в метакоде; предпочтительнее разрешить ему записывать выражения в том же виде, в каком они появляются в программе. Поэтому, вместо функции **Input**, которая вводит объектные выражения, применяется другая функция — назовём её **Inp-met** — которая считывает выражение и погружает его в метакод. Главное её отличие от **Input** состоит в том, что символы **'<'** и **'>'** воспринимаются как вычислительные скобки.

Теперь следовало бы подумать о том, как привязать это определение **Go** к существующим модулям РЕФАЛа. Если бы **Up** была динамической, не оставалось бы ничего другого как заменить **Go** в главном модуле на **Go**, которая вызывает интерпретирующую функцию **Job**, описанную выше. Это значило бы, что программа может использоваться только в режиме вычислителя.

Однако наша функция **Up** является статической. Упомянутая выше **Go** оформляется как отдельный модуль, который называется **E** (от английского 'Evaluator' - Вычислитель), и модифицируется путём замены **Up** на **UpD** и объявления **UpD** как внешней функции. Для того, чтобы использовать любой модуль, скажем, **Prog**, совместно с вычислителем, следует добавить к **Prog** стандартное определение:

```
$ENTRY UpD { e.X = <Up e.X> }
```

Начало программы **E** выглядит следующим образом:

```
* Evaluator E
$ENTRY Go { =  <Job>; }
Job { = <Prout 'Type e.Xpression to evaluate. '
             'To end: empty line.'>
   <Prout 'To end session: empty e.Xpression'>
   <Prout >  <Check-end <Inp-met>>; }

Check-end {
     = <Prout 'End of session'>;
  '*'Error = <Job>;
  e.X = <Out <UpD e.X>>; }

Out {e.X = <Prout 'The result is:'>
         <Prout e.X> <Prout>
         <Job>; }
$e.XTRN UpD;
```

Теперь, если **Prog** является главным модулем (т.е. включает определение **Go**), мы можем использовать его в Go-режиме как прежде:

```
refgo prog ...
```

(где многоточие указано вместо вспомогательных модулей). Но каким бы ни был модуль **Prog**, главным или вспомогательным, мы всегда можем использовать его в режиме вычислителя:

```
refgo e+prog ...
```

Теперь модуль-Вычислитель **e** является, по существу, главным модулем, но имеется доступ к любой функции из **Prog** через **UpD**. Стандартное определение **UpD** может быть добавлено к любому РЕФАЛ-модулю заранее: это не причинит ущерба. На самом деле будет использоваться только **UpD** из первого модуля, следующего за вычислителем **e**. Трассировщик **reftr** может запускаться таким же образом.

Модуль-Вычислитель **e.ref** поставляется вместе с РЕФАЛ-системой.

ЗАМЕЧАНИЕ: При использовании вычислителя отсутствует интерпретация. Дополнительное время, по сравнению с режимом Go, тратится только на ввод выражения, подлежащего вычислению. Если ввод произведён, **Up** создаёт активное выражение в поле зрения, которое выполняется точно так же, как и всегда.

6.4. ЗАМОРАЖИВАТЕЛЬ

При преобразовании программы метакодированные вызовы функций уровня 1 (которые обычно включают некоторые свободные переменные) становятся аргументами функций уровня 2:

```
<F2 ... ↓<F1 ... e.1 ...> ... >
```

то есть

```
<F2 ... '*'((F1) ... '*E'.1 ...) ... >
```

Это схематическое преставление ситуации. В аргументе F^2 может присутствовать некоторая комбинация вызовов функций наряду с пассивными выражениями; назовём эти комбинации *конфигурациями* РЕФАЛ-машины, загружаемыми вместе с определениями функций уровня 1.

Основу любой системы преобразования программ в таких языках, как РЕФАЛ, составляет замена вызова функции значением, которое она порождает, за один шаг алгоритма. Во время преобразования программы в РЕФАЛе вызов сравнивается с левыми частями определяющих функцию предложений, которые, подобно функциональным конфигурациям, используются в метакодированной форме. Затем моделируется шаг РЕФАЛ-машины. Этот процесс известен как *прогонка*. В процессе прогонки множества допустимых значений свободных переменных разбиваются на подмножества, соответствующие разным предложениям функционального определения. Это делает возможным однозначное выполнение шага для любого подмножества. Предположим, например, что имеется следующее определение функции **Fa**:

```
Fa {
  'a's.X e.1 = 'b's.X <Fa e.1>;
  s.2 e.1 = s.2 <Fa e.1>;
  = ; }
```

и произведём прогонку вызова **<Fa e.1>**. Результатом одного шага прогонки является граф, в котором направленные ребра обозначают некоторые подмножества полного множества допустимых значений переменных, в то время как узлы представляют конфигурации, порождаемые при этом шаге. В нашем случае граф будет иметь три ребра (ветви). Первая ветвь определяет шаг для случая, когда значение **e.1** начинается с **'a'**, за которым следует некоторый символ **s.X**. Это ведёт к конфигурации:

```
'b's.X <Fa e.1>
```

где **e.1** теперь означает то, что осталось от первоначального выражения **e.1** после удаления из него первых двух символов. Вторая ветвь соответствует подмножеству значений, в которых первым символом является не **'a'** — или же это символ **'a'**, но остальная часть выражения либо пуста, либо начинается с круглой скобки. Третье подмножество состоит из одного элемента: пустого выражения.

Очевидно, что прогонка, как и всякая форма интерпретации, занимает гораздо больше времени, чем непосредственное вычисление. Поэтому, когда в конфигурации не имеется свободных переменных, было бы гораздо быстрее непосредственно вычислять ее, чем использовать общий механизм прогонки. Более того, всякий раз, когда шаг может быть выполнен однозначно, независимо от значений свободных переменных, также было бы очень желательно выполнять его при помощи непосредственного вычисления. Предположим, например, что при рассмотренном выше определении мы имеем конфигурацию:

```
<Fa 'a's.1'bc'e.2>
```

Независимо от того, каковы значения **s.1** и **e.2**, в результате получится конфигурация:

```
'b's.1 <Fa 'bc'e.2>
```

Аналогичным образом, следующие два шага будут однозначными и приводят к:

```
'b's.1'bc' <Fa e.2>
```

Только теперь следующий шаг зависит от значений переменных, и прогонка является неизбежной. Таким образом, в некоторых случаях прогонка может сводиться к прямому *частичному вычислению*

вызовов функций.

РЕФАЛ-система обеспечивает средства для эффективного частичного вычисления до тех пор, пока непосредственное выполнение шагов РЕФАЛа не зависит от значений свободных переменных . Это достигается следующим образом.

В поле зрения допускается использование объектов дополнительного типа. Эти объекты будут рассматриваться как *неизвестные*. Информационным наполнением неизвестного является его тип (**S** , **T** или **E**), его уровень (неотрицальное целое число) и его индекс (макроцифра). Система знает, что неизвестное s-типа означает некоторый символ, а неизвестное t-типа — некоторый терм; это принимается во внимание при сопоставлении. До тех пор, пока значения неизвестных объектов не требуются для выполнения РЕФАЛ-шага, они переходят из одного выражения в другое и могут быть скопированы. Система не замечает их существования, или вернее, их отличия от нормальных объектных выражений. Неизвестные можно обнаружить с помощью трассировщика, который распечатывает их в виде:

```
#type.level  inde.X
```

Неизвестные объекты создаются функцией **Up** и преобразуются с помощью **Up** и **Dn**. Если неизвестный объект, имеющий тип *t*, уровень *n*, и индекс *i*, обозначить как:

```
unknown(t,n,i)
```

то расширение этих функциональных определений может быть описано следующими формулами:

```
<Up '*'s.T s.I>  =   unknown(s.T,0,s.I);
<Up  unknown(t,n,i)> =  unknown(t,n+1,i);
<Dn  unknown(s.T,0,s.I)> = '*'s.T s.I;
<Dn  unknown(t,n+1,i)> =  unknown(t,n,i);
```

Имеется встроенная функция **Ev-met** ('вычислять поверх метакода'). Она воздействует на поле зрения таким образом, словно она определена предложением:

```
Ev-met { e.X = <Freezer <Up e.X>>;}
```

Вдобавок она выполняет некоторые системные операции. **Freezer** ("Замораживатель") является фиктивной функцией, которая нужна для заключения в скобки выражений, которые возникают при подъёме, и для вычисления аргумента функции **Ev-met**. Она также "замораживает" ее аргумент, т.е. погружает его в метакод. **Freezer** не может быть вызвана пользователем и появляется только как размещаемая функцией **Ev-met**. После того, как аргумент **Ev-met** поднят, начинается процесс прямого вычисления. Есть три варианта возможного его завершения:

- Вычисление заканчивается успешно с некоторым результатом E, который является пассивным выражением (но ещё может содержать неизвестные объекты). Тогда вызов замораживателя заменяется на:

  0 ↓ **E**

- На некоторой стадии процесса вычисления следующий шаг становится невозможным из-за неизвестного объекта. Тогда производится *замораживание*. Отыскивается первый заключенный в скобки вызов **Freezer**. Если таких вызовов несколько, выбирается самый внутренний; если вызовов нет, значит возникла ошибочная ситуация. Тогда аргумент **E** функции **Freezer**

погружается в метакод и вызов замораживателя заменяется на:

```
1 ↓ E
```

- На некоторой стадии активизируется вызов функции ввода-вывода. Но он не должен запускаться и выполняться, так как это может произойти во время выполнения программы, а не в момент функционального преобразования. Поэтому производится замораживание, как и в предыдущем случае. Результатом снова является:

```
1 ↓ E
```

- На некоторой стадии вычисления встречается аварийный останов (отождествление невозможно), при этом некоторое выражение заключено замораживателем в скобки. Тогда вызов замораживателя заменяется на:

```
2 ↓ E
```

Заметим, что даже если неизвестные объекты по сути выступают в качестве свободных переменных для обозначения некоторых вещей, следует вводить специальное обозначение для них, чтобы описывать операции над ними с помощью РЕФАЛ-предложений. В самом деле, нечто подобное

```
<Up '*E'.X>  =  e.X
```

противоречило бы синтаксису РЕФАЛа (свободная переменная в правой части, которая в левой части отсутствует), в то время как:

```
<Dn e.X>  =  '*E'.X
```

определяло бы **Dn** как совершенно иную функцию.

Функцию **Up** следует применять только в случае, когда известно, что её аргумент не содержит свободных переменных. Если таковые имеются, **Up** создаст неизвестные объекты, и, поскольку они не заключены замораживателем в скобки, возникнет ошибка, как только неизвестная переменная помешает вычислению. Заметим, что первым действием большинства встроенных функций является преобразование собственных аргументов из списочных структур в массивы. Поэтому они вызывают замораживание даже перед началом своей специальной работы.

С неизвестными нельзя оперировать так же, как и с другими законными ("известными") объектами РЕФАЛа. Единственное из назначение — ускорить процесс вычислений. Что бы вы ни намеревались с ними проделать, следует сперва сконвертировать все выражения, содержащие неизвестные объекты, в метакод. Тогда неизвестные становятся метакодированным представлением свободных переменных.

Приведем краткий очерк применения замораживателя при преобразовании программы. Предположим, что необходимо исследовать (метакодированный) вызов функции **Fa**:

```
'*'((Fa) E)
```

где **E** - некоторое выражение-образец. Перед прогонкой используется функция **Try-pe** ('try partial evaluation' - "попытка частичного вычисления"), которая определена следующим образом:

```
Try-pe { e.E = <Checkfr <Ev-met e.E>> }
* Check the contents of freezer
Checkfr {
```

```
    0 e.E = <Passive e.E>;
    1 e.E = <Drive e.E>;
    2 e.E = <Undefined e.E>;
         }
```

Когда производится вызов:

```
<Try-pe '*'((Fa) E)>
```

вызов **Fa** подымается из метакода и помещается в замораживатель. Предположим, **E** является выражением **'*E'.1** (произвольный аргумент). Тогда, не производя ни одного шага, система погрузит его обратно в метакод и перейдёт к функции **Drive** для прогонки. Но если

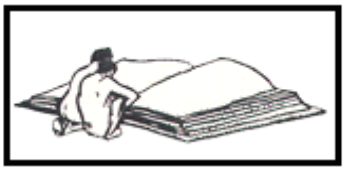
```
E  =  'a*S'.1 'bc*E'.2
```

как в приведённом выше примере, тогда непосредственно выполнятся три шага РЕФАЛ-машины, после чего будет предпринята прогонка. Если **E** является выражением **'a*S'.1'bc'**, вычисление будет продолжаться до получения пассивного результата **'b*S'.1'bc'**. Тогда функция **Passive** определит, что делать дальше.

Упражнение 6.5

Модифицировать вычислитель, описанный в *Разд. 6.3* таким образом, чтобы в случае, когда встречается аварийный останов, выдавалось сообщение, распечатывалось содержимое поля зрения в метакоде и управление было передано функции **Error**, которая, по предположению, справится с ситуацией.

7. ОТВЕТЫ К УПРАЖНЕНИЯМ



1.1.

```
'Joe''s Pizza is "cute"'
"Joe's Pizza is ""cute"""
```

1.2.

```
'-'16 0
```

1.3.

(a) **e1 sX sX** ; (b) **e1 tX e2 tX e3** ; (c) **t1 e2** or **e1 t2** .

1.4.

(a) **tX** превращается в **b**; (b) неудача; (c) **e6** превращается в **A(B)**, **e4** превращается в **C D**; (d) неудача.

1.5.

```
Add {
  (eX) '0' = eX;
  (eX) eY's'= <Add (eX) eY>'s'; }
```

1.6.

```
Addb {
  (eX'0')(eY s1) = <Addb (eX)(eY)> s1;
  (eX'1')(eY'0') = <Addb (eX)(eY)>'1';
  (eX'1')(eY'1') =
         <Addb (<Addb (eX)('1')>)(eY)>'0';
  (eX)(eY) = eX eY; }
```

2.1.

```
$ENTRY Go { = <Job> }
Job { = <Prout 'Please enter a line'>
       <Job1 <Card>>; }
Job1 {
  0  = <Prout 'Good-bye!'>;
  eX = <Prout 'The translation is:'>
       <Prout <Trans-line eX>>
       <Job>; }
```

2.2.

```
(a) Sum-1    and '#SUM_1 '
(b) Sum'-'1  and '#SUM -#1 '
(c) as above
(d) (9'+'25)'()' and       '(#9 +#25 )#(#)'
```

2.3.

```
* Conv-xx, conversion of a file typed for Input
* into a file for Xxin.
* Call: refgo conv-xx+reflib  file-name

$ENTRY Go { =
      <Xxout (<Arg 1>) <Input <Arg 1>>>; }
$EXTERN Input,Xxout;
```

2.4.

```
Merge { s1 s2 =
     <Implode <Explode s1> <Explode s2>>; }
```

2.5.

```
(a)
   7  6 8  5  4    9    3 2  10      11    1
   eA ( t2 )  (  Sunday ) ( s.Day  s.Night )
```

Переменная **eA** (No.7) является закрытой.

```
(b)
 1    3   2  4  5 7    9   8     10   6  11
 ( e.Word )  e1 ( ( e.Word ) e.Transl )  e2
```

No.3 является закрытым, No.4 открытым вхождением, No.9 повторным, No.11 закрытым.

```
(c)
  1 8   9  10  7  6  2  11  12  13  4  5  3
  ( e1 '+' e2 '+' e3 )  e1  '+' e2  (  e3 )
```

No.5 является закрытым, No.6 повторным, No.8 открытым, No.10 закрытым, No.11 и No.13 повторными.

2.6.

```
(a) ('di' ← eA)('f' ← t1)('ident' ← eB)
(b) (← e1)('d' ← sX)('iffi' ← e2) ('ent' ← e3)
(c) Failure
(d) (← e1)(← e2)(A B C ← e3)
(e) (A(A B) ← e1)((C) ← eX)(D ← e2)
```

3.1.

Второй знак уничтожения появится в ответе:

```
#Crocodile
```

Для выдачи сообщения в предложение следует вставить:

```
()'#'e2 = <Prout 'Extra delete signs!'>
            <Prout '#'e2>;
```

между предложениями 2 и 3.

3.2.

```
Cut {
  (e1)(e2) = <Cut1 (()e1) (()e2)>; }

Cut1 {
  ((e.R1) tX e1) ((e.R2) tX e2) =
         <Prout (e.R1 tX)(e.R2 tX)>
         <Cut1 (()e1) (()e2)>;
  ((e.R1) tX e1) ((e.R2) tY e2) =
         <Cut1 ((e.R1 tX) e1) ((e.R2 tY) e2)>;
  eZ = ; }
```

3.3.

```
Equal {
  (sX e1) (sX e2) = <Equal (e1) (e2)>;
  ((e.T1)e1) ((e.T2)e2) =
      <Equal1 <Equal (e.T1)(e.T2)> (e1)(e2)>;
  () () = T;
  (e1) (e2) = F; }

Equal1 {
  T eA = <Equal eA>;
  F eA = F; }
```

3.4.

```
F {
  sX e1 = <F1 sX()e1>;
   = <Prout 'No repeated symbols'>; }

F1 {
  sX(e2)sX e1 = sX e2 sX;
  sX(e2)sY e1 = <F1 sX(e2 sY) e1>;
  sX(e2) = <F e2>; }
```

3.5.

При неявной рекурсии:

```
Isect {
  (sX e1)(e2 sX e3) = sX <Isect (e1)(e2 e3)>;
  (sX e1)(e2) = <Isect (e1)(e2)>;
  ()(e2) = ; }
```

Без применения неявной рекурсии:

```
Isect {
  (sX e1) eI (sX e2) = sX <Isect (e1)(eI e2)>;
  (sX e1) eI (sY e2) =
         <Isect (sX e1) eI sY (e2)>;
  (sX e1) eI () = <Isect (e1) (eI)>;
  () eI (e2) = ; }
```

ЗАМЕЧАНИЕ: Был использован "закрытый" формат функции **`Isect`**, который обеспечивает память

ещё для одного выражения, вначале пустого. Если бы форматом функции был **<Isect (e1)e2>**, необходимо было бы ввести некоторую вспомогательную функцию.

3.6.

$2c + t$, где $t = 1$, если одна строка является началом другой. А при таком определении:

```
* Precedence-conservative
* <Prec eC(e1)(e2)> == T(e1)(e2)
*                   == F(e1)(e2)
* where eC is the common part of e1 and e2.
Prec {
  eC ()(e2) = T(eC)(eC e2);
  eC (e1)() = F(eC e1)(eC);
  eC (t1eX)(t1eY) = <Prec eC t1 (eX)(eY)>;
  eC (t1eX)(t2eY) =
      <Addc eC <Pre-term t1 t2>(t1eX)(t2eY)>;}

Addc { eC sT(eX)(eY) = sT(eC eX)(eC eY); }
```

время составляет $c + t$.

3.7.

```
String {
  e1 s2 = <String e1> s2;
   = T;
  e1(e2) = F e1(e2); }

Pal { eX = <Pal1 ()eX()>; }
Pal1 {
  (eL) s1 e2 s1 (eR) =
          <Pal1 (eL s1) e2 (s1 eR)>;
  (eL)(eR) = T eL eR;
  (eL) eI (eR) = F eL eI eR; }
```

3.8.

```
Fact {sN = <Loop F(1) I(1) N(sN)>; }

Loop {
  F(sF) I(sN) N(sN) = <* sN sF>;
  F(sF) I(sI) N(sN) =
        <Loop F(<* sI sF>) I(<+ sI 1>) N(sN)>;
     }
```

3.9.

```
* Recursive definition
Reverse {
  s1 e2 = <Reverse e2> s1;
   = ; }

*Iterative definition
Reversei { eX = <Revit ()(eX)>; }
```

```
Revit {
  (eR)(s1 e2) = <Revit (s1 eR)(e2)>;
  (eR)() = eR; }
```

3.10.

См. функцию **Pprout** в **reflib**.

3.11.

```
* Multibracket Backtracking
Mback {
 [e1(e2 ^e3)e4] = <Mback [e1 ^(e2 e3)e4]>;
 [.e1  ^e2.] = e1 e2; };
```

Функция достигает результата ,перемещая указатель обратно к началу выражения. Тот же результат может быть получен при перемещении указателя вперёд.

3.12.

(a)

```
* Call: <Tree [.  ^sX.]>
* where sX is the root node.

Tree {
*1. A leaf. Jump over.
 [e1 ^sX'.'e2] = <Tree [e1 sX'.' ^e2]>;
*2. A node with subtree. Enter it.
 [e1 ^sX(eT) e2] = <Tree [e1 sX( ^eT)e2]>;
*3. A 'hanging' node. Generate subtree.
 [e1 ^sX e2] = <Tree [e1 ^<Subtree sX> e2]>;
*4. Up the tree.
 [e1(e2 ^)e3] = <Tree [e1(e2) ^e3]>;
*5. End of work
 [.e1 ^.] =  e1; }
```

(b) Все те узлы, обработка которых завершена, находятся в левой мультискобке, причём в формате **sX(eS)**, где **eS** является обработанным поддеревом для **sX**. Те узлы, обработка которых уже начата, но не завершена, также находятся в левой мультискобке, но в формате

sX)(,

поскольку левая круглая скобка, следующая за **sX**, представляется инвертированной парой скобок. Поэтому указанное выше предложение 3 модифицируется как:

```
*3. A 'hanging' node. See whether repeated.
 [e1 ^sX e2] =  <Tree1
     (<Rep sX[. ^e.ML(e1 ).]> )(sX e2)e.MR>;
```

Две новые функции определяют следующим образом:

```
* See whether the subtree was already built
Rep {
*1. Repeated node found.
 sX [e1 ^sX(eS) e2] =
```

```
            Yes(eS) <Mback [e1 ^sX(eS) e2]>;
  *2. A different symbol.
   sX [e1 ^sY e2] = <Rep sX [e1 sY ^e2]>;
  *3. Enter a parenthesis
   sX [e1 ^(e2)e3] = <Rep sX [e1( ^e2)e3]>;
  *4. Exit a parethesis
   sX [e1(e2 ^)e3] = <Rep sX [e1(e2) ^e3]>;
  *5. End of work.
   sX [.e1 ^.] = No e1; }

  Tree1 {
  *1. Repeated node. Cope subtree.
   (Yes(eS)e1)(sX e2)e.MR =
           <Tree (e1)(sX(eS) e2)e.MR>;
  *2. New node. Compute subtree.
   (No e1)(sX e2)e.MR =
           <Tree (e1)(<Subtree sX> e2)e.MR>;
       }
```

4.1.

```
  Addop {
    e1 s.Op e2, '+-': eX s.Op eY =
             (e1) s.Op e2;
   e1 = <Prout 'No additive operations'>; }
```

4.2.

```
  Less {
   (e1) e2, <- e1 e2>: '-'eZ = T;
   (e1) e2 = F;
   eA = <Less <Format eA>>; }

  Lesseq {
   (e1) e2, <- e2 e1>: '-'eZ = F;
   (e1) e2 = T;
   eA = <Lesseq <Format eA>>; }

  Format {
   '-'s1 e2 = ('-'s1) e2;
   '+'s1 e2 = (s1) e2;
   s1 '-'e2 = (s1) '-'e2;
   s1 '+'e2 = (s1) e2;
   s1 e2 = (s1) e2; }
```

4.3.

```
  S123 {
    eA 1 e1, e1:
            {eB 2 e2, e2:
                     {eC 3 eD = T;
                      eC = F };
             eB = F };
    eA = F }
```

4.4.

```
Fa {
  e1 sX sY e2,
      <Less sY sX>: T,
      <Less 100 <+ sX sY>>: T = sX sY;
  e1 = <Prout 'No such pair'>;
  }
Fb {
  e1 sX sY e2, <Less sY sX>: T,
       <Less 100 <+ sX sY>>:
       {T = <Prout 'It is greater'>;
        F = <Prout 'It is not greater'>;
       };
  e1 = <Prout 'No such pair'>;
   }
```

4.5.

```
* Dictionary is stored as DICT
$ENTRY Go { =
         <Br 'dict='<Xxin 'DICT'>>
       <Prout 'Type in additions'>
         <Addition <Input>> ;
          }


Addition {
   =   <Prout 'No additions. Go ahead.'>
       <Job>;
  eX = <Prout 'OK. Go ahead.'>
       <Xxout ('DICT') eX<Cp 'dict'>>
       <Br 'dict='eX<Dg 'dict'>>
       <Job>;
         }
$EXTRN Input,Xxout,Xxin;
 ...  etc. as before
```

6.1.

```
Fun-n {
 sN eA = <Mu <Implode 'Fun'<Symb sN>> eA>; }
```

6.2.

Вставьте между третьим и четвёртым предложениями:

```
'*'e1 = <Prout 'Error: no upgrade of 'e1>;
```

6.3.

```
(a)'*'((Add) (35)16)
(b)'*'((Up) ('*V'((F) ('*VE'1)'*VE'2)) '*V!'('*E'3))
(c)<Comp 'A*B'>
```

(d) Невозможно. Это эквивалентно **<Dn 'A*B'>**, а ни одно выражение не становится активным,

будучи погруженным в метакод. Сравнить с Упражнением 6.4(b).

6.4.

```
(a)51
(b)'A*B'
```

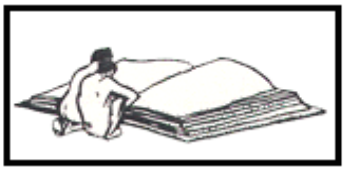
6.5.

Следует модифицировать две функции:

```
Check-end {
     = <Prout 'End of session'>;
  '*'Error = <Job>;
  eX = <Out <Ev-met eX>>; }

Out {
  0 eX = <Prout 'The result is:'>
         <Prout <Up eX>> <Prout> <Job>;
  1 eX = <Prout 'Result depends on unknowns'>
         <Prout> <Job>;
  2 eX = <Prout 'Recognition impossible'>
         <Prout 'View field in metacode:'>
         <Pprout eX> <Error eX>;
    }
$EXTRN Pprout,Error;
```

8. С П Р А В О Ч Н И К



ИНСТАЛЛЯЦИЯ И ИСПОЛЬЗОВАНИЕ ДЛЯ IBM XT/AT

Здесь приведены инструкции по инсталляции и использованию системы РЕФАЛ-5 на персональных компьютерах IBM AT или XT под управлением MS-DOS. Имеется также адаптация этой системы для Apple Macintosh. Её описание поставляется вместе с дискетой для Macintosh.

1. Инсталляция

Системная дискета содержит следующие файлы:

```
README
REFC     EXE
REFGO    EXE
REFTR    EXE
RDHELP   TXT
TEST     REF
TEST     RSL
E        REF
MBPREP   REF
REFLIB   REF
```

1. Скопируйте три **EXE** файла в любую поддиректорию, которая входит в **PATH**.

2. Скопируйте все остальные файлы, кроме **README**, в любую поддиректорию, допустим, **REFAL**.

3. Войдите в **REFAL** и выполните:

```
refc E
refc MBPREP
refc REFLIB
```

Это приведёт к появлению дополнительных файлов **E.RSL**, **MBPREP.RSL** и **REFLIB.RSL** в **REFAL**.

4. Введите строку:

```
SET RD_HELP=REFAL\RDHELP.TXT
```

в ваш **AUTOEXEC.BAT** файл. Это даст вам возможность читать help-файл из РЕФАЛ-трассировщика.

5. Теперь все установлено. Для проверки, что все работает как нужно, запустите:

```
refgo test
```

Это приведёт к запуску интерактивной программы, выполняющей трансляцию арифметических

выражений.

2. Использование

В дальнейшем **XXX** , **YYY** и т.д. будут означать некоторые имена файлов (без расширений).

Используя любой редактор текстов, создайте РЕФАЛ-программу в **XXX.REF**. Чтобы скомпилировать её, выполните:

```
refc XXX
```

Если в программе нет ошибок, никакие сообщения не выдаются. Файл **XXX.RSL**, который содержит результат трансляции РЕФАЛ-программы в интерпретируемый язык RASL, будет размещён в текущей директории. Если имеется сообщение об ошибках, смотрите файл **XXX.LIS**, содержащий листинг РЕФАЛ-программы с диагностикой ошибок.

Для запуска РЕФАЛ-программы выполните:

```
refgo XXX
```

Если потребуются функции, определяемые в других модулях: **YYY**, **ZZZ** и т.д., скомпилируйте **YYY**, **ZZZ** и т.д. и запустите:

```
refgo XXX+YYY+ZZZ  etc.
```

В частности, не забывайте включать модуль **reflib**, когда используете некоторые из стандартных функций, определённых в этом руководстве.

ЗАМЕЧАНИЕ: Убедитесь, что между знаками плюс и именами файлов нет пробелов.

При использовании **refgo** можно задать опции:

```
refgo -nt XXX+YYY etc.
```

либо

```
refgo XXX+YYY etc. -nt
```

Тогда по окончании работы на печать будут выдаваться количество шагов и время выполнения. **-n** соответствует выдаче только количества шагов, **-t** — только времени выполнения.

Чтобы запустить программу через трассировщик, вместо **refgo** используется **REFTR**:

```
reftr XXX+YYY+ZZZ  etc.
```

(описание трассировщика см. в Разделе справочника D).

В случае запуска программы в аргументами выполните:

```
refgo XXX+YYY etc. Arg1 Arg2 etc.
reftr XXX+YYY etc. Arg1 Arg2 etc.
```

Аргумент не может начинаться с **-**. Для организации доступа к аргументам из РЕФАЛ-программы используйте **<Arg s.N>** (см. Раздел справочника C).

При вычислении любых вызовов функций, определённых в некотором файле **XXX**, с применением вычислителя **E** (о нем можно справиться в разделе 6.3), проследите, чтобы этот файл содержал строку:

```
$ENTRY Upd { e.X = <Up e.X>; }
```

Затем введите:

```
refgo e+XXX+...
```

либо

```
reftr e+XXX+...
```

СПРАВОЧНИК ПО СИНТАКСИСУ

1. Лексические единицы

Лексические единицы РЕФАЛа подразделяются на специальные знаки, символы и переменные. Между лексическими единицами может проставляться любое количество как пробелов, так и знаков переноса. Пробел становится лексической единицей, когда он появляется в строке, заключённой в кавычки.

1.1 Специальными знаками являются:

- одинарная и двойная кавычки **'** и **"**;
- структурные, или круглые, скобки **(** и **)**;
- функциональные, или вычислительные, скобки **<** и **>**;
- индикаторы типа переменной **s**, **t**, **e** (малыми буквами - это важно);
- следующие ниже разделители, которые означают соответственно:

```
знак равенства   =   ''заменить на''
точка с запятой  ;   ''иначе''
двоеточие        :   ''является''
запятая          ,   ''где''
точка            .   ''далее следует индекс''
фигурные скобки { }  ''начало'' и ''конец''
```

Имеются также специальные системные ключевые слова: **$ENTRY** и **$EXTERNAL** (последнее можно использовать также в форме **$EXTERN** или **$EXTRN**). Системные ключевые слова следует записывать большими буквами.

1.2 Символами являются:

- символические имена, обычно называемые идентификаторами;
- макроцифры;
- действительные числа;
- знаки, набираемые на клавиатуре.

1.2.1 Идентификатор является строкой алфавитно-цифровых знаков, начинающейся с заглавной буквы. Его длина не должна превышать 15 знаков. Тире и подчёркивания также допустимы в идентификаторах; они являются эквивалентными. Большие и малые буквы внутри идентификаторов также эквивалентны.

1.2.2 Макроцифрами являются целые неотрицательные числа. Они представляются строками десятичных цифр. Значением наибольшей макроцифры служит 2^{32} - 1.

1.2.3 Действительные числа, не снабжённые знаком, должны начинаться и заканчиваться цифрой и включать десятичную точку, либо букву **E**, либо и то и другое. Точный синтаксис действительных чисел может быть задан следующей BNF (бэкусовской нормальной формой); здесь и далее альтернативы

располагаются в разных строчках:

```
real-number ::= unsigned-real
                - unsigned-real
                + unsigned-real

unsigned-real ::= digits . digits
                  digits . digits E digits
                  digits E digits

digits ::= decimal-digit
           decimal-digit digits
```

1.2.4 Знаковые символы заключаются в одинарные или двойные кавычки. Цепочка (строка) знаковых символов заключается в кавычки целиком; так, **'ab c'** есть последовательность четырёх знаковых символов (третьим символом является пробел). Недопустим перенос цепочки знаковых символов со строки на строку. Кавычка внутри цепочки, которая ограничена кавычками того же вида, представляется сдвоенной кавычкой. Следующие две строки-цепочки:

```
"Jimmy's Pizza"
'Jimmy''s Pizza'
```

представляют один и тот же РЕФАЛ-объект. Для того, чтобы избежать ложных кавычек, следует разделять заключённые в кавычки цепочки пробелами в том случае, когда они следуют непосредственно одна за другой. Размер цепочки ограничен 255 знаками, поэтому следует разбивать длинные цепочки на части.

1.3 Переменная представляет собой индикатор типа, за которым следует точка, а за ней, в свою очередь, следует индекс. Индикаторами типа являются:

- **s** (символьная переменная)
- **t** (переменный терм)
- **e** (переменное выражение).

Они должны всегда представляться малыми буквами. Индекс может быть идентификатором, либо числом. В случае односимвольного идентификатора либо числа, представленного единственной цифрой, точка может быть опущена. Если точка не опущена, идентификатор может начинаться с малой буквы. Каждое из представлений **eX**, **e.X** и **e.x** означает один и тот же РЕФАЛ-объект. **t12** и **eTree** не являются правильными переменными; необходима точка: **t.12**, **e.Tree**.

1.4 Знаки пробела, табулирования и возврата каретки (известные под общим названием "невидимые символы") игнорируются РЕФАЛ-компилятором, если они появляются между лексическими элементами. Они всегда играют роль разделителей, предотвращая поглощение лексических единиц каким-либо фрагментом, как, например, в случае последовательности идентификаторов или чисел. В то же время лексические элементы могут следовать друг за другом, не перемежаясь разделителями, до тех пор, пока это не мешает лексическому анализу. Например, употребление синтаксической конструкции **s1s2s3** совершенно законно и эквивалентно **s1 s2 s3**, так как за идентификатором типа ожидается появление в точности одного символа в случае, когда за ним не следует непосредственно точка. Однако, запись **s.1s.2s.3** является синтаксической ошибкой.

2. Выражения

Синтаксис РЕФАЛ-выражений в BNF-представлении имеет вид:

```
expression ::= empty
               term expression

term ::= symbol
         variable
         (expression)
         <f-name expression>

f-name ::= identifier
empty ::=
```

Выражение-образец, или просто образец, представляет собой выражение, которое не включает никаких вычислительных скобок. Основным выражением является такое выражение, которое не содержит ни одной переменной.

3. Предложения и программы

РЕФАЛ-программа есть перечень функциональных определений (*f-definition*'s — см. ниже) и внешних объявлений функций (*external-decl*). Для отделения одного внешнего объявления от другого должны использоваться точки с запятыми; ими могут также разделяться функциональные определения.

```
program ::= f-definition
            f-definitionprogram
            f-definition ;  program
            external-decl ;  program
            program  external-decl ;

f-definition ::= f-name {  block }
                 $ENTRY  f-name {  block }

external-decl ::= $EXTERNAL  f-name-list
                  $EXTERN  f-name-list
                  $EXTRN  f-name-list

f-name-list ::= f-name
                f-name , f-name-list

block ::= sentence
          sentence ;
          sentence ;  block

sentence ::= left-sideconditions= right-side
             left-sideconditions,  block-ending

left-side ::= pattern

conditions ::= empty
               , arg : pattern conditions

arg ::= expression

right-side ::= expression

block-ending ::= arg : { block }
```

4. Комментарии

Комментарии могут вставляться в текст программы, как это было определено выше. Они игнорируются

РЕФАЛ-компилятором. Имеются два вида комментариев:

- Строковые комментарии. Строка, которая начинается со звёздочки, воспринимается как комментарий.
- Комментарии-вставки. Подстрока, ограниченная комбинациями **/*** и ***/**, стоящая между лексическими элементами, является комментарием.

Строковые комментарии обычно относятся к следующему предложению(ям); комментарии-вставки относятся к предшествующему тексту.

ВСТРОЕННЫЕ ФУНКЦИИ

РЕФАЛ-5 содержит библиотеку стандартных встроенных функций. Она включает операции ввода/вывода и арифметические операции, различные системные функции, которые не могут быть заданы в РЕФАЛе, и некоторые простые процедуры, встроенные в систему для повышения ее эффективности. В случае нужды функциональное имя, используемое для встроенной функции, может быть присвоено функции, определяемой посредством РЕФАЛа. Тогда встроенная функция становится недоступной.

1. Функции ВВОДа/ВЫВОДа

<Card>

возвращает (замещается на) следующую строку из входного файла. Обычно она поступает с терминала, но ввод может быть переназначен любым допустимым в MS-DOS способом. Возвращаемое выражение представляет собой последовательность набираемых символов (быть может, пустую). Байт признака конца файла в неё не включается. Если ввод производится из файла, при достижении конца файла возвращается макроцифра 0 (строк больше нет). Это используется в программах как индикатор конца, так как в противном случае макроцифра не может возникать при вводе. При считывании с терминала тот же эффект вызывает ввод комбинации Control-Z .

<Print e.Expr>

распечатывает выражение **e.Expr** на текущее выводное устройство и возвращает (заменяется на) **e.Expr**.

<Prout e.Expr>

распечатывает выражение **e.Expr** на текущее выводное устройство и возвращает пустое выражение.

Функциям, которые работают с файлами, требуется в качестве аргумента файловый дескриптор. Дескриптор файла является макроцифрой в диапазоне 1-19; в некоторых операциях допустим дескриптор 0 , он считается ссылкой на терминал.

<Open s.Mode s.D e.File-name>

открывает файл **e.File-name** и связывает его с файловым дескриптором **s.D**. **s.Mode** является одним из символов: **'w', 'W'** (открыть для записи), либо **'r', 'R'** (открыть для чтения). **e.File-name** может быть пустым; **Open** в этом случае попытается открыть файл **REFAL*descr*.DAT**, где *descr* является десятичным представлением десткриптора **s.D**. Если текущим режимом является чтение, а такого файла не существует, выдаётся ошибка. Если режимом является запись, такой файл будет создан.

<Get s.D>

где **s.D** является файловым дескриптором либо нулём, действует подобно **<Card>**, за исключением того, что она получает входные данные из файла, указанного в **s.D**. Если ни один из файлов с таким файловым дескриптором ещё не открыт, **Get** попытается открыть файл **REFAL*descr*.DAT**, где *descr* является десятичным представлением **s.D**. В случае неудачи возникает ошибочная ситуация и выполнение программы завершается. Если значением **s.D** является 0, **Get** будет читать с терминала.

<Put s.D e.Expr>

где **s.D** является файловым дескриптором либо нулём, записывает **e.Expr** в файл с дескриптором **s.D** и возвращает **Expr** (подобно операции **Print**). Если ни один из файлов с таким файловым дескриптором ещё не открыт для записи, **Put** откроет файл **REFAL*descr*.DAT**, где *descr* является десятичным представлением **s.D**. Если значением **s.D** является 0, **Put** осуществит вывод на терминал. (Заметим, что этот вывод не является переадресуемым.)

<Putout s.D e.Expr>

возвращает пустое значение (подобно **Prout**). Во всем остальном **Putout** идентична функции **Put**.

---

2. Арифметические функции

Представление чисел. Целые числа представляются как последовательности *макроцифр* с использованием основания 2^{32}. Перед отрицательными целыми числами ставится символ **'-'**. Положительные числа могут снабжаться впереди знаком **'+'**. Арифметические функции возвращают целые числа в стандартной форме: **'-'** и последовательность макроцифр для отрицательного числа; отсутствие знака **'+'** для 0 или для положительного числа.

Действительные числа (произвольного знака) представлены как единичные символы и занимают 32-битное слово. (О синтаксисе действительных чисел см. Раздел справочника B.)

Основным форматом бинарной арифметической операции является следующий:

<*ar-function* (e.N1) e.N2>

Однако круглые скобки могут опускаться. В этом нет проблемы для действительных чисел, так всякое такое число представлено ровно одним символом. Когда же первый аргумент представляет собой целое число, по умолчанию от него будет браться одна макроцифра, возможно с предшествующим знаком, в

то время как остальная часть поступит во второй аргумент.

Если оба аргумента арифметической функции являются целыми числами, результат представляет собой также целое число; в противном случае он является действительным числом.

Реализованы следующие арифметические функции (как *ar-function* в формате):

`Add` или **`+`**
возвращает сумму операндов.

`Sub` или **`-`**
вычитает **`e.N2`** из **`e.N1`** и возвращает разность.

`Mul` или **`*`**
возвращает произведение операндов.

Div или **`/`**
если по крайней мере один из аргументов является действительным, функция возвращает действительное частное. Если оба являются целыми, **`Div`** возвращает целое частное от деления **`e.N1`** на **`e.N2`**; остаток игнорируется. В реализации этой и двух других функций деления возникает ошибка, если значением **`e.N2`** является 0.

`Divmod`
рассчитана на целые аргументы и возвращает

```
(e.Quotient) e.Remainder
```

Остатку присваивается знак **`e.N1`** .

`Mod`
рассчитана на целые аргументы и возвращает остаток от деления **`e.N1`** на **`e.N2`**.

`Compare`
сравнивает два числа и возвращает **`'-'`**, когда **`e.N1`** меньше, чем **`e.N2`**; **`'+'`**, когда больше; **`'0'`**, когда числа равны.

`<Trunc e.N>`

где **e.N** является целым числом, возвращает усечённое целое число.

<Real e.N>

где **e.N** является целым числом, возвращает равное ему действительное число.

<Realfun (e.Function) s.N> или

<Realfun (e.Function) s.N1 s.N2>

возвращает значение функции **e.Function** одного или двух аргументов. **e.Function** должно быть строкой символов, которая является названием функции, имеющейся в языке C. Например, **<Realfun ('log') s.N>** возвращает логарифм **s.N**. (Информацию о перечне имеющихся функций см. на системной дискете.)

---

3. Операции со стэком

<Br e.Name'=' e.Expr>

закапывает (см. Главу 4) выражение **e.Expr** под именем **e.Name**. Имя не должно содержать знак **'='** на верхнем уровне структуры.

<Dg e.Name>

выкапывает выражение, закопанное под именем **e.Name**, то есть возвращает последнее выражение, закопанное под этим именем и удаляет его из стэка. Если не существует выражения, закопанного под именем **e.Name**, **Dg** возвращает пустое выражение.

<Cp e.Name>

работает как **Dg**, но не удаляет выражение из стэка.

<Rp e.Name'=' e.Expr>

замещает выражение, закопанное под именем **e.Name**, на **e.Expr**.

<Dgall>

выкапывает стэк полностью. Стэк является строкой термов вида

```
(e.Name'=' e.Value)
```

Всякий раз, когда вызывается **Br**, такой терм добавляется к левой части. **Dg** удаляет самый левый терм, в то время как **Cp** копирует его, а **Rp** заменяет его.

---

4. Обработка символов и строк

<Type e.Expr>

возвращает **s.Type e.Expr**, где **e.Expr** является неизменным, а **s.Type** зависит от типа первого элемента выражения **e.Expr**.

s.Type	**e.Expr** начинается с:
'L'	буквы
'D'	цифры
'F'	идентификатора или имени функции
'N'	макроцифры
'R'	действительного числа
'O'	любого другого символа
'B'	левой скобки
'*'	**e.Expr** является пустым

<Numb e.Digit-string>

возвращает макроцифру, представленную строкой **e.Digit-string**.

<Symb s.Macrodigit>

является обратной к функции **Numb**. Она возвращает строку десятичных цифр, представляющую **s.Macrodigit**.

<Implode e.Expr>

берет начальные алфавитно-цифровые символы выражения **e.Expr** и создаёт из них идентификатор (символическое имя). Начальная строка в **e.Expr** должна начинаться с буквы и заканчиваться неалфавитным символом, скобкой или признаком конца выражения. Строка не должна превышать 15 знаков. Подчёркивание и тире также допустимы. **Implode** возвращает идентификатор, за которым следует необработанная функцией часть выражения **e.Expr**. Если первый символ не является буквой, **Implode** возвращает макроцифру **0**, за которой следует аргумент.

<Explode s.Identifier>

возвращает строку символов, которая составляла **s.Idenitifier**.

<Chr e.Expr>

замещает всякую макроцифру в **e.Expr** знаком клавиатуры с таким же ASCII кодом по модулю 256.

<Ord e.Expr>

является обратной к **Char**. Она возвращает выражение, в котором все символы замещены на макроцифры, совпадающие с символьными ASCII кодами.

<First s.N e.Expr>

где **s.N** является макроцифрой, разбивает **e.Expr** на две части — **e.1** и **e.2**, и возвращает **(e.1)e.2**. Если исходное выражение **e.Expr** имеет по крайней мере **s.N** термов (на верхнем уровне структуры), то первые **s.N** термов поступают в **e.1**, а остальные **e.2**. В противном случае **e.1** совпадает с **e.Expr**, а **e.2** является пустым.

<Last s.N e.Expr>

подобна **First**, но **s.N** термов поступают в **e.2**.

<Lenw e.Expr>

возвращает длину выражения **e.Expr** в термах, за которой следует **e.Expr**.

<Lower e.Expr>

возвращает исходное выражение **e.Expr**, в котором все заглавные буквы замещены малыми буквами.

<Upper e.Expr>

подобна функции **Lower**. Все малые буквы замещаются большими.

---

5. Системные функции

<Step>

возвращает последовательный номер текущего шага в виде макроцифры.

<Time>

возвращает строку, в которой указано текущее время работы системы.

<Arg s.N>

где **s.N** является макроцифрой, возвращает аргумент командной строки, который имеет порядковый номер **s.N**. Аргументы в командной строке не должны начинаться с **'-'** (чтобы не перепутать их с флажками).

<Mu s.F-name e.Expr> , или

<Mu (e.String) e.Expr>

отыскивает функцию, имеющую имя **s.F-name** или

<Implode e.String>

(когда оно задано в форме символьной строки) и применяет её к выражению **e.Expr**, то есть замещается на **<s.F-name e.Expr>**. Если ни одна такая функция не является *видимой* из вызова функции **Mu**, возникает ошибка.

<Up e.Expr>

производит подъём выражения **e.Expr** (восстанавливает его по метакоду). Об ограничениях на **e.Expr** см. в Главе 6.

<Dn e.Expr>

производит погружение **e.Expr** (метакодирует его).

РЕФАЛ-ТРАССИРОВЩИК

Трассировщик производит выдачу основных выражений из поля зрения в коде, который является расширением кода 'подмены выражений' (см Разд.2.3):

имеется в поле зрения	выдаётся как
s.Identifier	**#e.Char-string** *blank*
s.Number	**#e.Char-string** *blank*
(	**(**
)	**)**
<	**<**
>	**>**
'('	**#(**
')'	**#)**
'<'	**#<**
'>'	**#>**
'#'	**##**
неизвестное(*t*,*n*,*i*)	**#t.ni**

Идентификаторы выдаются большими буквами.

Для вызова РЕФАЛ-трассировщика используется **reftr** (см. Раздел справочника A). Перечень команд

трассировщика приведён ниже. В начале работы и после исполнения каждой команды (за исключением **quit**), трассировщик выдаёт подсказку:

TRACE>

названия команд и их параметры могут набираться как на верхнем, так и на нижнем регистре (за исключением спецификации образца *P*, когда регистр является важным). Командные параметры могут разделяться пробелами, табуляторами или запятыми (в приводимом ниже перечне использованы пробелы). Команды разделяются возвратами каретки или точками с запятой. Для того, чтобы продолжить команду в следующей строке, в конце текущей строки следует набрать косую черту **'\'**.

В нижеприведенном перечне сокращения названий команд и ключевых слов выделены с помощью круглых скобок; необязательные параметры заключены в квадратные скобки.

Команды трассировщика

set_break(set,break) *<F P>*

где ***F*** означает функцию, определённую в текущий момент (в каком-либо из используемых модулей), а **P** — образец для её аргумента. Команда устанавливает точку прерывания. Трассировщик остановится, когда активной функцией является ***F***, а её аргумент сопоставим с **P .**

---

print(pr,p) *value*

распечатывает значение *value*, определённое в текущий момент. Параметром *value* может быть:

view(v)	выдаётся текущее поле зрения.
некоторая РЕФАЛ-переменная	выдаётся значение переменной, определённое в текущей точке прерывания.
act	выдаётся текущее активное выражение.
result(res)	выдаётся результат выполнения команд **compute** или **step**.
call	выдаётся активное выражение, которое было вызвано для текущего замещения — применимое только после **step** или **compute**.

---

quit(q)

Останов РЕФАЛ-машины и выход из трассировщика.

---

exit(e.x)

То же, что и **quit**, но сопровождается выдачей некоторой информации: число шагов, содержимое поля зрения и т.д.

---

go

Возобновление выполнения программы до следующей точки прерывания либо до конца.

---

compute(comp,com) [result(res)]

Вычисление текущего активного выражения с последующим остановом. Если присутствует **result** либо **res**, по окончании выдаётся результат.

---

delete(del,d) *N*

Ликвидирует точку прерывания с номером *N*.

---

step(s) *N* **[result(res)]**

Производится *N* шагов и затем останов. Если *N* отсутствует, считается, что *N* = 1 . Если присутствует параметр **result** и *N* = 1 , выдаётся результат шага.

---

show(sho,sh) *key*

Демонстрирует некоторые текущие характеристики процесса вычислений. Параметром *key* может быть:

step	выдаётся текущий номер шага.
point	выдаётся текущая точка прерывания.
[break] *N*	выдаётся точка прерывания номер *N*.
all	выдаются все точки прерывания.
modules(mod) [*name*]	выдаются все функции, видимые из модулей с указанными именами. Если в качестве *name* задано **'*'** , то все модули выводятся на дисплей. Если *name* не задано, выдаётся только перечень доступных модулей.
function(fun) *name*	выдаётся имя модуля, в котором определена функция *name* .

---

help

Выдаёт перечень команд трассировщика.

. [*N*]

Точечная команда повторяет *N* раз последнюю группу команд, находящуюся в одной строке. Отсутствие *N* означает, что *N* = 1. Точка может занимать первую позицию в командной строке.