Московский Государственный Университет
имени М. В. Ломоносова

Факультет вычислительной математики и кибернетики

Е. И. Большакова, Н. В. Груздева

Основы программирования на языке Рефал

Учебное пособие

Москва
2009

УДК
ББК

Большакова Е. И., Груздева Н. В.
Основы программирования на языке Рефал (учебное пособие) – М.: Издательский отдел факультета ВМК МГУ (лицензия ИД № 05899 от 24.09.2001), 2009 – 93 с.

В учебном пособии описываются ключевые понятия и базовые механизмы функционального языка Рефал – одного из немногих отечественных языков программирования, получивших известность за рубежом. В последние годы учебные материалы по этому языку не публиковались, и данное пособие восполняет этот пробел. Рассматриваются особенности двух наиболее известных диалектов языка (Рефал-2 и Рефал-5), подробно разбираются примеры рефал-программ. В пособие включено также описание заданий практикума на языке Рефал, проводимого для студентов кафедры алгоритмических языков факультета ВМиК МГУ.

Авторы благодарят Н.В. Баеву за помощь в подготовке учебного пособия.

Рецензенты: доцент, к.ф.-м.н. Т.В. Руденко
ст. научн. сотрудник, к.ф.-м.н. С.И. Рыбин

Печатается по решению Редакционно-издательского совета факультета вычислительной математики и кибернетики МГУ им. М. В. Ломоносова

ISBN

Содержание

Язык Рефал (рекурсивных функций алгоритмический язык) был предложен в конце 60-х годов прошлого века для формализации синтаксического и семантического анализа программ, записанных на алгоритмических языках (Алгол, Фортран и др.) [1,2]. Хотя Рефал был задуман как *метаалгоритмический* язык, в итоге был создан функциональный язык программирования с уникальной концепцией вычислений [3-6] удобный для решения широкого круга задач обработки символьной информации. К этим задачам относятся не только интерпретация и трансляция языков программирования, но и выполнение аналитических преобразований в математике и теоретической физике, автоматическое доказательство теорем и другие задачи из области искусственного интеллекта. Общим для всех приложений языка Рефал является анализ и преобразование сложных выражений, записанных на формальных языках (алгоритмических языках, языке исчисления предикатов и др.).

1. Базисный Рефал

Настоящий раздел посвящён описанию основополагающих понятий языка Рефал, в совокупности образующих алгоритмически полное ядро первых известных его диалектов – языков Рефал-2 и Рефал-5. Эти диалекты более подробно рассматриваются в следующих разделах пособия. Используемый в этом разделе синтаксис базисного Рефала незначительно отличается от синтаксиса языка Рефал-2.

1.1. Рефал и нормальные алгоритмы Маркова

Нормальные алгоритмы, предложенные А.А. Марковым для формализации и изучения понятия алгоритма [7], можно рассматривать как концептуальную основу модели вычислений языка Рефал.

Нормальный алгоритм Маркова (НАМ) работает со словами (цепочками), составленными из символов некоторого конечного алфавита, и записывается как последовательность *предложений-правил* вида $A \to B$ или $A \mapsto B$, где A и B – некоторые слова в заданном алфавите. Каждое предложение-правило определяет некоторую замену слова A на слово B и может быть либо *простым* ($A \to B$), после применения которого НАМ

продолжает работу, либо *заключительным* ($A \mapsto B$), после применения которого НАМ завершает работу. Порядок составляющих алгоритм предложений существенен. Применение алгоритма к заданному слову C состоит в общем случае из нескольких шагов-преобразований исходного слова:

$C \to C_1 \to C_2 \to \ldots \to C_n$.

Каждый шаг включает:

I. Последовательный поиск применимого правила $A \to B$ или $A \mapsto B$, начиная с первого правила НАМ и далее до последнего. Правило $A \to B$ или $A \mapsto B$ применимо, если его левая часть – слово A входит как подслово в обрабатываемое в текущий момент слово C_j.

II. Преобразование слова C_j путём применения найденного правила: замены входящего в него слова A на слово B и получение таким образом слова C_{j+1} . Если слово A входит в C_j несколько раз, то заменяется первое такое вхождение.

Если применённое правило было заключительным, то работа НАМ заканчивается, и результатом алгоритма является полученное слово. В ином случае цикл «поиск-замена» продолжается, причём каждый последующий шаг применяется к результату предыдущего. Работа алгоритма завершается либо в случае применения заключительного правила, либо когда на одном из шагов цикла не было найдено ни одного применимого правила.

Хотя описанная модель вычислений алгоритмически полна (т.е. любой алгоритм можно записать в виде НАМ), она бедна с точки зрения выразительных средств и поэтому не может служить инструментом практического программирования. Даже для записи простейших преобразований слов требуются специальные приёмы, к примеру, введение в преобразуемое слово дополнительных специальных символов (при этом исходный алфавит дополняется этими символами). Количество же правил НАМ получается довольно большим.

Приведём в качестве примера три разных НАМ: первые два алгоритма работают со словами в исходном алфавите $\{a, b\}$, третий – со словами в исходном алфавите $\{0, 1, 2, 3\}$.

1. НАМ, удаляющий первый символ слова:

$*a \mapsto$

$*b \mapsto$

$\to *$

Алгоритм использует дополнительный знак * для того, чтобы пометить удаляемый символ. Третье правило, вводящее в слово знак *

вместо пустого подслова, намеренно помещено последним: поскольку это правило применимо всегда (пустое подслово входит в любое слово), его перемещение в начало приведёт к зацикливанию алгоритма.

2. НАМ, удаляющий в слове все буквы, кроме первой:

$$
\begin{aligned}
&*a \to a\$\\
&*b \to b\$\\
&\$a \to \$\\
&\$b \to \$\\
&\$ \mapsto\\
&\to *
\end{aligned}
$$

Для записи этого алгоритма требуются уже два дополнительных знака: знак * используется для пропуска первой буквы слова, а знак $ – для прохода по оставшейся части слова (слева направо) и последовательного удаления нужных символов.

3. НАМ, осуществляющий перевод целого числа без знака, записанного в четверичной системе счисления, в двоичное число:

$$
\begin{aligned}
&*0 \to 00*\\
&*1 \to 01*\\
&*2 \to 10*\\
&*3 \to 11*\\
&*\mapsto\\
&\to *
\end{aligned}
$$

Заметим, что с увеличением размера исходного алфавита для записи обрабатываемых слов растёт соответственно и число правил в нормальных алгоритмах Маркова.

Язык Рефал имеет с НАМ общую ориентацию на преобразования символьных выражений (слов), а также похожее строение алгоритма: алгоритм записывается как последовательность *предложений*, каждое из которых выполняет замену некоторой части обрабатываемого символьного выражения. В тоже время Рефал имеет более сложную модель вычислений и предоставляет программисту широкий набор языковых средств. К средствам, обеспечивающим бóльшую его выразительную мощность (по сравнению с НАМ), в первую очередь относятся:

- *Переменные*, позволяющие записывать символы и их последовательности в общем виде и тем самым определять сразу целый класс символов. Переменные используются и в левой, и в правой части предложений Рефала, общий вид которых:

выражение-образец = выражение-замена.

- *Структурные скобки*, задающие иерархическую структуру обрабатываемых символьных выражений. Поскольку с помощью структурных скобок можно записывать деревья, язык Рефал ориентирован на обработку древовидных структур.
- *Функции*, с помощью которых указываются те предложения программы, среди которых ищется предложение, применимое на текущем шаге вычислений. Под рефал-функцией понимается последовательность из нескольких предложений, а рефал-программа – это несколько функций, как правило взаимосвязанных. Использование функций позволяет также указать в правой части предложения часть символьного выражения, подлежащую дальнейшей обработке.

Приведём ниже три рефал-функции, реализующие те же символьные преобразования, что и представленные выше НАМ. Первые две функции состоят из одного предложения, а третья содержит 5 предложений. Строки, начинающиеся со знака *****, являются *строками комментария*.

```
* Функция удаления первого символа слова
f1   s1 e2 = e2

* Функция удаления всех букв слова, кроме первой
f2   s1 e2 = s1

* Функция перевода числа без знака, записанного в
* четверичной системе счисления, в двоичное число
f3   '0'  e2 = '00' <f3  e2>
     '1'  e2 = '01' <f3  e2>
     '2'  e2 = '10' <f3  e2>
     '3'  e2 = '11' <f3  e2>
          =
```

В приведённых примерах `f1`, `f2` и `f3` – это имена функций, а `s1` и `e2` – переменные, обозначающие соответственно некоторый символ и произвольное символьное выражение. Знак равенства в каждом рефал-предложении отделяет его левую часть (выражение-образец) от правой части (выражение-замена).

Угловые скобки в записи третьей функции означают вызов функции, имя вызываемой функции записывается сразу после угловой скобки <. Таким образом, функция `f3` является рекурсивной. Последнее предложение этой функции (интерпретируемое как *пусто* заменить на *пусто*) обеспечивает окончание рекурсии.

1.2. Выражения и переменные

Основной конструкцией языка Рефал, служащей для записи обрабатываемых символьных данных и правил их обработки, является так называемое *рефал-выражение*. Рефал-выражения, как и рефал-программы в целом, записываются при помощи знаков, которые принято делить на *собственные* и *объектные*.

Собственные знаки – это специальные символы языка, выполняющие в нём функцию, аналогичную роли ключевых слов в других языках программирования. К числу собственных знаков относятся круглые и угловые скобки **(** и **)**, **<** и **>**, знак равенства **=**, одинарная кавычка (апостроф) **'**, а также латинские буквы **S** и **s**, **T** и **t**, **E** и **e**.

Объектные знаки, или ***символы-литеры***, служат для изображения символов (литер), встречающихся в обрабатываемых данных. В набор объектных знаков входят цифры, буквы, знаки арифметических операций и другие знаки кода ASCII. В отличие от собственных знаков, любой объектный знак при записи рефал-выражения должен быть заключён в одинарные кавычки (апострофы), называемые ***символьными скобками***, например: `'B'`, `'+'`, `'e'`, `')'`. Таким образом, собственный знак **)** при записи отличается от объектного `')'`, а собственный знак **E** от объектного `'E'` и т.п. Запись `' '` обозначает литеру пробела, которая является такой же полноценной литерой, как и все остальные.

Если необходимо записать несколько подряд идущих объектных знаков, то эти знаки можно написать друг за другом и все их заключить в апострофы, например: `'x+y'`, `'define'`, `'  '` (последняя цепочка состоит из двух символов пробела). Заметим, что такая запись цепочек символов есть сокращение последовательной записи символов-литер, при которой между символами-литерами должен стоять один или несколько пробелов: `'x' '+' 'y','d' 'e' 'f' 'i' 'n' 'e',' ' ' '`.

Выражения языка Рефал строятся из *атомарных* (т.е. неделимых) *элементов*, каковыми являются *символы-литеры* и *переменные*, а также скобок трёх видов:

- ✓ структурные **( )**;
- ✓ функциональные < >;
- ✓ символьные `' '`.

Структурные скобки служат для структурирования обрабатываемых данных, а ***функциональные скобки*** – для записи обращений к функциям. Упрощённо говоря, рефал-выражение представляет собой некоторую

последовательность атомарных элементов, сбалансированную по всем трём видам скобок, например:

(`'for'` **(**`'i=1'`**))** `sa` **(**`'g'` `<f5 e1>` **(**`e2`**))**

В этом примере `'for'`, `'i=1'`, `'g'` – цепочки символов-литер, `sa`, `e1`, `e2` – переменные, `f5` – имя функции, а `<f5 e1>` обращение к этой функции.

Переменная языка Рефал включает:

признак_типа индекс

Признак типа записывается буквой `s`, `t` или `e` и определяет тип значений, которые может принимать эта переменная:

- `s` (или `S`) – значением переменной может быть только символ-литера;
- `t` (или `T`) – значением переменной может быть так называемый *терм* – символ-литера или выражение в структурных скобках;
- `e` (или `E`) – значением переменной может быть произвольное рефал-выражение (в том числе и пустое).

В качестве *индекса* переменной может выступать либо цифра, либо буква латинского алфавита. Фактически индекс является именем переменной и служит для её идентификации, поэтому, к примеру, `e1` и `e2` – разные переменные.

Если переменные и символы-литеры служат атомарными элементами рефал-выражений, то более крупной структурной единицей выражения является *терм*.

Атомарный элемент представляет простейший случай терма, а в общем случае ***терм*** – это рефал-выражение, взятое в структурные или функциональные скобки и называемое соответственно *структурным* или *функциональным термом*. Заметим, что функциональный терм есть по сути вызов функции.

Рефал-выражение (или просто – выражение) представляет собой последовательность термов, его синтаксис можно описать следующими правилами в форме Бэкуса-Наура (далее – БНФ-правила):

выражение ::= *пусто* | *терм выражение*

терм ::= *атомарный_элемент* | *структурный_терм* |
функциональный_терм

атомарный_элемент ::= *переменная* | *символ-литера*

структурный_терм ::= **(***выражение***)**

функциональный_терм ::= <*имя_функции выражение*>

пусто ::=

Отметим, что простейшим видом выражения является *пустое выражение*. Непустое выражение состоит из одного или нескольких термов, называемых *термами верхнего уровня*. Например, выражение `'A'('BE'('C')'D')` состоит на верхнем уровне из двух термов: символа-литеры `'A'` и структурного терма `('BE'('C')'D')`, выражение внутри которого в свою очередь состоит из четырёх вложенных термов (трёх символов-литер `'B'`, `'E'`, `'D'` и одного структурного терма `('C')`).

В зависимости от того, какие атомарные элементы и какие скобки используются при образовании выражения, различают следующие виды выражений:

- *объектное выражение* – выражение, не содержащее переменных и функциональных скобок, например: `'type'('int3 = 1..3')`;
- *выражение-образец* – выражение, не содержащее функциональных скобок, например: `e1 '+' e2 (sa e3)`;
- *рабочее выражение* – выражение без переменных, например: `'a-b' <pr '+c+g'>(<fun 'dg'>)`;
- *выражение общего вида*, в котором могут встретиться все виды скобок и атомарных элементов (далее – *рефал-выражение* или просто *выражение*).

Важно, что рефал-переменные могут иметь значением только объектные выражения. Объектным выражением должен быть и аргумент вычисляемого обращения к функции.

1.3. Функции и предложения

Каждая функция языка Рефал имеет только один аргумент, однако такое ограничение числа аргументов рефал-функций не снижает выразительной и вычислительной мощности языка. Действительно, аргументом функции является объектное выражение – последовательность из нескольких термов, которые могут рассматриваться как последовательные аргументы функции.

Обращение к рефал-функции имеет вид функционального терма, причём имя функции стоит внутри функциональных скобок:

<имя_функции выражение>

Определение рефал-функции состоит из нескольких *предложений* вида

выражение-образец = выражение

записываемых последовательно друг за другом по строкам. Каждое предложение включает *левую часть* (выражение-образец) и *правую часть* (рефал-выражение общего вида) и описывает правило преобразования и замены: выражение-образец характеризует заменяемое выражение, выражение в правой части определяет результат его замены. Подчеркнём, что в левой части предложения не могут встречаться обращения к функциям.

При определении функции, в начале её первого предложения, перед его левой частью записывается имя функции, которое отделяется от последующего выражения-образца одним или несколькими пробелами. Остальные предложения функции не нуждаются в именовании, но их запись на строке должна начинаться хотя бы с одного пробела.

Приведём пример функции, описываемой двумя предложениями. В нашем примере функции первое предложение описывает нужное преобразование цепочки символов (замену в ней знака +), а второе соответствует случаю, когда знак + в цепочке символов отсутствует, и она остаётся прежней.

```
* Функция ChangeFirstPlus заменяет в цепочке символов
* знак сложения (если он есть) на знак вычитания,
* например, 'A+B' будет преобразовано в 'A-B',
* а 'XY+Z85' – в 'XY-Z85'
ChangeFirstPlus   e1 '+' e2  =  e1 '-' e2
                  e1  =  e1
```

Порядок записи предложений в рефал-функции существенен, поскольку каждое предложение может быть применено только тогда, когда не применимы все предыдущие предложения.

Выражения-образцы в левой части предложений обычно содержат рефал-переменные. Эти переменные локальны, областью их действия является только то предложение функции, в котором они употребляются. Для нашего примера функции `ChangeFirstPlus` это означает, что значение переменной `e1` при использовании первого предложения может быть отлично от значения этой же переменной при использовании второго предложения.

Переменные получают свои значения при проверке применимости предложения, и эти значения используются для выполнения описанной в этом предложении замены. Тем самым, в правой части предложения могут быть использованы только те переменные, которые были использованы в левой части.

В общем случае может быть несколько вхождений одной и той же переменной как в левую, так и в правую часть предложения, например:

```
s1 e2 s1 e3 = s1 s1 e2 e3
```

Все вхождения одной и той же переменной (т.е. переменной с конкретным индексом) в левую и правую часть предложения должны иметь одинаковый признак типа. Тем самым недопустимы предложения вида `s1 e1 = e1` или `s1 e2 = e1`.

Приведём пример ещё одной рефал-функции:

```
* Функция ChangeAllPlus заменяет в цепочке символов
* все знаки сложения на знаки вычитания, например,
* 'A+B+C' будет преобразовано в 'A-B-C'
ChangeAllPlus e1 '+' e2  = <ChangeAllPlus e1 '-' e2>
              e1  =  e1
```

В отличие от функции `ChangeFirstPlus`, функция `ChangeAllPlus` *рекурсивна*: в правой части первого предложения стоит обращение к ней самой, рекурсия реализует последовательную, циклическую замену знаков сложения. Этот процесс заканчивается, когда в обрабатываемой цепочке символов не останется знаков +, и применяется второе предложение функции.

Использование рекурсии является одной из основных особенностей программирования на языке Рефал.

1.4. Абстрактная Рефал-машина

Программа на Рефале представляет собой несколько взаимосвязанных рефал-функций. Выполнение рефал-программы (её операционная семантика) описывается в терминах *абстрактной рефал-машины*, роль которой на практике играет *рефал-интерпретатор*.

Рефал-машина имеет два запоминающих устройства:

- ✓ ***поле зрения***, в котором содержится обрабатываемое рабочее выражение,
- ✓ ***поле памяти***, в котором хранится рефал-программа.

Перед началом работы рефал-машины в поле зрения заносится исходное символьное выражение, точнее, некоторый функциональный терм – обращение к функции, аргументом которой является это исходное выражение, например: `<ChangeAllPlus 'A+B+C'>`.

Работа рефал-машины разбивается на *шаги*, на каждом из которых применяется одно рефал-предложение, в результате чего некоторое выражение в поле зрения заменяется на другое. Таким образом, выражение, находящееся в поле зрения, последовательно, по шагам преобразуется. В ходе преобразований в поле зрения может содержаться сколь угодно много функциональных термов, в том числе – вложенных

один в другой. По окончании работы в поле зрения должно остаться объектное выражение (выражение без переменных и функциональных термов), рассматриваемое как результат вычислений. Опишем более подробно шаг работы рефал-машины, он включает следующие этапы:

I. Поиск в поле зрения заменяемого выражения – *ведущего функционального терма*. Ведущим функциональным термом является самый левый функциональный терм, не содержащий внутри себя других функциональных термов. Он представляет собой обращение к некоторой функции: `<f Et>`, `f` – имя функции, `Et` – объектное выражение (аргумент функции).

II. Поиск в поле памяти применимого предложения функции `f`. Для этого в группе предложений, определяющих функцию `f`, последовательным перебором, начиная с первого предложения, ищется такое предложение `El = Er`, левую часть `El` которого можно *синтаксически отождествить* с аргументом `Et` функционального обращения `<f Et>`.

    Говорят, что имеет место ***синтаксическое отождествление*** выражения-образца `El` с объектным выражением `Et`, если переменным из выражения-образца `El` можно задать такие допустимые значения (согласно их типу), что при подстановке в `El` этих значений вместо соответствующих переменных получается выражение, в точности совпадающее с объектным выражением `Et`.

    Если возможно синтаксическое отождествление `El` с `Et`, то соответствующее предложение функции `f` считается *применимым*. Это предложение вместе с найденными в процессе синтаксического отождествления значениями переменных передаётся на следующий этап.

III. Применение найденного предложения функции `f`. При этом в поле зрения производится замена найденного на этапе I ведущего функционального терма `<f Et>` на выражение `Er'`. Выражение `Er'` формируется из правой части `Er` применяемого предложения подстановкой вместо всех переменных их значений, найденных в результате синтаксического отождествления.

По окончании этапа III рефал-машина вновь переходит к этапу I и все описанные этапы повторяются, но уже с изменённым содержимым поля зрения.

Возможны два случая останова рефал-машины – нормальный и аварийный. *Нормальный останов* происходит в случае, когда на этапе I в

поле зрения нет функциональных термов (функциональных обращений). Рефал-машина сообщает, что вычисление окончено и останавливается, результатом её работы считается выражение, оставшееся в поле зрения.

Аварийный останов рефал-машины происходит тогда, когда для найденного на этапе I ведущего функционального терма на этапе II не обнаружено ни одного применимого предложения. В этом случае рефал-машина останавливается с сообщением «отождествление невозможно». Такой останов означает, что либо программа на Рефале, либо исходные данные для неё были заданы неверно.

Отметим, что аварийный останов рефал-машины может произойти и в том случае, когда исчерпана память, отводимая под поле зрения, обычно это происходит в результате зацикливания рекурсивных рефал-программ.

Рассмотрим примеры работы рефал-машины.

Пусть в поле памяти рефал-машины содержится рассмотренное выше описание функции `ChangeFirstPlus`, а в поле зрения помещено выражение `<ChangeFirstPlus 'XY+Z85'>`. На первом шаге работы рефал-машины к этому функциональному терму, являющемуся ведущим, будет применено первое предложение функции `ChangeFirstPlus`, поскольку выражение `'XY+Z85'` отождествимо с образцом `e1 '+' e2` при значениях переменных `e1` ↔ `'XY'`, `e2` ↔ `'Z85'`. В результате в поле зрения рефал-машины останется выражение `'XY-Z85'`, которое не содержит функциональных термов, и поэтому на втором шаге рефал-машина остановится, а полученное выражение – окончательный результат её работы.

Пусть теперь в поле памяти кроме определения функции `ChangeFirstPlus` содержатся определения двух функций:

```
Num1     = '731'
Num2     = '+1230'
```

В обоих определениях по одному предложению, в левой части каждого предложения – пустое выражение.

Пусть в поле зрения рефал-машины помещено выражение

```
<ChangeFirstPlus <Num1> <Num2>>
```

Тогда на первом шаге работы рефал-машины ведущим будет терм `<Num1>`, который заменится на `'731'`, поскольку аргумент этого функционального вызова – пустое выражение, а значит, применимо единственное предложение функции `Num1`. В результате замены поле зрения примет вид:

```
<ChangeFirstPlus '731' <Num2>>
```

Теперь ведущим становится терм <Num2>, и в конце второго шага работы в поле зрения появится выражение

```
<ChangeFirstPlus '731+1230'>
```

Это выражение содержит только один функциональный терм, поэтому на следующем шаге применяется первое предложение функции ChangeFirstPlus при e1 ↔ '731', e2 ↔ '1230'. В поле зрения остаётся выражение '731-1230', которое уже не содержит функциональных скобок. Поэтому на четвёртом шаге рефал-машина остановится с результатом '731-1230' в поле зрения.

1.5. Правила синтаксического отождествления

Синтаксическое отождествление выражения-образца и объектного выражения является наиболее сложной операцией, выполняемой рефал-машиной. Отождествление означает, что объектное выражение является частным случаем выражения-образца, как в рассмотренных выше и приводимых ниже примерах.

Выражение-образец 'for ' e1 '=' e2 'to' e3 'do' e4 отождествимо со строкой литер 'for i=1 to 12 do n:=n+1' при значениях переменных: e1 ↔ 'i', e2 ↔ '1 ' (значение e2 – два символа, знак единицы и знак пробела), e3 ↔ ' 12 '(значение e3 – четыре символа: пробел, 1, 2, пробел), e4 ↔ ' n:=n+1'.

Выражение-образец s1 e2 ')' e3 отождествимо со строкой 'A*(B+C)-D' при значениях переменных: s1 ↔ 'A', e2 ↔ '*(B+C', e3 ↔ '-D' (скобка в выражении-образце не является структурной, в нём указана литера круглой скобки).

Выражение-образец e1 '+' t2 e3 синтаксически отождествимо со строкой литер 'X-5*Y+8/Z' при таких значениях переменных: e1 ↔ 'X-5*Y', t2 ↔ '8', e3 ↔ '/Z'.

Сформулируем общие правила выполнения синтаксического отождествления. Синтаксическое отождествление выражения-образца и объектного выражения выполняется последовательно по входящим в них атомарным элементам и всем парам структурных скобок. При этом должны выполняться следующие условия:

1. Символ-литера синтаксически отождествим только с таким же символом-литерой, например, 'g' отождествим с 'g'.
2. Переменная отождествима только с выражением, вид и структура которого соответствует типу переменной. В частности, e-переменная отождествима с любым рефальским выражением, в том числе пустым;

t-переменная отождествима с термом; s-переменная отождествима с любым символом-литерой.

3. Все вхождения в выражение-образец одной и той же переменной должны получить одно и то же значение. Поэтому, к примеру, выражение-образец `s1 e2 s1` отождествимо со строкой `'abca'` (`s1` ↔ `'a'`, `e2` ↔ `'bc'`), но не со строкой `'abcd'`.
4. При отождествлении левых структурных скобок выражений должны отождествляться и соответствующие (парные к ним) правые структурные скобки этих выражений. Это означает, что при отождествлении баланс структурных скобок не может быть нарушен.

Реализованный в рефал-интерпретаторе алгоритм, выполняющий синтаксическое отождествление выражений и в случае успеха определяющий значения употреблённых в выражении-образце переменных, называется *алгоритмом проектирования* выражения-образца *на* объектное выражение. Для написания эффективных рефал-программ необходимо понимать важные особенности этого алгоритма.

В общем случае выражение-образец содержит так называемые *жёсткие элементы* – элементы, которые однозначно проектируются на соответствующие элементы объектного выражения. Символы-литеры, структурные скобки, s- и t-переменные – это жёсткие элементы выражения, и если у такого элемента спроектирован один конец, то однозначно проектируется и второй, либо делается вывод о невозможности дальнейшего отождествления. Для символов-литер, структурных скобок и s-переменных это очевидно. Для t-переменных возможная проекция – это либо символ (тоже очевидный случай), либо выражение в структурных скобках (но в этом случае однозначно находится парная скобка). На всех шагах проектирования действует общий принцип: если спроектирован один конец жёсткого элемента, то делается попытка спроектировать и его второй конец, т.е. весь элемент в целом.

Например, выражение-образец `'A'(e1 t2)s3` будет отождествляться с объектным выражением `'A'(('2B'))'7'` следующим образом (см. Рис. 1):

1. символ-литера `'A'` отождествится с символом-литерой `'A'`;
2. первая открывающаяся структурная скобка из объектного выражения отождествится с открывающейся скобкой из выражения-образца;
3. парная ей последняя закрывающаяся скобка из объектного выражения отождествится с закрывающейся структурной скобкой из выражения-образца;

4. переменная `t2` выражения-образца отождествится со структурным термом **(**`'2B'`**)**;
5. переменная `e1` отождествится с пустым выражением;
6. переменная `s3` отождествится с символом-литерой `'7'`.

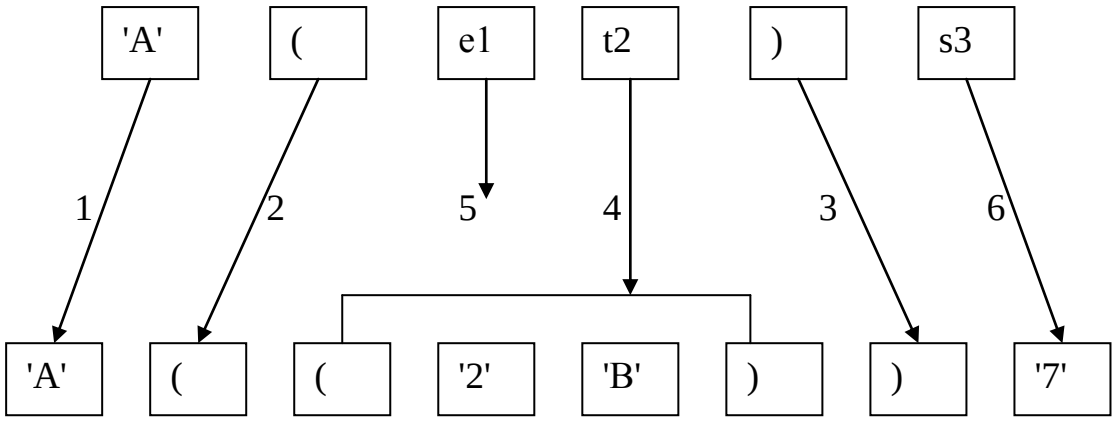

Рисунок 1. Пример отождествления

Рассмотрим дополнительные примеры.

Выражение-образец `s1`**(**`e2 ',' e3`**)** синтаксически отождествим с объектным выражением `'c'`**(**`'ga,d'`**)**, при значениях переменных `s1` ↔ `'c'`, `e2` ↔ `'ga'`, `e3` ↔ `'d'`. Образец `sx ty` отождествим с тем же объектным выражением при `sx` ↔ `'c'`, `ty` ↔ **(**`'ga,d'`**)**.

Выражения `')'` и `'('` являются обычными символами-литерами и могут быть значениями `s`-переменных, в отличие от структурных скобок. Баланс символов-литер скобок при отождествлении не проверяется. Поэтому единственным вариантом отождествления образца **(**`e1`**(**`s2'('`**))** с объектным выражением **(**`'a)'`**(**`'b('`**))** являются значения переменных `e1` ↔ `'a)'` и `s2` ↔ `'b'`.

Можно заметить, что в ряде случаев при использовании в выражении-образце нескольких `e`-переменных синтаксическое отождествление возможно для нескольких вариантов приписывания этим переменным значений. К примеру, при отождествлении образца `e1'+'e2` и выражения `'a+b+c'` возможны два варианта: `e1` ↔ `'a'`, `e2` ↔ `'b+c'` и `e1` ↔ `'a+b'`, `e2` ↔ `'c'`. Возникновение таких неоднозначностей связано с тем, что `e`-переменные не ограничиваются по

длине, их значениями могут быть произвольные последовательности термов.

Ясно, что алгоритм проектирования должен устранять неоднозначности такого типа в случае их возникновения. Возможны два способа, которые называются соответственно отождествление слева направо и отождествление справа налево, или ***левое*** и ***правое согласование.***

При левом согласовании рефал-машина выбирает тот вариант отождествления (вариант приписывания переменным значений), при котором первая слева `e`-переменная выражения-образца принимает самое короткое возможное для неё значение. Если это не устраняет неоднозначности, то такой же отбор производится по второй слева `e`-переменной, затем третьей слева и т.д.

При правом согласовании рефал-машина выбирает тот вариант отождествления, при котором первая справа `e`-переменная принимает самое короткое возможное значение. Если же это не снимает неоднозначности, то такой же отбор производится по второй справа `e`-переменной, затем третьей справа и т.д.

В нашем примере первый вариант отождествления соответствует левому согласованию, а второй – правому. По умолчанию рефал-машина реализует левое согласование.

Заметим, что при отождествлении образца `e1 e2` с тем же выражением `'a+b+c'` при левом согласовании переменные получат значения: `e1` ↔ *пусто*, `e2` ↔ `'a+b+c'`; а при правом: `e1` ↔ `'a+b+c'`, `e2` ↔ *пусто*.

Рассмотрим вновь функцию `ChangeAllPlus`, определённую в разделе 1.3, и приведём пример её работы.

Пусть в поле зрения рефал-машины помещено выражение `<ChangeAllPlus 'A+B+C'>`. На первом шаге работы рефал-машины к этому функциональному терму применимо первое предложение функции `ChangeAllPlus`, поскольку выражение `'A+B+C'` отождествимо с образцом `e1 '+' e2` при `e1` ↔ `'A'`, `e2` ↔ `'B+C'` (по умолчанию используется левое согласование). В результате в поле зрения рефал-машины появится выражение

```
<ChangeAllPlus 'A-B+C'>
```

Поскольку это выражение содержит только один функциональный терм, он будет ведущим, и к входящему в него выражению `'A-B+C'` на втором шаге работы рефал-машины будет вновь применено первое предложение функции `ChangeAllPlus`, но уже при значениях переменных

e1 ↔ 'A-B', e2 ↔ 'C'. Поле зрения после выполнения этого шага примет вид: <ChangeAllPlus 'A-B-C'>

На третьем шаге работы рефал-машины единственный (и ведущий) функциональный терм заменится на выражение 'A-B-C', так как применимым будет только второе предложение функции при e1 ↔ 'A-B-C' (в аргументе функционального обращения нет символа +).

На четвёртом шаге произойдёт останов рефал-машины, поскольку в поле зрения нет функциональных термов, а оставшееся в нём выражение 'A-B-C' является результатом вычисления функционального вызова <ChangeAllPlus 'A+B+C'>

1.6. Примеры рефал-функций

Завершая описание базисного Рефала, отметим, что программирование на нём имеет ряд характерных особенностей, присущих функциональному программированию. Центральную роль в нём играют функции: в виде функции оформляются все содержательные действия программируемого алгоритма, а нужная последовательность этих действий достигается вызовом функций в определённом порядке, т.е. суперпозицией этих функций. Вместо традиционных циклов императивных языков программирования в Рефале применяется более мощное средство – *рекурсия*.

Другая характерная особенность языка Рефал – отсутствие оператора присваивания. Роль рефал-переменных в корне отлична от их роли в традиционных императивных языках программирования. Переменные могут получить значения только в процессе синтаксического отождествления, и эти значения могут быть использованы только в пределах одного рефал-предложения.

Рассмотрим теперь несколько рефал-функций разной степени сложности.

Пример 1: Вернёмся к определению функции ChangeAllPlus, приведённому в конце раздела 1.3:

```
ChangeAllPlus e1 '+' e2  = <ChangeAllPlus e1 '-' e2>
              e1  =  e1
```

Функция реализует замену во входной цепочке символов всех знаков + на знаки −. Поскольку по умолчанию используется левое согласование, то в случае, когда применимо первое предложение (в обрабатываемой цепочке символов есть знак +), переменная e1 получит самое короткое возможное значение, и знак + в него входить не будет. Поэтому целесообразно оставить внутри функциональных скобок рекурсивного

вызова лишь переменную e2 – только её значение может содержать знаки +, подлежащие замене. Таким образом, получаем следующее определение функции:

```
* Функция ChangeAllPlus заменяет в цепочке символов
* все символы сложения на символы вычитания,
* например: 'A+B+C' будет преобразовано в 'A-B-C'
ChangeAllPlus  e1 '+' e2 = e1 '-' <ChangeAllPlus e2>
               e1  =  e1
```

Так определённая функция выполняет преобразование цепочки символов более эффективно, т.к. на каждом шаге работы рефал-машины синтаксическое отождествление применяется ко всё более короткой цепочке символов.

Например, при вычислении функционального вызова

```
<ChangeAllPlus 'AX+BY+CZ'>
```

на первом шаге работы рефал-машины к нему применимо первое предложение функции ChangeAllPlus, так как выражение 'AX+BY+CZ' отождествимо с образцом e1 '+' e2 при e1 ↔ 'AX', e2 ↔ 'BY+CZ'. После применения этого предложения в поле зрения рефал-машины появится выражение

```
'AX-'<ChangeAllPlus 'BY+CZ'>
```

Поскольку это выражение содержит только один функциональный терм, он будет ведущим, и к входящему в него выражению 'BY+CZ' на втором шаге работы рефал-машины будет вновь применено первое предложение функции ChangeAllPlus, но уже при значениях переменных e1 ↔ 'BY', e2 ↔ 'CZ'. Поле зрения после выполнения этого шага примет вид:

```
'AX-BY-'<ChangeAllPlus 'CZ'>
```

На третьем шаге работы рефал-машины единственный (и ведущий) функциональный терм заменится на выражение 'CZ', поскольку применимым будет только второе предложение функции при e1 ↔ 'CZ' (в аргументе функционального терма нет символа +). В поле зрения останется выражение 'AX-BY-CZ', которое уже не содержит функциональных термов и является результатом вычисления исходного функционального вызова <ChangeAllPlus 'AX+BY+CZ'>.

Пример 2: функция Palindrom, которая проверяет, является ли входная цепочка символов палиндромом. *Палиндромом* является цепочка символов, читаемая одинаково как справа налево, так и слева направо, например, «авва», «шалаш», строка «а роза упала на лапу азора», если в ней убрать пробелы.

Функция содержит 4 предложения и, по сути, является предикатом (логической функцией) – в зависимости от результата проверки она вырабатывает цепочку литер 'yes' или 'no'.

```
* Функция Palindrom, проверяющая, является ли
* входная строка палиндромом
Palindrom    s1 e2 s1  =  <Palindrom  e2>
             s1  =  'yes'
                 =  'yes'
             e1  =  'no'
```

Эта функция рекурсивна: если входная цепочка символов начинается и кончается одним и тем же символом s1, то процесс проверки продолжается для e2 – оставшейся части цепочки. Рекурсивным является только первое предложение функции Palindrom. Следующие два предложения служат для завершения рекурсии.

Если цепочка символов состоит из единственного символа (второе предложение функции), то она является палиндромом (это предложение срабатывает в случае, когда исходная цепочка-палиндром состояла из нечётного количества символов).

Пустое выражение также считается палиндромом (третье предложение функции, оно срабатывает, если исходная цепочка-палиндром состояла из чётного количества символов).

Если же ни одно из первых трёх предложений функции Palindrom не может быть применено, то исходное выражение не является палиндромом – это фиксируется в последнем предложении функции.

Заметим, что последнее предложение функции Palindrom (любое выражение заменить на 'no') применимо всегда, поэтому оно не может быть переставлено с другими предложениями. В то же время первые три предложения могут быть поставлены в любом порядке, поскольку их области применимости не пересекаются: образец первого предложения может сопоставиться только с цепочками, состоящими не менее чем из двух символов.

Рассмотрим пошаговую работу рефал-машины при вычислении функционального обращения <Palindrom 'шалаш'>. На первом шаге работы рефал-машины применится первое предложение функции Palindrom, значения переменных: s1 ↔ 'ш', e2 ↔ 'ала'. В результате в поле зрения появится выражение <Palindrom 'ала'>.

На втором шаге также применится первое предложение функции, переменные получат следующие значения: s1 ↔ 'а', e2 ↔ 'л', а содержимое поля зрения примет вид: <Palindrom 'л'>.

На третьем шаге первое предложение уже не может быть применено, применится второе предложение, значение переменной: s1 ↔ 'л'. В результате применения этого предложения функциональный терм <Palindrom 'л'> заменится в поле зрения цепочкой символов 'yes'.

На четвёртом шаге произойдёт останов рефал-машины, поскольку в поле зрения больше нет функциональных термов. Оставшаяся в поле зрения цепочка 'yes' – результат вычисления исходного функционального вызова <Palindrom 'шалаш'>.

Пример 3: функция DelRepSymb, удаляющая в исходной цепочке символов повторные вхождения каждого символа, т.е. из нескольких вхождений одного символа она должна оставить только одно. Тем самым, в результирующей цепочке все символы попарно различны.

```
* Функция DelRepSymb удаляет повторные вхождения
* символов, оставляя только первое вхождение
* каждого символа.
* Например, 'abaacebf' будет преобразовано в 'abcef'
DelRepSymb e1 sA e2 sA e3  = e1 <DelRepSymb sA e2 e3>
          e1  =  e1
```

Функция рекурсивна и завершает свою работу только тогда, когда первое, рекурсивное предложение не применимо, т.е когда в обрабатываемом выражении не осталось одинаковых символов. Причём каждое применение первого предложения удаляет из обрабатываемого выражения одно повторное вхождение символа.

Также, как и в примере 1, при построении рекурсивного предложения используется тот факт, что значение переменной e1 не может содержать символов, входящих в выражение более одного раза (поскольку по умолчанию согласование левое), а значит, e1 можно не включать в аргумент рекурсивного вызова. В то же время, в этом аргументе, в его начале содержится sA – именно это обеспечивает сохранение в результирующей цепочке символов первого вхождения каждого символа. В аргументе рекурсивного вызова также стоят переменные e2 и e3, в значениях которых могут быть повторные вхождения как sA, так и других символов.

Заметим, что поскольку второе предложение функции применимо всегда, оно помещено последним.

Рассмотрим работу рефал-машины при вычислении функционального вызова <DelRepSymb 'abaacebf'>. На первом шаге применяется первое предложение функции DelRepSymb, причём в результате отождествления образца e1 sA e2 sA e3 с выражением

'abaacebf' переменные получают следующие значения: e1 ↔ *пусто,* sA ↔ 'a', e2 ↔ 'b', e3 ↔ 'acebf'. После выполнения этого шага в поле зрения рефал-машины появится выражение <DelRepSymb 'abacebf'>.

На втором шаге также применяется первое предложение функции, при этом значения переменных: e1 ↔ *пусто,* sA ↔ 'a', e2 ↔ 'b', e3 ↔ 'cebf'. Содержимое поля зрения после выполнения второго шага: <DelRepSymb 'abcebf'>.

На третьем шаге опять применяется первое предложение, на этот раз значения переменных: e1 ↔ 'a', sA ↔ 'b', e2 ↔ 'ce', e3 ↔ 'f', и итоговое поле зрения: 'a'<DelRepSymb 'bcef'>.

На четвёртом шаге первое предложение уже не может быть применено, поскольку в аргументе функции все символы попарно различны. Применяется второе, заключительное предложение функции DelRepSymb, и в поле зрения остаётся выражение 'abcef' без функциональных термов. Эта строка и является результатом вычисления исходного функционального вызова.

Другим возможным вариантом удаления повторных вхождений символов является такое определение функции:

```
* Функция DelRepSymb2 удаляет повторные вхождения
* символов, оставляя только последнее вхождение
* каждого символа
* Например, 'abaacebf' будет преобразовано в 'acebf'
DelRepSymb2 e1 sA e2 sA e3 = e1<DelRepSymb2 e2 sA e3>
            e1  =  e1
```

В этом варианте при удалении повторных символов в рекурсивном вызове сохраняется второе их вхождение в цепочку, тем самым в итоге остаётся только последнее вхождение каждого символа, и результатом вычисления функционального вызова <DelRepSymb2 'abaacebf'> будет строка 'acebf'.

Пример 4: функция, осуществляющая зеркальное отображение исходной цепочки символов, т.е. изменение порядка символов в ней на противоположный: например, из цепочки 'abcde' получается цепочка 'edcba'.

```
* Функция RevString зеркально
* отображает (реверсирует) цепочку символов
RevString   s1 e2  =  <RevString e2> s1
                   =
```

Основное преобразование делает первое предложение функции, оно расщепляет цепочку на первый символ `s1` и остаток `e2`, и помещает `s1` в конец цепочки, получающейся в результате реверсирования `e2`. Второе предложение (*пусто* заменить на *пусто*) служит для завершения рекурсии. Предложения можно поменять местами, поскольку области их применимости не пересекаются.

Рассмотрим вычисление функционального обращения `<RevString 'abc'>`. На первом шаге применяется первое предложение функции (значения переменных: `s1` ↔ `'a'`, `e2` ↔ `'bc'`) и в поле зрения появляется выражение: `<RevString 'bc'>'a'`. Первое предложение применяется и на втором (`s1` ↔ `'b'`, `e2` ↔ `'c'`) и на третьем шаге (`s1` ↔ `'c'`, `e2` ↔ *пусто*), а в поле зрения появляются соответственно выражения `<RevString 'c'>'ba'` и `<RevString >'cba'`. На четвёртом шаге работы рефал-машины применится второе предложение функции, поскольку аргумент функционального вызова – пустое выражение, и в поле зрения останется строка `'cba'`, являющаяся окончательным результатом вычислений.

Функция `RevString` зеркально отображает выражения, состоящие только из символов-литер. Для реверсирования выражений, в которых есть структурные скобки, потребуется написать более сложную функцию.

Пример 5: функция, зеркально отображающая произвольное рефал-выражение на всех его уровнях. К примеру, зеркальным отображением выражения `'a'('b'('cd'))('ef')` будет выражение `('fe')(('dc')'b')'a'`.

```
* Функция RevExp зеркально отображает
* структурированное выражение на всех его уровнях
RevExp  s1 e2  =  <RevExp e2> s1
        (e1) e2  =  <RevExp e2> (<RevExp e1>)
                 =
```

Первое предложение функции `RevExp` срабатывает в том случае, если в начале выражения стоит символ – этот символ `s1` переносится в конец, а остаток выражения `e2` зеркально отображается рекурсивным вызовом `RevExp`. Второе предложение содержит 2 рекурсивных вызова: один нужен для реверсирования содержимого структурного терма, а другой – для реверсирования остатка выражения. Третье предложение функции `RevExp` завершает рекурсию.

Рассмотрим пошаговую работу рефал-машины при вычислении функционального вызова `<RevExp 'a'('b'('cd'))('ef')>` (исходное содержимое поля зрения рефал-машины).

Шаг 1. Ведущий функциональный терм:

`<RevExp 'a'('b'('cd'))('ef')>`

Применяется первое предложение функции `RevExp`. При этом значения переменных: `s1` ↔ `'a'`, `e2` ↔ `('b'('cd'))('ef')`. Подставляя эти значения в правую часть применяемого предложения, получаем в поле зрения выражение `<RevExp ('b'('cd'))('ef')> 'a'`.

Шаг 2. Ведущий функциональный терм –

`<RevExp ('b'('cd'))('ef')>`

Применяется второе предложение функции (т.к. аргумент этого терма начинается со структурной скобки), значения переменных: `e1` ↔ `'b'('cd')`, `e2` ↔ `('ef')`. Подстановка этих значений в правую часть применяемого предложения и замена ею ведущего терма даёт следующее выражение в поле зрения:

`<RevExp ('ef')> (<RevExp 'b'('cd')>) 'a'`

Шаг 3. Ведущий функциональный терм – `<RevExp ('ef')>`

Применимо второе предложение функции, значения переменных: `e1` ↔ `'ef'`, `e2` ↔ *пусто*. Ведущий терм заменяется на `<RevExp >` `(<RevExp 'ef'>)`, и содержимое поля зрения после выполнения шага:

`<RevExp > (<RevExp 'ef'>)(<RevExp 'b'('cd')>) 'a'`

Шаг 4. Ведущий функциональный терм – `<RevExp >`

Применимо третье предложение функции `RevExp`, и ведущий терм заменяется на пустое выражение. Содержимое поля зрения рефал-машины после выполнения шага:

`(<RevExp 'ef'>)(<RevExp 'b'('cd')>) 'a'`

Шаг 5. Ведущий функциональный терм – `<RevExp 'ef'>`

Применяется первое предложение функции со значениями переменных `s1` ↔ `'e'`, `e2` ↔ `'f'`. Терм заменяется на `<RevExp 'f'>'e'`, и содержимое поля зрения:

`(<RevExp 'f'>'e')(<RevExp 'b'('cd')>) 'a'`

Шаг 6. Ведущий функциональный терм – `<RevExp 'f'>`

Применимо первое предложение, значения переменных: `s1` ↔ `'f'`, `e2` ↔ *пусто*. Замена терма на `<RevExp >'f'`, и содержимое поля зрения после выполнения шага:

`(<RevExp >'fe')(<RevExp 'b'('cd')>) 'a'`

Шаг 7. Ведущий функциональный терм: `<RevExp >`
Применяется третье предложение функции, и ведущий функциональный терм заменяется на пустое выражение. В поле зрения рефал-машины появляется выражение **(**`'fe'`**)(**`<RevExp 'b'`**(**`'cd'`**)>`**)** `'a'`

Шаг 8. Ведущий функциональный терм: `<RevExp 'b'`**(**`'cd'`**)>`
Применимо первое предложение функции `RevExp`, значения переменных: `s1` ↔ `'b'` и `e2` ↔ **(**`'cd'`**)**. Терм заменяется на `<RevExp`**(**`'cd'`**)>'b'`, что даёт в поле зрения **(**`'fe'`**)(**`<RevExp`**(**`'cd'`**)> 'b'`**)** `'a'`

Шаг 9. Ведущий функциональный терм: `<RevExp`**(**`'cd'`**)>`
Применяется второе предложение функции со значениями переменных `e1` ↔ `'cd'`, `e2` ↔ *пусто*. Замена терма на `<RevExp>`**(**`<RevExp 'cd'>`**)**, содержимое поля зрения:
(`'fe'`**)(** `<RevExp>`**(**`<RevExp 'cd'>`**)**`'b'`**)** `'a'`

Шаг 10. Ведущий функциональный терм – `<RevExp >`
Применимо третье предложение: *пусто* заменить на *пусто*, и содержимое поля зрения рефал-машины: **(**`'fe'`**)((**`<RevExp 'cd'>`**)**`'b'`**)** `'a'`

Шаг 11. Ведущий функциональный терм: `<RevExp 'cd'>`
Применяется первое предложение функции, значения переменных: `s1` ↔ `'c'`, `e2` ↔ `'d'`. Замена терма на `<RevExp 'd'> 'c'`, и в поле зрения после выполнения шага появляется выражение:
(`'fe'`**)((**`<RevExp 'd'>'c'`**)**`'b'`**)** `'a'`

Шаг 12. Ведущий функциональный терм: `<RevExp 'd'>`
Применимо первое предложение со значениями переменных `s1` ↔ `'d'`, `e2` ↔ *пусто*. Терм заменяется на `<RevExp >'d'`, и содержимое поля зрения: **(**`'fe'`**)((**`<RevExp> 'dc'`**)**`'b'`**)** `'a'`

Шаг 13. Ведущий функциональный терм: `<RevExp >`
Применимо третье предложение функции, и содержимое поля зрения рефал-машины после выполнения шага: **(**`'fe'`**)((**`'dc'`**)**`'b'`**)**`'a'`

Шаг 14. Поскольку в поле зрения не осталось функциональных термов, то рефал-машина останавливается с результатом `'a'`**(**`'b'`**(**`'cd'`**))(**`'ef'`**)**.

2. Язык Рефал-2

Данный раздел посвящён первому известному диалекту языка Рефал, реализованному в своё время для отечественных ЭВМ БЭСМ-6 и ЕС ЭВМ. Язык Рефал-2 является расширением базисного Рефала, и в этом разделе не только уточняется синтаксис и семантика конструкций базисного Рефала, но и подробно описывается ряд возможностей Рефала-2 (прежде всего – спецификаций переменных), не имеющих аналогов в других диалектах этого языка. Средства языка Рефал-2, не рассмотренные в настоящем пособии (в частности, статические и динамические ящики), описаны в [8].

2.1. Символы-литеры и составные символы

Собственными знаками языка Рефал-2 являются:

- функциональные скобки `< >`
- структурные скобки **( )**
- знаки `/ ' : = +`
- буквы `S W V E s w v e R L`

Понятие символа было расширено в языке Рефал-2. Кроме собственных знаков и *символов-литер* (`'z'`, `'+'`, `'7'` и т.п.) используются *составные символы*. Составные символы служат для записи в программе сложных знаков, состоящих из нескольких литер, но в тоже время интерпретирующихся как единое целое. В частности, составные символы отождествляются с `s`-переменными. Для отличия составных символов от цепочки объектных знаков составные символы обрамляются знаками `/`, например: `/alpha/`, `/178/`, `/abcd/`. Внутри знаков косой черты не может встречаться знак `/`.

Множество составных символов разбивается на *символы-метки* и *символы-числа*.

Символами-числами или *макроцифрами* являются последовательности десятичных цифр, представляющие целые неотрицательные числа в диапазоне от 0 до 16777215 (2^{24}-1), заключённые в знаки `/`. Например: `/0/`, `/1/`, `/512/`, `/23/`. Символы-числа служат для записи в рефал-программе целых чисел, для их обработки в языке Рефал-2 есть набор встроенных функций, реализующих целочисленную арифметику. Знак числа, если он необходим, должен быть записан перед символ-числом как самостоятельный объектный знак. К примеру, отрицательное число −38 записывается как два отдельных символа:

`'-'/38/`. Таким образом, в языке Рефал-2 знак не является составной частью символ-числа.

Для записи больших чисел (бóльших, чем 2^{24}), используется последовательность *макроцифр*, играющих роль цифр в системе счисления по основанию 2^{24}. Например, запись `/1//0/` представляет число 2^{24}, т.е. число 16777216, а запись `/3//21//35/` представляет число $3\cdot(2^{24})^2+21\cdot(2^{24})^1+35\cdot(2^{24})^0=844425282453539$.

Символами-метками являются идентификаторы, т.е. последовательности из букв, цифр и знаков `-`, начинающиеся с буквы. Длина символа-метки не ограничивается, однако принимаются во внимание только первые 255 литер, а все последующие игнорируются. Таким образом, все символы-метки, у которых совпадают первые 255 литер, считаются совпадающими. Примеры символов-меток: `/aLpha/`, `/L2a4/`, `/это-пример-символа-метки/`.

Символы-метки позволяют ввести и использовать в рефал-программе новые символы, отличные от символов-литер. Чаще всего символы-метки используются в качестве мнемоничных обозначений для отдельных частей обрабатываемого символьного выражения, поскольку они отличны от любого объектного знака. Приведём пример такого использования меток в рефал-выражении:

```
(/const/ 'π=3.14, one=1')(/type/ 'i, n: integer').
```

В языке Рефал-2 имена рефал-функций также являются символами-метками, но при записи имён функций в функциональных обращениях знаки `/` могут быть опущены, так что запись `<func 'get+2+3'>` эквивалентна записи `</func/ 'get+2+3'>`. При необходимости в такой сокращённой записи после имени функции вставляется разделяющий пробел: обращение `</func/ sx>` сокращённо записывается как `<func sx>`, а не как `<funcsx>` (поскольку последнее интерпретируется как обращение к функции с именем `funcsx`). В тоже время в сокращённой записи пробел слева от имени функции не требуется, и имя функции обычно записывается сразу после левой функциональной скобки.

Обратим внимание ещё на одну особенность Рефала-2. При описании переменных их тип можно задавать как строчной, так и прописной буквой (`s` или `S`, `w` или `W`, `v` или `V`, `e` или `E`). Регистр букв в теле символа-метки также не важен. Например, если в программе встречаются символы-метки `/ab/`, `/Ab/`, `/aB/`, `/AB/`, то все они будут восприняты рефал-системой как одна и та же символ-метка `/AB/`. Поскольку имена функций – это также символы-метки, регистр букв при написании имён функций также не важен. Таким образом, при записи программы необходимо учитывать регистр букв только в двух случаях: при описании символов-литер

(например, символы-литеры 'A' и 'a' являются разными знаками), и при описании индексов переменных: переменные SX и Sx – это разные переменные, в то же время SX и sX – это одна и та же переменная.

Как и в базисном Рефале, апостроф является собственным знаком языка, используемым для записи символов-литер. В то же время он сам может быть литерой, которую необходимо уметь обрабатывать. Поэтому укажем особенности записи апострофа как объектного знака. Два идущих подряд апострофа '' означают символ-литеру апострофа (попутно заметим, что похожее выражение ' ' обозначает не апостроф, а литеру пробела). К примеру, цепочка символов A'C записывается как 'A''C', цепочка 'A'B записывается как '''A''B', а цепочка A'B' – как 'A''B'''. Для записи нескольких подряд идущих апострофов каждый из них должен быть удвоен, но дополнительными знаками апострофа такая цепочка не обрамляется, например, цепочка '' записывается как ''''.

Цепочку литер можно разбить на несколько подцепочек при помощи пробела, например, 'XY' '+Z3' означает цепочку из пяти литер XY+Z3, а запись 'XY''+Z3' обозначает цепочку литер XY'+Z3.

Часто одну и ту же цепочку символов-литер можно изобразить многими способами. Например, цепочка литер A'B может быть представлена любым из следующих способов: 'A' '' 'B', 'A''' 'B', 'A' '''B' или 'A''B'

Для наглядности в рефал-программу можно вставлять любое количество дополнительных пробелов между переменными, структурными и функциональными скобками, составными символами и цепочками литер.

2.2. Переменные и их спецификации

Переменная языка Рефал-2 имеет вид:

признак_типа [*спецификация*] *индекс*

Индекс переменной – это цифра или буква русского или латинского алфавита. Спецификация может отсутствовать. Признаком типа служат следующие буквы: s и S, w и W, e и E, v и V (соответствующие буквы верхнего и нижнего регистра являются эквивалентными). Например, S-переменные: S1, S2 и sA; W-переменные: W1 и Wx; V-переменные: v9 и VZ; E-переменные: E1, e5 и EA. Во всех этих переменных спецификация пуста. При записи переменных пробелы между признаком типа и спецификацией, а также спецификацией и индексом недопустимы.

По типу значения переменные разбиваются на:

- S-*переменные*, значением которых может быть только символ-литера, символ-метка или символ-число;
- W-*переменные*, значением которых может быть только символ или структурный терм;
- E-*переменные*, значением которых может быть произвольное объектное выражение (в том числе и пустое).
- V-*переменные*, значением которых может быть только непустое объектное выражение;

К примеру, для S-переменной в качестве значения допустимы следующие символьные выражения: `'G'`, `'+'`, `/172/`, `/alpha-beta/`. В то же время её значением не могут быть выражения `'172'`, `'-'/172/`, `('a+b')`, поскольку первые два выражения представляют собой цепочки соответственно из трёх и двух символов, а третье выражение – структурный терм. В качестве значения W-переменной допустимы, например, выражения `'G'`, `'+'`, `/172/`, `/alpha/`, **(**`'a+b'`**)**, но недопустимы выражения `'172'`, `'-'/172/`, `'X'`**(**`'a+b'`**)**, поскольку первые два есть цепочки символов, а третье состоит из двух термов.

V- и E-переменные, называемые также *переменными типа выражение*, могут иметь в качестве значения произвольное объектное выражение, однако пустое выражение допустимо в качестве значения только для E-переменных. Заметим, что пустое выражение не может быть значением и W-переменной.

Спецификация переменной в языке Рефал-2 представляет собой мощное выразительное средство, позволяющее гибко ограничивать множество объектных выражений, которые могут быть значениями переменной. Спецификация бывает двух видов – либо непосредственное задание ограничений на значения переменной, либо указание имени спецификатора, определяющего множество допустимых значений:

спецификация ::= (*набор_ограничений*) | *спецификатор*

Рассмотрим сначала первый вид спецификации. Набор ограничений состоит из *элементов спецификации*, каждый из которых задаёт допустимые объектные выражения. Элементом спецификации может быть конкретный символ, например: `'к'`, `/21/`, `/beta/`, стандартное множество однотипных символов или термов, или же множество, задаваемое спецификатором. Соответствующие БНФ-правила:

элемент_спецификации ::= *символ-литера* | *символ-число* | *символ-метка* | *стандартное_множество* | *спецификатор*

стандартное_множество ::= `S | F | N | O | L | D | W | B`

Как *стандартные* зафиксированы следующие множества:

`S` – множество всех символов;

`F` – множество символов-меток;

`N` – множество символов-чисел;

`O` – множество символов-литер (объектных знаков);

`L` – множество букв (русских и латинских);

`D` – множество десятичных цифр;

`W` – множество всех термов;

`B` – множество структурных термов, т.е. термов вида **(***R***)**, где *R* – произвольное объектное выражение Рефала.

Из элементов спецификации может быть составлена *цепочка*, которая обозначает объединение соответствующих множеств допустимых значений:

цепочка_элементов_спецификации ::= *пусто* |
элемент_спецификации цепочка_элементов_спецификации

Например, цепочка `L'5'` задаёт в качестве допустимых значений все буквы и цифру 5, а цепочка `LD` – все буквы и цифры.

Если в цепочке элементов спецификации содержится несколько символов-литер подряд, их можно слить в одну цепочку литер. К примеру, цепочка `'+-*/'` описывает множество знаков арифметических операций, а цепочка `D'ABCDEFabcdef'` обозначает множество из цифр и латинских букв A, B, C, D, E, F, a, b, c, d, e, f, т.е. представляет все шестнадцатеричные цифры.

Набор ограничений строится из цепочек элементов спецификации следующим образом:

набор_ограничений ::= *цепочка_элементов_спецификации* |
цепочка_элементов_спецификации
(*цепочка_элементов_спецификации*) *набор_ограничений*

В общем случае набор имеет вид $C_n^+ (C_n^-) C_{n-1}^+ (C_{n-1}^-) \ldots C_1^+ (C_1^-) C_0$, $n \geq 0$, где C_i^+ и C_i^- – цепочки элементов спецификации, и интерпретируется следующим образом. Если n=0, то в набор ограничений входит одна цепочка C_0, и именно она описывает множество допустимых значений. Если же $n \geq 1$, то множество допустимых значений M_n определяется по рекуррентной формуле:

$$M_n = C_n^{+} \cup (M_{n-1} \setminus C_n^{-})$$

где $\cup$ и $\setminus$ – операции объединения и разности множеств, а M_{n-1} – это множество, определяемое набором ограничений $C_{n-1}^{+}(C_{n-1}^{-})$ ... $C_1^{+}(C_1^{-})\ C_0$. Отметим, что цепочка C_0 при $n \geq 1$ может быть опущена – в этом случае по умолчанию считается, что $C_0 \equiv$ `W` (множество всех термов).

Приведём примеры наборов ограничений и соответствующих допустимых значений:

`'ABC'` – любой из символов-литер `'A'`, `'B'`, `'C'`;

`('ABC')` – любое выражение-терм, за исключением символов-литер `'A'`, `'B'`, `'C'`;

`('A')L` – любая буква, за исключением буквы `'A'`;

`('XY')L('0')D` - любая буква, за исключением букв `'X'` и `'Y'`, или любая цифра, за исключением цифры `'0'`.

Набор ограничений, записанный в скобках, представляет собой спецификацию переменной, и трактовка этой спецификации зависит от типа переменной. Спецификация `S`- или `W`-переменной описывает множество допустимых значений этой переменной, в то время как спецификация `V`- или `E`-переменной определяет, к какому множеству должен принадлежать каждый терм верхнего уровня выражения, являющегося значением этой переменной.

Приведём примеры специфицированных переменных, указывая их возможные значения:

`S(('89')D)1` – восьмеричная цифра;

`S(D'abcdefABCDEF')2` – шестнадцатеричная цифра, причём все цифры, большие `9`, могут записываться как строчной, так и прописной латинской буквой;

`S((/0/)N)3` – целое неотрицательное число, отличное от нуля;

`W(B)4` – произвольное объектное выражение в структурных скобках;

`V(('89')D)5` – восьмеричное число без знака;

`V(D'abcdefABCDEF')6` – шестнадцатеричное число без знака;

`V('+-')x` – непустая последовательность из знаков `'+'` и `'-'`;

`E('+-')x` – произвольная, возможно пустая, последовательность из знаков `'+'` и `'-'`;

`E(('+-'))X` – выражение, которое не содержит на верхнем уровне ни одной литеры `'+'` или `'-'`, значением переменной может быть, например, выражение **`('+')('-')`**, но не выражение `'+'`**`('-')`**;

`E(B)x` – последовательность, возможно пустая, из структурных термов, например: **(**`'A+D-S'`**)(**`/56/'+'/45/`**)()(**`'A'`**(**`'BC'`**)'**`DE'`**)**.

Две стоящие подряд переменные `S(L)x E(LD)y` означают идентификатор в обычном его понимании, т.е. последовательность из букв и цифр, начинающаяся с буквы.

Заметим, что при записи спецификации переменной можно вставлять пробелы после открывающей `(` и перед закрывающей `)` скобками спецификации, а также между элементами цепочки спецификации, например: `S( (/0/) N )3` эквивалентно `S((/0/)N)3`.

Рассмотрим теперь второй вид спецификации – указание имени спецификатора, при этом слева и справа от имени записываются знаки двоеточия **:**

спецификатор ::= **:***имя_спецификатора***:**

В этом случае требуется предварительно определить этот спецификатор рефал-предложением вида

имя_спецификатора `S` *набор_ограничений*

Такое предложение вводит (определяет) новый спецификатор с указанным именем и связывает это имя с заданным набором ограничений. В дальнейшем это имя можно употреблять в качестве спецификации или элемента спецификации. К примеру, определив спецификаторы знаков арифметических и логических операций:

```
ArOper  S '+-*/'
LogOper  S '¬V&'
```

можно далее использовать их, записывая в рефал-программе переменные `S:ArOper:x` или `S(:ArOper: :LogOper:)y`. Значением переменной `Sx` может быть только знак арифметической операции, а значением `Sy` – знак арифметической или логической операции.

Заметим, что поскольку элементом спецификации может быть спецификатор, спецификация вида **:***имя_спецификатора***:** равносильна спецификации **(:***имя_спецификатора***:)**, поэтому переменные `S:ArOper:x` и `S(:ArOper:)y` имеют одно и то же множество допустимых значений.

Именем спецификатора может быть произвольный идентификатор. Имена спецификаторов, в отличие от имён рефал-функций, не являются символами-метками. На использование имён спецификаторов наложено следующее ограничение: если имя некоторого спецификатора используется при описании другого спецификатора, то оно должно быть описано к моменту использования. Соответствующий пример показывает

последовательное введение и использование спецификаторов `Binary`, `Octal` и `Decimal` для двоичных, восьмеричных и десятеричных цифр:

```
Binary S '01'
Octal S :Binary: '234567'
Decimal S :Octal: '89'
```

2.3. Особенности синтаксического отождествления

Уточним правила синтаксического отождествления рефал-выражений для случаев специфицированных переменных.

Одна и та же переменная может входить в левую часть предложения (выражение-образец) несколько раз, причём одни вхождения могут быть специфицированы, а другие – нет. В общем случае у каждого вхождения переменной может быть своя спецификация. При отождествлении выражения-образца с объектным выражением такая переменная получает значение, которое должно удовлетворять всем её спецификациям. Таким образом, множество допустимых значений специфицированной переменной представляет собой пересечение множеств допустимых значений, определяемых её спецификациями. При этом спецификации переменной, которые заданы для её вхождений в правую часть рефал-предложения при отождествлении игнорируются (и поэтому не имеют смысла). К примеру, рефал-предложение

```
S(('A'))X S(('B'))X = SX
```

равносильно предложению `S(('AB'))X SX = SX`

и равносильно `S(('AB'))X SX = S('CD')X`

При синтаксическом отождествлении выражения-образца с объектным выражением `S`- и `W`-переменные являются жёсткими элементами. В случае же использования в выражении-образце нескольких `V`- или `E`-переменных возможно возникновение неоднозначности при отождествлении. Для устранения неоднозначности применяются *левое* или *правое согласование*.

При левом согласовании из возможных значений для `V`- и `E`-переменных выбираются самые короткие по числу термов на верхнем уровне (в том числе пустое – для `E`-переменных), в порядке их рассмотрения слева направо, т.е. сначала для первой слева такой переменной выбирается наикратчайшее значение, затем для второй слева и т.д. При правом согласовании самые короткие значения для `V`- и `E`-переменных выражения-образца выбираются в порядке рассмотрения их справа налево.

Например, при левом согласовании выражения-образца `e1';'e2` с объектным выражением `'a:=2; b:=a; c:=a+b'` переменная `e1` получит значение `'a:=2'`, а переменная `e2` – значение `' b:=a; c:=a+b'`. При правом согласовании переменная `e1` получит значение `'a:=2; b:=a'`, а переменная `e2` – значение `' c:=a+b'`.

Для выполнения правого согласования при отождествлении выражения-образца некоторого рефал-предложения следует в его записи использовать знак **R**. За знаком **R** всегда должен следовать хотя бы один пробел. В общем случае знак **R** записывается в предложении после имени функции, но если оно опущено, то перед знаком **R** должен стоять пробел, например:

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

Для левого согласования вместо знака **R** используется знак **L**, например:

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

В результате применения предложения с правым согласованием к выражению `'a:=2; b:=a; c:=a+b'`, это выражение будет заменено на `'a:=2; b:=a'(' c:=a+b')`. В результате же применения предложения с левым согласованием указанное выражение заменится на `'a:=2'(' b:=a; c:=a+b')`.

Если направление отождествления в рефал-предложении не указано, рефал-машина по умолчанию реализует левое согласование, поэтому знак **L** обычно опускается при записи предложений.

Часто направление отождествления не существенно, однако в ряде задач именно правое согласование упрощает программирование.

Рассмотрим функцию `Implic`, расставляющую скобки в логическом выражении, состоящем из переменных (записанных латинскими буквами) и знаков импликации `->`. В полученном выражении порядок вычисления операций импликации задаётся явно и соответствует общепринятому порядку вычисления вложенных операций, например, выражение `'p->q->r->z'` преобразуется в выражение `'((p->q)->r)->z'` .

```
* Функция Implic расставляет скобки в логическом
* выражении, указывая порядок вычисления импликации
Implic  R  e1 '->' s2 = '('<Implic e1>')->' s2
           e1 = e1
```

Применение именно правого отождествления в первом предложении этой функции позволило так просто запрограммировать необходимое преобразование логической формулы.

2.4. Встроенные функции

В языке Рефал-2 реализован достаточно широкий набор встроенных функций, включающий:

- арифметические функции;
- функции-преобразователи типа;
- функции лексического анализа;
- функции ввода и вывода;
- функции для работы с файлами;
- функция порождения процесса;
- функции для работы с копилкой (описаны в следующем разделе).

Приведём описание основных встроенных функций языка Рефал-2.

Арифметические функции

Арифметические функции предназначены для работы с целыми числами. Целое число представляется в языке Рефал-2 как символ-число или как последовательность макроцифр (для чисел, больших по модулю 2^{24}), которым может предшествовать символ-литера `'+'` или `'-'`. Целое число ноль представляется либо символом-числом `/0/`, либо пустым выражением.

Поскольку рефал-функция имеет один аргумент, а арифметические функции имеют обычно два аргумента, принят следующий формат обращения к арифметическим рефал-функциям, позволяющий разделять аргументы: **(***первый_аргумент***)***второй_аргумент*, далее – **(***N1***)***N2*.

При описании каждой арифметической функции будем указывать форму обращения к ней и получаемый результат. В примерах использования функций знак → отделяет функциональное обращение от результата его вычисления.

1) `<Add (`*N1*`)`*N2*`>` – результатом вычисления является рефальское число, равное сумме аргументов функции *N1* и *N2*. Например,
   `<Add ('+'/3/)'-'/5/>` → `'-'/2/`
   `<Add (/34/)/90/>` → `/124/`
2) `<Sub (`*N1*`)`*N2*`>` – вычисляется разность чисел *N1* и *N2*. Например,
   `<Sub (/12/)/5/>` → `/8/`
   `<Sub ('-'/2/)'-'/8/>` → `/6/`
3) `<Mul (`*N1*`)`*N2*`>` – результатом вычисления является произведение чисел *N1* и *N2*. Например,
   `<Mul (/3/)'-'/5/>` → `'-'/15/`

4) <Div (*N1*)*N2*> – вычисляется частное целочисленного деления числа *N1* на *N2*. Например,
   <Div (/12/)/5/> → /2/
5) <Dr (*N1*)*N2*> – возвращает результат в формате *частное*(*остаток*), причём остаток имеет знак первого аргумента *N1* и выполняется равенство: *N1=N2*частное+остаток*. Например,
   <Dr ('-'/14/)/5/> → '-'/2/('-'/4/)
6) <P1 *N*> – функция увеличивает свой аргумент символ-число *N* на единицу и выдаёт в качестве результата; функция работает с числами в диапазоне от 0 до 16777214 (=2^{24}-2). Например:
   <P1 /0/> → /1/
   <P1 /675/> → /676/
7) <M1 *N*> – функция уменьшает свой аргумент символ-число *N* на единицу и выдаёт в качестве результата; функция работает с числами в диапазоне от 1 до 16777215 (=2^{24}-1). Например,
   <M1 /350/> → /349/
8) <Nrel (*N1*)*N2*> – результатом вычисления является выражение вида *знак*(*N1*)*N2*, где *знак* – один из символов '<', '=' или '>', в зависимости от того, является ли целое число *N1* меньшим, равным или большим *N2*. Например:
   <Nrel ('+'/3/)'-'/5/> → '>'('+'/3/)'-'/5/
   <Nrel ('-'/8/)'-'/5/> → '<'('-'/8/)'-'/5/
   <Nrel ('+'/3/)/3/> → '='('+'/3/)/3/

Приведём дополнительные примеры работы арифметических функций с большими числами:

<Add (/12//34/)/7//90/> → /19//124/

<Add ('-'/16777210/)'-'/8/> → '-'/1//2/

<Sub (/1//2/)/8/> → /16777210/

<Mul ('-'/12345/)'-'/12345/> → /9//1404081/

<Div ('-'/9//1404081/)/12345/ → '-'/12345/

<Dr (/1//4/)/25/> → /671088/(/20/)

<Nrel (/16777215/)/1//2/> → '<'(/16777215/)/1//2/

<Nrel ('-'/16777215/)'-'/1//2/> →
'>'('-'/16777215/)'-'/1//2/

Функции-преобразователи типа

9) Функция `Numb` преобразует свой аргумент – цепочку символов-литер, представляющую десятичное целое число со знаком или без него, в соответствующее рефальское число и выдаёт его в качестве результата (функция работает с числами в диапазоне от -2147483647 до 2147483647=2^{31}-1). Например,

```
<Numb '-27'> → '-'/27/
<Numb '+15'> → /15/
<Numb '2147483647'> → /127//16777215/
```

10) Функция `Symb` выполняет обратное преобразование, т.е. преобразует рефальское число в соответствующую цепочку символов-литер (работает с числами в том же диапазоне, что и функция `Numb`). Например,

```
<Symb '-'/348/> → '-348'
<Symb /127//16777215/> → '2147483647'
```

11) Функция `Cvb` – расширение `Numb` для работы с большими числами. Например,

```
<Cvb '123456789012345678901'> →
                /43860//11111659//2034386/
<Cvb '-27'> → '-'/27/
```

12) Функция `Cvd` – расширение `Symb` для работы с большими числами. Например,

```
<Cvd '-'/43860//11111659//2034386/> →
                '-123456789012345678901'
<Cvd /348/> → '348'
```

13) Функция `Ftochar` в качестве результата выдает цепочку символов-литер, из которых состоит её аргумент – символ-метка. Например,

```
<Ftochar /REF/> → 'REF'
```

14) Обратная к ней функция `Chartof` превращает цепочку символов-литер в символ-метку и выдает её в качестве результата, при этом строчные буквы цепочки заменяются соответствующими прописными. Например,

```
<Chartof 'Lr25'> → /LR25/
<Ftochar <Chartof 'AsDf'>> → 'ASDF'
```

Все созданные функцией `Chartof` символы-метки сохраняются в специальной таблице.

15) Функция `Functab` вызывается с аргументом, являющимся символом-меткой. Она нужна в случаях, когда в тексте рефал-программы используется символ-метка, и такая же символ-метка генерируется

функцией `Chartof`. Тогда в начале работы рефал-программы должно вычислиться обращение `<Functab` *символ-метка*`>` с этой символ-меткой.

Функции лексического анализа

16) Функция `Lengw` вычисляет длину своего аргумента-выражения, измеренную как количество термов верхнего уровня, и выдает результат в виде символ-числа, например,

    `<Lengw 'AS'('ADF')'DF'>` → `/5/`.

    Длина пустого выражения равна нулю, т.е. `<Lengw>` → `/0/`.

17) Функция `First` получает в качестве аргумента символ-число, за которым следует произвольное рефал-выражение, и отщепляет от начала этого выражения заданное количество термов на верхнем уровне, заключая их в структурные скобки. Преобразованное таким образом выражение выдается в качестве результата. Если число термов верхнего уровня в этом выражении меньше заданного в аргументе функции числа, то функция возвращает исходное выражение, помещая перед ним символ-литеру `'*'`. Например,

    `<First /3/'a'('b')'cd'('e')>` → `('a'('b')'c')'d'('e')`

    `<First /5/'a'('bcd')'e'>` → `'*a'('bcd')'e'`

18) Функция `Last` осуществляет аналогичное действие, отщепляя заданное количество термов от конца заданного выражения, если это возможно, или же приписывая в конец выражения символ-литеру `'*'`. Например,

    `<Last /3/'a'('b')'cd'('e')>` → `'a'('b')('cd'('e'))`

    `<Last /5/'a'('bcd')'e'>` → `'a'('bcd')'e*'`

Функции ввода и вывода

19) Функция ввода строки с клавиатуры `Card` считывает строку символов до управляющего символа Enter. Обращение к этой функции имеет вид `<Card>`, после его вычисления оно заменяется в поле зрения на введённую строку символов.

20) Функции вывода `Print`, `Printm`, `Prout` и `Proutm` служат для вывода информации на экран компьютера, их единственный аргумент – объектное выражение. При вычислении соответствующего функционального вызова он заменяется в поле зрения для первых двух функций – выражением-аргументом функции, для последних двух – пустым выражением. Отличие функций `Print` и `Prout` от соответствующих `Printm` и `Proutm` заключается в формате вывода выражения: две последние функции выводят выражение в том же виде, как оно записывается в исходных рефал-программах, а `Print` и `Prout`

изменяют при выводе некоторые знаки выводимого выражения: у символов-литер опускаются апострофы, у символов-меток и символов-чисел вместо знаков косой черты печатаются апострофы. Например, в результате вычисления

```
<Printm /VARX/':='/25/';'/Y/':=-'/9/>
```

на экране компьютера появится строка:

```
/VARX/':='/25/';'/Y/':=-'/9/
```

а в результате вычисления

```
<Print /VARX/':='/25/';'/Y/':=-'/9/>
```

на экране компьютера появится строка:

```
'VARX':='25';'Y':=-'9'
```

Функции для работы с файлами

21) Функции `Openget` и `Openput` с аргументом – цепочкой символов, представляющей собой спецификацию файла, открывают указанный файл либо на чтение (функция `Openget`) либо на запись (функция `Openput`). При вычислении функциональный вызов заменяется в поле зрения на пустое выражение. Пример обращения к функции:
```
<Openget 'EXAMPLES\data.txt'>
```
22) Функции `Clsget` и `Clsput` без аргументов закрывают файл, открытый на чтение или запись. При вычислении функциональные вызовы `<Clsget>` и `<Clsput>` заменяются в поле зрения на пустое выражение.
23) Функция `Libget` без аргументов читает очередную строку из файла, предварительно открытого для чтения, при вычислении её вызов заменяется в поле зрения считанной строкой символов либо пустым выражением, если файл прочитан полностью.
24) Функция `Libput` выводит в предварительно открытый для записи файл свой аргумент-выражение, заменяя в этом выражении структурные скобки соответствующими символами-литерами `'('` и `')'`. Аргументом функции может быть объектное выражение, не содержащее составных символов. При вычислении вызов функции `Libput` заменяется в поле зрения на пустое выражение.

Заметим, что в любой момент времени в программе может быть открыто не более одного файла на чтение и не более одного файла на запись.

Функция порождения процесса `Apply` позволяет писать программы, анализирующие аварийные ситуации вида “отождествление невозможно” и “свободная память исчерпана” и предпринимающие некоторые действия по обработке таких ситуаций.

Вызов функции имеет вид: <Apply *имя_функции выражение*>, причём *имя_функции* должно быть задано в виде символа-метки. Такой вызов порождает новое поле зрения, в которое помещается функциональный терм <*имя_функции выражение*>, и делается попытка его вычислить. Результат же возвращается в то поле зрения, из которого была вызвана функция Apply, после чего созданное дополнительное поле зрения уничтожается. По первому символу возвращаемого значения можно определить, удачна ли попытка вычисления функционального терма: символ 'N' означает успешное вычисление, 'R' – произошел останов «отождествление невозможно», 'S' – останов «свободная память исчерпана».

2.5. Функции для работы с копилкой

Как уже отмечалось, особенностью функционального программирования является отсутствие оператора присваивания. Передача данных между функциями осуществляется через их аргументы и возвращаемые результаты. Однако в ряде случаев было бы удобно хранить и передавать информацию с помощью структур данных, доступных всем функциям программы.

В языке Рефал-2 рефал-машина, кроме поля памяти и поля зрения, имеет ещё одно запоминающее устройство, называемое *копилкой*. Копилка является некоторым аналогом глобальных переменных в императивных языках программирования, она даёт возможность именовать и хранить рефал-выражения, предоставляя доступ к ним из любой точки программы.

Копилка содержит последовательность структурных термов вида:

(*имя1* **'='** *выражение1***)(***имя2* **'='** *выражение2***)...(***имяN* **'='** *выражениеN***)**

где *имя1*, *имя2*, ..., *имяN* и *выражение1*, *выражение2*, ..., *выражениеN* – произвольные объектные выражения. Единственное ограничение, накладываемое на выражения *имя1*, *имя2*, ..., *имяN* – они не могут содержать на верхнем уровне символ-литеру '='.

Смысл копилки и том, что в ней под именами *имя1*, *имя2*, ..., *имяN* закопаны соответственно рефал-выражения *выражение1*, *выражение2*, ... , *выражениеN*. Перед началом работы программы копилка пуста.

Работа с копилкой осуществляется с помощью встроенных функций Br (Burry – закопать), Dg (Dig Out – выкопать), Cp (Copy – скопировать), Rp (Replace – заменить) и Dgall (Dig Out All – выкопать всё). Все эти функции, кроме функции Cp, имеют побочный эффект – они меняют содержимое копилки.

Обращение к функции `Br` имеет вид `<Br` *имя*`'='`*выражение*`>`, при его вычислении в поле зрения сам этот вызов заменяется пустым выражением, а содержимое копилки изменяется – в начало содержащейся в копилке последовательности термов добавляется терм **(***имя*`'='`*выражение***)**, т.е. аргумент функции `Br`, заключенный в структурные скобки.

Обращение к функции `Dg` имеет вид `<Dg` *имя*`>`. Эта функция просматривает содержимое копилки от начала в конец в поисках терма вида **(***имя*`'='`*выражение***)** и если находит его, удаляет этот терм из копилки. Результатом обращения к функции является *выражение*, оно заменяет в поле зрения исходное функциональное обращение. Если же указанный терм в копилке не найден, то в качестве результата выдается пустое выражение.

Функция `Cp` имеет тот же формат, что и `Dg`, и работает аналогичным образом, за исключением того, что содержимое копилки не меняется, т.е. найденный терм **(***имя*`'='`*выражение***)** остается в копилке.

Обращение к встроенной функции `Rp` имеет вид `<Rp` *имя*`'='`*новое_выражение*`>`, при его вычислении в поле зрения этот вызов заменяется пустым выражением, а в копилке ищется самый первый терм вида **(***имя*`'='`*выражение***)** и заменяется термом `(`*имя*`'='`*новое_выражение*`)`. Если же в копилке не обнаружено терма искомого вида, то функция `Rp` делает то же самое, что и функция `Br`, т.е. добавляет в начало копилки терм **(***имя*`'='`*новое_выражение***)**.

Функция `Dgall` позволяет вынуть из копилки всё содержимое и поместить его в поле зрения рефал-машины, т.е. вызов `<Dgall>` заменяется в поле зрения последовательностью термов – текущим содержимым копилки, т.е. последовательностью хранящихся в ней термов, начиная с первого. В результате вычисления этого вызова копилка становится пустой.

Можно заметить, что копилка представляет собой также некоторый аналог стека. Так, если под одним и тем же именем было закопано несколько выражений (например, было выполнено: `<Br 'X=A'>`, а затем `<Br 'X=B'>`), то выражение, закопанное последним по времени (в нашем примере – терм **(**`'X=B'`**)**), будет находиться в начале копилки. После этого функцией `Dg` или `Cp` с аргументом `'X'` в копилке будет найден терм **(**`'X=B'`**)**, и значением функции будет выражение `'B'`. Аналогичным образом вычислится и вызов функции `Rp`: в результате него произойдёт замена выражения с заданным именем, закопанного в копилку последним по времени.

2.6. Оформление и запуск программы

В системе программирования Рефал-2 исходная рефал-программа должна быть подготовлена в виде текстового файла с расширением *.ref*. Текст программы представляет собой последовательность *строк*, в которых используются только первые 72 позиции (остальные позиции игнорируются для совместимости со старыми реализациями языка). Если рефал-предложение занимает больше, чем 72 позиции, все его символы, начиная с 73-го, при загрузке в поле памяти обрезаются. Если для записи рефал-предложения требуется большее число позиций, оно записывается на нескольких строках. Для переноса предложения с одной строки на другую служит собственный знак **+**, его можно поставить всюду, где допустимы незначащие пробелы, и затем продолжить запись предложения на следующей строке.

Кроме предложений рефал-функций в тексте программы возможны *директивы* и *строки комментария*. Строка комментария начинается с литеры *****, после которой идут символы комментария. Комментарии игнорируются рефал-интерпретатором и не влияют на выполнение рефал-программы. Допустимы также пустые строки и строки, состоящие из одних пробелов, которые также не влияют на исполнение рефал-программы и могут использоваться для наглядности её записи.

Основное место в рефал-программах занимают описания рефал-функций, состоящие из последовательности предложений. Каждое рефал-предложение записывается на отдельной строке, длинные предложения могут занимать несколько подряд идущих строк. Первое предложение любой функции должно начинаться с идентификатора – имени функции, которое записывается с первой позиции строки. Остальные предложения функции записываются без имени, после начального пробела. Все предложения без имени, вплоть до первого предложения другой функции (или директивы), относятся к этой функции.

При описании в рефал-программе спецификатора его имя также должно быть записано с первой позиции строки.

Как указывалось ранее, имена функций являются символами-метками (но при их записи в функциональных обращениях разрешено опускать знаки косой черты). Язык Рефал-2 требует обязательного описания всех символов-меток, используемых в программе. Это означает, что если некоторые символ-метки используются в программе не в качестве имён функций, а как мнемоничные обозначения в обрабатываемом выражении, они должны быть описаны как *пустые функции*.

Пустой называется функция, в которой нет ни одного предложения (поэтому функциональное обращение к пустой функции приводит к останову «отождествление невозможно»). Описание пустой функции состоит из одной строки, на которой с первой позиции записано имя функции. Например, следующие две строки описывают пустые функции с именами `Psi` и `Alpha`:

```
Psi
Alpha
```

Более компактно пустые функции можно описать с помощью специальной директивы `EMPTY`, которая имеет следующий вид:

`EMPTY` *список_идентификаторов*

где *список_идентификаторов* задаёт имена определяемых пустых функций.

список_идентификаторов ::= *идентификатор* |
идентификатор `,` *список_идентификаторов*

В этой директиве системное слово `EMPTY` должно быть отделено от начала строки по крайней мере одним пробелом. Приведём пример директивы – описание пустых функций `Psi` и `Alpha`:

```
EMPTY  Alpha,Psi
```

Все используемые в программе встроенные функции должны быть объявлены в директиве `EXTRN`, имеющей следующий вид:

`EXTRN` *список_объявлений*

список_объявлений ::= *объявление_функции* |
объявление_функции `,` *список_объявлений*

объявление_функции ::= *внешнее_имя* |
внутреннее_имя `(`*внешнее_имя*`)`

Слово `EXTRN` должно быть отделено от начала строки по крайней мере одним пробелом.

Внешнее имя – это имя встроенной функции языка Рефал-2. При желании можно ввести в программе другие, *внутренние* имена для встроенных функций. Например, встроенные функции `Mul` и `Sub` можно использовать в программе под другими именами, если записать в программе директиву:

```
EXTRN умножение(Mul),разность(Sub)
```

После такого объявления встроенные функции `Mul` и `Sub` в программе должны использоваться под именами `умножение` и `разность` соответственно.

Рефал-программа должна начинаться с директивы START и кончаться директивой END. Общая структура программы такова:

Идентификатор START
рефал-предложения и директивы
END

В первой строке программы, перед словом START обычно ставится *идентификатор* – имя программы, но он может быть и опущен. Перед системными словами START и END должен стоять по крайней мере один пробел.

Запуск рефал-программы осуществляется функцией с именем Go, которая должна быть объявлена в программе директивой ENTRY:

```
ENTRY  Go
```

причём перед словом ENTRY должен стоять по крайней мере один пробел. Функция Go предназначена для инициализации поля зрения рефал-машины, и её описание должно состоять из одного предложения:

Go = *рабочее_выражение*

Это выражение должно содержать вызовы нужных рефал-функций программы, которые осуществляют ввод исходных данных, запускают их обработку и выполняют вывод полученных результатов. Поскольку в начале работы рефал-интерпретатора в поле зрения рефал-машины автоматически помещается функциональный терм <Go>, То на первом шаге работы рефал-машины функциональный терм <Go> заменяется в поле зрения на *рабочее_выражение*, которое необходимо вычислить.

Приведём пример рефал-программы, запрашивающей у пользователя целое положительное число, вычисляющей факториал этого числа и выводящей результат вычисления на экран.

```
Factorial START
* Программа вычисления факториала
* Объявление встроенных функций
    EXTRN умножить(Mul),M1,Prout,Card
    ENTRY Go
* Функция Go инициализирует поле зрения
Go = <prout 'Введите целое положительное число: '>  +
        <fact <card>>
* Объявление встроенных функций
    EXTRN numb,cvd
```

```
* Функция вычисления факториала
fact  v(D)1 = <prout <cvd <fact1 (/1/)<numb v1>>>>
      E1 = <prout 'Число введено неверно'>
* Вспомогательная функция
fact1  (v1)/1/ = v1
       (v1)v2 = <fact1 (<умножить (v1)v2>)<M1 v2>>
    END
```

Для запуска рефал-программы, находящейся, к примеру, в файле с именем `prog.ref`, достаточно поместить указанный файл в директорию `Refal2` и набрать в командной строке `refgo prog.ref`.

Для отладки рефал-программ следует учесть, что интерпретатор языка Рефал-2 во время своей работы в качестве функциональных скобок использует символы **k** и **.** (вместо < используется **k**, а вместо > – точка), а имена всех функций записаны как символы-метки. Именно в таком виде выводится содержимое поля зрения в случае аварийного останова.

3. Язык Рефал-5

Язык Рефал-5 является более поздним, чем Рефал-2, диалектом языка Рефал, включающим в себя практически все средства базисного Рефала и некоторые возможности языка Рефал-2. В тоже время он имеет ряд существенных отличий – в нём появились такие новые средства, как *условия* и *присоединённые блоки*, предложенные взамен средств спецификации переменных. За счёт этих новых средств язык Рефал-5 по своей выразительной мощности не уступает языку Рефал-2.

3.1. Основные особенности

В сравнении с языком Рефал-2 к числу основных особенностей языка Рефал-5 относятся:

- Отсутствие `W`- и `V`-переменных. Как и в базисном Рефале, в Рефале-5 используются только три вида переменных: `s`-переменные, `t`-переменные и `e`-переменные – их значениями могут быть соответственно символы, термы (т.е. символы и выражения в структурных скобках) и произвольные выражения.
- Отсутствие спецификаций переменных и, как следствие, определяемых спецификаторов.
- Использование для разрешения неоднозначностей при синтаксическом отождествлении только левого согласования, т.е. правое согласование не реализовано.
- Возможность записывать в левой части рефал-предложений дополнительное условие применимости этого предложения в виде так называемой *where-конструкции*, или *условия*. Такая *условная конструкция* состоит из рефал-выражения и образца, с которым должно быть успешно отождествлено это рефал-выражение, причём последнее может содержать функциональные термы.
- Использование *присоединённых блоков* или *with-конструкций*, позволяющих задавать вместо правой части рефал-предложения последовательность рефал-предложений, предваряя его рефал-выражением, к которому они должны быть применены. Фактически это даёт возможность вводить и использовать в рамках рефал-предложения новую функцию, без её оформления как самостоятельной функции.

Язык Рефал-5 имеет также ряд отличий в синтаксисе. Дополнительными собственными знаками языка являются:

- ✓ точка **.** используется для отделения признака типа переменной от её индекса; в качестве признака типа могут использоваться только строчные буквы `s`, `t` или `e`;
- ✓ точка с запятой **;** служит разделителем рефал-предложений, а также разделителем функциональных определений;
- ✓ фигурные скобки **{** и **}** служат для записи *блока рефал-предложений*: первая скобка открывает блок, вторая – закрывает;
- ✓ двоеточие **:** используется как знак операции отождествления, для отделения образца и выражения при записи where-конструкций или для отделения блока предложений от рефал-выражения, к которому он применяется, в with-конструкциях;
- ✓ запятая **,** и амперсенд **&** используются при записи where- и with-конструкций для их отделения от предшествующих выражений.

В итоге, набор *собственных* знаков языка Рефал-5 таков:

' ( ) < > = s t e = . ; : { } , &

В этом наборе отсутствует знак косой черты, используемый в Рефале-2 для записи составных символов: символов-меток и символов-чисел. В Рефале-5 соответствующие конструкции, называемые соответственно *идентификаторами* и *макроцифрами*, записываются без всяких спецзнаков, например, идентификаторы `True`, `X`, `Begin`, макроцифры `3`, `853`. Идентификаторы и макроцифры могут входить в качестве атомарных элементов в рефал-выражения, например:

```
X23 '+' 678 '*'(Z45 '-' Fsr '/' 936)
```

Идентификатором в Рефале-5 считается строка из не более чем 15 алфавитно-цифровых знаков и знаков тире и подчеркивания, начинающаяся с заглавной буквы, внутри идентификаторов строчные и прописные буквы различаются. Переменные языка Рефал-5 не включают спецификацию, а в качестве индекса допускаются не только буква или цифра, но и целое положительное число и *идентификатор*. В качестве признака типа используются только строчные (малые) буквы `s`, `t`, `e`.

переменная ::= *признак_типа буква* | *признак_типа цифра* |
признак_типа.*идентификатор* | *признак_типа*.*целое_число*

признак_типа ::= `s` | `t` | `e`

Примеры переменных: `e.True`, `tA`, `t.28`, `s.First`, `e.last`, `s2`. Идентификатор, используемый как индекс после точки, может начинаться как со строчной буквы, так и с заглавной (прописной). Подчеркнём, что

при записи переменной точка может быть опущена только в случаях, когда индексом является буква или цифра: записи `e.X` и `eX` представляют одну и ту же переменную.

Пробел используется как разделитель переменных, символов и чисел. В ряде случаев разделяющий пробел может быть опущен – тогда, когда это не приводит к неоднозначности трактовки конструкции. Например, допустима запись `t1 t2 t3`, как и эквивалентная ей запись `t1t2t3`, поскольку отсутствие точки за указателем типа `t` означает, что индексом может быть только один символ. В тоже время запись `t.1t.2t.3` ошибочна.

В языке Рефал-5 запись определений функций отличается от их записи в языке Рефал-2. Рефал-предложения не обязательно должны записываться построчно. Предложения, определяющие одну функцию, группируются в *блок предложений* – последовательность рефал-предложений, разделяемых точкой с запятой. Определение функции представляет собой заключённый в фигурные скобки блок, перед которым записано имя функции.

определение_функции ::= *имя_функции* { *блок* } |
`$ENTRY` *имя_функции* { *блок* }

блок ::= *предложение* | *предложение* **;** | *предложение* **;** *блок*

К примеру, определение функции `Palindrom` (см. раздел 1.6) на языке Рефал-5 имеет такой вид:

```
* Функция Palindrom проверяет, является ли
* входная строка палиндромом
Palindrom { s.1 e.2 s.1 = <Palindrom  e.2> ;
            s.1 =  'yes' ;
                =  'yes' ;
            e.1 =  'no' ;   }
```

Заметим, что системное слово `$ENTRY` при определении функции используется в случае, когда рефал-программа состоит из нескольких модулей (см. раздел 3.5).

3.2. Условная конструкция

Условная, или where-конструкция, а более коротко – *условие* языка Рефал-5 может записываться за образцом в левой части рефал-предложения, отделяясь от него запятой (допустим также знак амперсенда **&**, но далее везде будем использовать только запятую). Эта конструкция служит для задания дополнительных условий на применимость

предложения. В общем случае в одном предложении допускается несколько условий, разделяемых между собой запятой или амперсендом, фактически означающими конъюнкцию этих условий.

Напомним, что образцом называется рефальское выражение, не содержащее функциональных термов. В Рефале-5 предложение функции может иметь вид:

образец последовательность_условий = выражение

последовательность_условий ::=

, *условие последовательность_условий* | *пусто*

условие ::= *аргумент* **:** *образец*

аргумент ::= *выражение*

Каждое условие имеет вид *выражение-аргумент* **:** *выражение-образец*, или более кратко – **E:P**, знак двоеточия означает выполнение синтаксического отождествления заданного образца с аргументом. Условие считается выполненным, если отождествление успешно. При наличии условий в рефал-предложении сначала происходит синтаксическое отождествление первого, основного образца этого предложения с обрабатываемым выражением в поле зрения, и, если оно успешно, последовательно выполняется проверка записанных в предложении условий, в том порядке, как они указаны в предложении. При этом происходит отождествление выражения-аргумента каждого условия с соответствующим выражением-образцом. Если все проверки успешны (условия выполнены) данное рефал-предложение считается применимым.

Выражение-аргумент каждого условия может содержать как функциональные термы (т.е. вызовы функций), так и переменные. Единственное ограничение, налагаемое на выражение-аргумент, состоит в том, что он может включать только те переменные, чьи значения определены к моменту проверки этого условия – такие переменные называются *связанными*. Поскольку рефал-переменные локализованы в предложениях, требование связности переменных означает, что в выражении-аргументе могут использоваться только те переменные, которые использованы в основном образце предложения и в выражениях-образцах предыдущих условий. В свою очередь, образец каждого условия может включать как переменные без значений, так и связанные переменные, значения которых уже определены в ходе предыдущих вычислений.

В общем случае в ходе проверки очередного условия получают значения переменные его образца, и при проверке следующих условий они являются уже связанными. После проверки всех условий все переменные

предложения становятся связанными, их значения могут быть использованы в правой части предложения.

Опишем подробнее процесс проверки условий. При вычислении каждого условия $\mathbf{E_i : P_i}$ рефал-машина порождает новое временное поле зрения, в которое помещает выражение-аргумент $\mathbf{E_i}$, заменив в нём все связанные переменные их значениями, и работает (в обычном режиме) над этим полем до тех пор, пока в нём есть необработанные функциональные вызовы. Затем рефал-машина переходит к синтаксическому отождествлению результирующего выражения с образцом $\mathbf{P_i}$, предварительно заменив в $\mathbf{P_i}$ все связанные переменные на их значения. Если отождествление было успешным, то аналогичным образом вычисляется следующее условие. Если же условий больше нет, то рефал-машина применяет рассматриваемое предложение, т.е. производит действия, определяемые его правой частью или присоединённым блоком.

В случае неуспешного отождествления $\mathbf{E_i}$ и $\mathbf{P_i}$ из очередного условия, рефал-машина возвращается к предыдущему условию $\mathbf{E_{i-1} : P_{i-1}}$ и пытается найти другой вариант отождествления этой пары, придавая более длинные возможные значения e-переменным в $\mathbf{P_{i-1}}$ (при i=1 возврат происходит к обрабатываемому в основном поле зрения выражению и основному образцу предложения). Если других вариантов отождествления нет, то текущее рефал-предложение считается неприменимым. Если же найден другой вариант отождествления $\mathbf{E_{i-1}}$ и $\mathbf{P_{i-1}}$, рефал-машина снова пытается отождествить пару $\mathbf{E_i}$ и $\mathbf{P_i}$, но уже при новых найденных значениях входящих в них связанных переменных. Каждый раз после того, как проверка условия $\mathbf{E_i : P_i}$ завершается либо успехом, либо неудачей, временное поле зрения, созданное для $\mathbf{E_i}$, уничтожается.

В качестве примера использования условной конструкции приведём функцию, которая ищет первое вхождение знака `'+'` или `'-'` на верхнем уровне обрабатываемого выражения (оно может содержать структурные скобки) и разбивает это выражение на части, заключая соответственно подвыражение, предшествующее найденному знаку, и подвыражение, следующее за ним, в структурные скобки. Например, в результате применения этой функции к выражению

```
'X*Y/8*'('Z/3-56')'+9*A-23*B'
```

получится выражение `('X*Y/8*'('Z/3-56'))'+'('9*A-23*B')`

```
* Функция Search-plus-minus заключает в структурные
* скобки выражение, предшествующее и
* следующее за первым знаком + или -
```

```
Search-plus-minus {
    e.1 s.2 e.3 , '+-' : e.X s.2 e.Y =
                (e.1) s.2 (e.3);
    e.X  =  e.X  }
```

Первое предложение этой функции после основного образца `e.1 s.2 e.3`, выделяющего в обрабатываемом выражении символ `s.2`, содержит дополнительное условие его равенства одному из знаков `'+'` и `'-'` – для этого используется образец условия `e.X s.2 e.Y` и выражение-аргумент `'+-'`. Второе предложение функции добавлено для того, чтобы она была всюду определена.

Основной образец первого предложения этой функции применим к любому выражению, содержащему хотя бы один символ, и при отождествлении с непустым выражением переменная `s.2` получит в качестве значения первый символ этого выражения. Дополнительное условие первого предложения требует отождествимости значения переменной `s.2` с одним из знаков `'+'` или `'-'` (при успешном отождествлении либо переменная `e.X`, либо переменная `e.Y` будет иметь в качестве значения пустое выражение). При успешной проверке условия, когда значением `s.2` является один из знаков `'+'` или `'-'`, выполняется правая часть предложения, в которой выражения `e.1` и `e.3` заключаются в скобки.

При неуспешной проверке дополнительного условия (когда значение переменной `s.2` – любой знак, отличный от плюса и минуса) происходит возврат к отождествлению основного образца предложения, значение переменной `e.1` удлиняется, и находится новый вариант отождествления, при котором `s.2` получает в качестве значения второй (от начала) символ исходного выражения. После чего снова происходит проверка дополнительного условия, и если она неуспешна, вновь происходит возврат к отождествлению основного образца и находится новое значение переменной `s.2`, а значение переменной `e.1` снова удлиняется – и так до тех пор, пока значением `s.2` не станет знак `'+'` или `'-'` и будет выполнена правая часть предложения.

Если же в обрабатываемом выражении вообще нет символов `'+'` и `'-'`, то после безуспешного перебора всех символов его верхнего уровня (в качестве значения `s.2`) первое предложение функции будет признано неприменимым, и выполнится второе предложение.

3.3. Присоединённый блок

В языке Рефал-5 предложение функции может также иметь вид:

образец последовательность_условий **,** *присоединённый_блок*

присоединённый_блок ::= *аргумент* **:** { *блок* }

Присоединённый блок вида *выражение-аргумент* **:** *блок* служит для определения новой функции без имени и её вызова внутри рефал-предложения.

Блок (последовательность рефал-предложений) присоединяется к левой части рефал-предложения, он записывается после его основного образца и следующими за ним дополнительными условиями, в совокупности определяющими применимость этого предложения. Считается, что правая часть предложения в этом случае отсутствует, отсутствует и знак =, разделяющий левую и правую часть предложения, а перед присоединённым блоком ставится запятая, отделяющая его от левой части рефал-предложения.

Выражение-аргумент присоединённого блока, как и выражение-аргумент условной конструкции, может содержать функциональные термы и переменные, но только связанные. Двоеточие между выражением-аргументом и блоком предложений означает в данном случае применение блока к заданному выражению-аргументу, т.е. применение функции, определённой предложениями блока, к заданному аргументу.

Укажем правила обработки присоединённого блока рефал-машиной. Вычисление выражения-аргумента присоединенного блока осуществляется так же, как вычисление выражения-аргумента при обработке условной конструкции. Само выполнение присоединённого блока предложений реализуется так же, как и выполнение вызова функции по имени: вычисленное значение аргумента поочередно сопоставляется с левыми частями предложений блока – с тем, чтобы найти применимое предложение. Применение найденного предложения определяет не только результат вычисления блока, но и результат выполнения рефал-предложения, к которому этот блок был присоединён. В случае, когда ни одно из предложений блока неприменимо, возникает аварийный останов рефал-машины (аналогично обычному функциональному вызову).

В присоединённом блоке могут использоваться переменные из основного образца рефал-предложения и дополнительных условий, к моменту выполнения блока эти переменные уже имеют значения (являются связанными).

Подчеркнём, что начало выполнения присоединённого блока означает полное завершение процесса отождествления, осуществляемого

при проверке основного образца предложения и дополнительных условий. Это означает, что никакие другие варианты отождествления основного образца и образцов этих условий в ходе дальнейших вычислений рассматриваться не будут.

В качестве примера присоединённого блока рассмотрим функцию `Tran`, она ищет цепочку из трёх символов, средний символ – знак равенства, среди фрагментов, на которые входная цепочка символов делится знаками `*`. Если такая цепочка обнаруживается, `Tran` заключает её в скобки и выдаёт в качестве результата, иначе выдаёт `'No'`. Например, результатом применения функции `Tran` к строке `'X-2+Y*P/3=9*A=5*Z-8'` будет выражение **(**`'A=5'`**)**. В своей работе `Tran` использует `Fract` – вспомогательную функцию, которая возвращает `T`, если её аргумент является искомой цепочкой, и `F` в ином случае.

```
* Вспомогательная функция проверки цепочки символов
Fract { s.1 '=' s.2 = T;
        e.X = F }

* Функция Tran возвращает найденную цепочку
* из трёх символов, заключённую в структурные скобки,
* либо строку 'No'
Tran { e.1 '*' e.2 , <Fract e.1>: { T = (e.1)
                                   F = <Tran e.2> };
       e.X = 'No'  }
```

Как можно заметить, присоединённый блок позволяет определить безымянную функцию именно в том месте, где нужно её использовать.

В общем случае присоединённый блок может содержать внутри себя другие присоединённые блоки, и уровень вложенности не ограничивается. Функция `OrdABC` представляет пример функции, в которой используются вложенные присоединённые блоки. Эта функция выдает `T`, если в обрабатываемом выражении на верхнем уровне встречаются символы `'A'`, `'B'` и `'C'`, причём символ `'A'` стоит левее символа `'B'`, а символ `'B'` – левее символа `'C'`. Во всех остальных случаях результатом является `F`.

```
* Функция OrdABC возвращает T,
* если в исходном выражении
* на верхнем уровне есть символ 'A',
* правее него встречается символ 'B',
* а правее этих двух символов встречается 'C'.
* Иначе функция возвращает F.
```

```
OrdABC {
    e.X  'A'  e.1 ,  e.1 :
             { e.Y  'B'  e.2 ,  e.2 :
                      { e.Z  'C'  e.3  =  T ;
                        e.Z  =  F  } ;
               e.Y  =  F  } ;
    e.X  =  F  }
```

Для решения рассмотренной задачи можно написать более простую и понятную функцию:

```
SimpleOrdABC {
    e.X  'A'  e.Y  'B'  e.Z  'C'  e.D  =  T ;
    e.X  =  F   }
```

Эта функция является менее эффективной, т.к. образец первого предложения содержит большое количество e-переменных, и в ходе его отождествления будут пробоваться их различные значения. В функции `OrdABC` отождествление выражений будет происходить быстрее: попытка найти символ `'B'` будет происходить только после того, как в выражении-аргументе найден символ `'A'`. Причём `'B'` ищется только в подвыражении, расположенном правее `'A'`, а подвыражение, стоящее левее, более не рассматривается. Аналогичным образом будет происходить поиск символа `'C'`. В итоге результат (`T` или `F`) будет получен быстрее.

3.4. Встроенные функции

Приведём перечень основных встроенных функций языка Рефал-5, характеризуя подробно лишь те функции, аналоги которых отсутствуют в языке Рефал-2. Если формат задания аргументов встроенной функции не указывается, то он точно такой же, как и в языке Рефал-2. Более подробное описание встроенных функций Рефала-5 можно найти в [9, 10].

Арифметические функции

Аргументами арифметических функций и вычисляемыми ими значениями являются числа, представленные как знак числа и макроцифра, причём знак может отсутствовать. Так же, как и в Рефале-2, формат задания аргументов для двухместных арифметических функций (*N1*)*N2*. Перечислим основные арифметические функции:

1) `Add` (может обозначаться как +) вычисляет сумму своих аргументов.
2) `Sub` (может обозначаться как −) вычисляет разность своих аргументов.
3) `Mul` (может обозначаться как *) вычисляет произведение своих аргументов.

4) `Div` (может обозначаться как `/`) вычисляет частное целочисленного деления первого аргумента на второй.
5) `Mod` возвращает остаток от целочисленного деления первого аргумента на второй.
6) `Divmod` (аналог `Dr` в языке Рефал-2) возвращает результат в формате (*частное*) *остаток*, причём остаток имеет знак первого аргумента и выполняется равенство: *N1=N2*частное+остаток*.
7) `Compare` (аналог `Nrel` в Рефале-2) – выполняет сравнение двух заданных чисел: возвращается либо знак `'-'`, если первый аргумент-число *N1* меньше второго аргумента *N2*, либо знак `'+'`, если первый аргумент больше второго, либо `'0'`, когда аргументы равны.

Функции символьной обработки охватывают несколько групп:

- ***Функции-преобразователи типа***, к которым относятся:

8) `Numb` – перевод заданной цепочки десятичных цифр в число (макроцифру), представленное этими символами.
9) `Symb` – обратное преобразование, т.е. перевод заданного числа (макроцифры) в цепочку десятичных цифр.
10) `Explode` – перевод заданного идентификатора в цепочку символов-литер.
11) `Implode` – создание идентификатора из всех начальных алфавитно-цифровых символов заданного выражения, возвращается идентификатор, за которым следует остаток строки, или макроцифра `0` и исходная строка, если строка начинается не с буквы. Например:

```
<Implode 'Asd12'> → Asd12
<Implode 'Asd12y/89'> → Asd12y '/89'
<Implode '12(Aas)kl'> → 0 '12(Aas)kl'
```

- ***Функции выделения*** в заданном выражении заданного количества термов на верхнем уровне:

12) `First` выделяет нужное количество термов верхнего уровня от начала заданного выражения и заключает их в структурные скобки. Если в аргументе функции задано число, превышающее количество термов выражения (на верхнем уровне), то всё выражение заключается в скобки. Например:

```
<First 4 'A+'('B-C')'*D'> → ('A+'('B-C')'*')'D'
<First 7 'A+'('B-C')'*D'> → ('A+'('B-C')'*D')
```

13) `Last` – выделяет нужное количество термов с конца заданного выражения и заключает оставшиеся в начале выражения термы в

структурные скобки. Например:

```
<Last 3 'A+'('B-C')'*D'> → ('A+')('B-C')'*D'
<Last 7 'A+'('B-C')'*D'> → ()'A+'('B-C')'*D'
```

- ***Функции изменения регистра*** букв, входящих в заданное выражение:

14) `Lower` замещает прописные буквы на строчные.
15) `Upper` замещает строчные буквы на прописные.

- ***Функция вычисления длины заданного выражения***, т.е. количества входящих в него термов верхнего уровня:

16) `Lenw` (аналог функции `Lengw` Рефала-2), её результат – макроцифра – длина выражения в термах, за которой следует само выражение. Например,

```
<Lenw 'A+'('B-C')'*D'> → 5 'A+'('B-C')'*D'
```

- ***Функция определения типа*** первого символа заданного выражения:

17) `Type` возвращает два символа-индикатора типа первого терма выражения-аргумента, за которыми следует само исходное выражение. Возвращаемые индикаторы соответствуют:

`'Ll'` – символ-буква;

`'D0'` – символ-цифра;

`'Wi'` – идентификатор;

`'N0'` – макроцифра;

`'B0'` – выражение начинается с левой структурной скобки;

`'Pl'` – любой другой объектный символ;

`'*0'` – если исходное выражение является пустым.

Например:

```
<Type '5bn9'(36)> → 'D05bn9'(36)
<Type (Num)'5bn9'(36)> → 'B0'(Num)'5bn9'(36)
<Type '/5bn9'(36)> → 'Pl/5bn9'(36)
<Type> → '*0'
```

Функции ввода, вывода и работы с файлами

18) `Card` выполняет чтение цепочки символов из входного файла, её значение – считанная цепочка литер; по окончании файла функция возвращает число `0`.

19) `Print` и `Prout` осуществляют печать заданного выражения, значением функции является соответственно заданное выражение или пустое выражение.

20) `Open` – открытие файла для записи или чтения. Обращение к функции имеет вид: `<Open Mode Descr File-name>`, где
`Mode` – либо `'r'` или `'R'` (открыть для чтения), либо `'w'` или `'W'` (открыть для записи);
`Descr` – файловый дескриптор (число в диапазоне 1-19, 0 соответствует терминалу);
`File-name` – строка, задающая имя файла.

21) `Get` вводит строку текста из файла, заданного дескриптором, значением функции является считанная цепочка символов.

22) `Put` выводит заданное выражение в заданный дескриптором файл, результат функции – выводимое выражение.

23) `PutOut` аналогична `Put`, но возвращает пустое значение.

Заметим, что использование файловых дескрипторов позволяет, в отличие от Рефала-2, работать одновременно с несколькими файлами чтения или записи.

Функции работы с копилкой

24) `Br` – закапывание выражения в копилку.

25) `Dg` – выкапывание выражения из копилки.

26) `Cp` – копирование выражения из копилки.

27) `Rp` – замена выражения, закопанного под заданным именем.

28) `Dgall` – выкапывание всей копилки.

Системные функции

29) Функция без аргумента `Time` возвращает строку, обозначающую текущий день недели, месяц, дату, время и год.

30) Функция `Mu` задаётся в одном из двух форматов: `<Mu Fname Expr>` или `<Mu (String) Expr>`. Она осуществляет вызов функции с именем `Fname` (для первого формата), либо функции с именем, являющимся результатом вычисления `<Implode String>` (для второго формата) и аргументом – выражением `Expr`. Результатом функции является результат вычисления `<Fname Expr>` или `<<Implode String> Expr>`. Если в точке вызова функции `Mu` нужная функция не является видимой (т.е. либо определенной в текущем модуле, либо объявленной как внешняя), то возникает ошибка.

3.5. Оформление программы

Если программа на языке Рефал-5 состоит из нескольких модулей, доступные для использования в других модулях функции помечаются в начале своего описания системным словом `$ENTRY`. Если в модуле используются функции, описанные в других модулях, их имена должны быть объявлены как внешние в соответствующей *external-директиве*. Приведём соответствующие БНФ-правила:

программа ::= *определение_функции* | *определение_функции программа* |
определение_функции **;** *программа* | *external-директива* **;** *программа* |
программа external-директива **;**

external-директива ::= `$EXTERNAL` *список_имён_функций* |
`$EXTERN` *список_имён_функций* |
`$EXTRN` *список_имён_функций*

список_имён_функций ::= *имя_функции* |
имя_функции **,** *список_имён_функций*

Отметим, что системные слова `$ENTRY`, `$EXTERNAL` и его сокращения всегда записываются прописными (заглавными) буквами.

Обратим также внимание, что определение функции обычно отделяется от последующей программы знаком **;**, но этот знак может и отсутствовать. Неоднозначности не возникает, поскольку определение функции всегда состоит из двух элементов – имени функции и блока рефал-предложений в фигурных скобках. В то же время в конце *external-директивы* должен стоять знак **;**.

В программах на Рефале-5 могут использоваться два вида комментариев – строковые комментарии (как в Рефале-2) и *комментарии-вставки*. Строкой комментария считается любая строка программы, начинающаяся со знака `*`. Комментарием-вставкой является любая строка, начинающаяся комбинацией символов `/*` и заканчивающаяся `*/`, она возможна в любом месте программы, где может быть вставлен пробел.

Как и в Рефале-2, для инициализации поля зрения рефал-машины используется функция с именем `Go`; перед её именем необходимо поставить системное слово `$ENTRY`.

В отличие от Рефала-2, встроенные функции объявлять в *external-директиве* не нужно.

Ниже приведён пример программы вычисления факториала, состоящей из двух модулей: основной модуль находится в файле `prog.ref`, функция вычисления факториала описана в файле `fact.ref`.

```
* Основной модуль программы вычисления факториала
$EXTRN Fact ;
$ENTRY Go {
      = <Prout 'Введите целое положительное число: '>
             <Check <Card>>
} ;
* Функция проверяет, что введённая цепочка символов -
* число, и выводит факториал этого числа. Если
* введено не число, выводится строка
* 'Число введено неверно '.
Check { s.1 e.2 ,
         <Subset s.1 e.2 ('1234567890')> : True
            = <Prout <Fact <Numb s.1 e.2>>>  ;
        e.1 = <Prout 'Число введено неверно '>
} ;
* Вспомогательная функция проверки, что введённая
* цепочка символов состоит из цифр.
Subset { (e.1) = True ;
         s.1 e.2 (e.3 s.1 e.4) =
               <Subset e.2 (e.3 s.1 e.4)> ;
         e.1 = False
}

* Модуль вычисления факториала
$ENTRY Fact { e.1 = <Fact1 (1) e.1>  };
Fact1 { (e.1) 1 = e.1 ;
       (e.1) e.2 = <Fact1 (<* (e.1) e.2>)<- (e.2) 1>>
 }
```

Для запуска приведённой программы необходимо сначала откомпилировать каждый из файлов, выполнив две команды:

```
refc prog
refc fact
```

В результате будут сформированы два новых файла: `prog.rsl` и `fact.rsl`, и запуск программы осуществляется командой:

```
refgo prog+fact
```

4. Примеры решения задач на языке Рефал

В данной главе на примере решения нескольких задач рассматриваются полезные приёмы программирования на языке Рефал. Если программы решения рассматриваемых задач для языков Рефал-2 и Рефал-5 отличаются только синтаксисом, приводится только программа на языке Рефал-2.

4.1. Посимвольная обработка текста

В этом подразделе рассматриваются задачи обработки текста, рассматриваемого как простая последовательность символов (как строка).

Задача 1.

Определить функцию `Erase`, удаляющую все лишние пробелы в исходной строке символов, т.е. замещающую каждую группу подряд стоящих пробелов единственным пробелом. Далее для наглядности пробел будем изображать символом `'␣'`. Например, результатом применения функции `Erase` к строке

`'He␣writes␣␣␣as␣␣␣well␣as␣she'`

будет строка `'He␣writes␣as␣well␣as␣she'`

Самым очевидным решением будет следующее: в любой обнаруженной внутри строки паре соседних пробелов следует удалить один, и повторять это до тех пор, пока имеются соседние пробелы.

```
* Функция Erase удаляет в строке лишние пробелы,
* вариант 1
Erase    e1 '␣␣' e2  =  <Erase  e1 '␣' e2>
         e1  =  e1
```

Полученная программа содержит два предложения: рекурсивное (удаляющее пробелы) и завершающее (применяющееся тогда, когда соседствующих пробелов уже нет). Каждое применение первого предложения удаляет один лишний пробел.

Проанализируем работу этой функции. Так как по умолчанию при отождествлении используется левое согласование, то подстрока, являющаяся значением переменной `e1`, не содержит подряд стоящих пробелов. Но `e1` входит в аргумент рекурсивного обращения, поэтому на следующем шаге работы рефал-машины эта подстрока снова будет просматриваться при отождествлении с образцом первого предложения и поиске пары соседних пробелов. Избежать повторного просмотра можно, вынося `e1` за пределы функциональных скобок:

```
* Функция Erase удаляет в строке лишние пробелы,
* вариант 2
Erase    e1 '␣␣' e2  =  e1 <Erase  '␣' e2>
         e1  =  e1
```

В этом более эффективном решении левую угловую скобку вызова функции `Erase` можно рассматривать как *разделитель* обработанной части строки от части, ещё не подвергавшейся обработке.

Для второго варианта решения задачи приведём изменение поля зрения рефал-машины в процессе обработки строки
`'He␣writes␣␣␣as␣␣␣well␣as␣she'`:

№ шага	Содержимое поля зрения рефал-машины перед началом шага
1	`<Erase 'He␣writes␣␣␣as␣␣␣well␣as␣she'>`
2	`'He␣writes'<Erase '␣␣as␣␣␣well␣as␣she'>`
3	`'He␣writes'<Erase '␣as␣␣␣well␣as␣she'>`
4	`'He␣writes␣as'<Erase '␣␣well␣as␣she'>`
5	`'He␣writes␣as'<Erase '␣well␣as␣she'>`
6	`'He␣writes␣as␣well␣as␣she'`

Задача 2.

Определить функцию `Correct`, исключающую из заданной строки любой символ, за которым следует знак `#`. Если в исходной строке стоят подряд несколько знаков `#`, функция должна исключать соответствующее число предшествующих им символов. Например, результатом применения этой функции к строке

```
'Children### is playe#ing'
```

будет строка

```
'Child is playing'.
```

Не задумываясь об эффективности, можно дать такое определение функции:

```
* Функция Correct исключает всякий символ
* перед знаком #, первый вариант решения
Correct  e1 sA '#' e2  =  <Correct  e1  e2>
         e1  =  e1
```

Первое предложение функции находит в строке `'#'` и исключает его и предшествующий ему символ, продолжая рекурсивную обработку полученной строки. Второе предложение завершает рекурсию.

Согласно этому определению обрабатываемая строка будет просматриваться полностью (в ходе её отождествления с образцом первого предложения функции) столько раз, сколько в ней встречается символов #. Но для исключения ненужных повторных просмотров нельзя уже просто вынести подстроку e1 за пределы функциональных скобок, т.к. её самые правые символы могут ещё удаляться. Возникает необходимость в *отделении* внутри функциональных скобок просмотренной части строки от непросмотренной. В языке Рефал для подобных целей обычно используются структурные скобки.

В нашей задаче будем заключать в структурные скобки начальную, уже просмотренную часть строки, т.е. e1. Для введения в обрабатываемую строку структурных скобок определим функцию Correct, обращающуюся к функции удаления Cor и ставящую в начале выражения структурные скобки:

```
* Функция Correct исключает всякий символ
* перед знаком #, второй вариант решения
Correct  e1  =  <Cor  ( ) e1>
Cor  (e1  sX) '#' e2  =  <Cor  (e1) e2>
     (e1) e2 sX '#' e3  =  <Cor  (e1 e2) e3>
     (e1) e2  =  e1  e2
```

Второе предложение функции Cor исключает символ перед первым слева (т.к. по умолчанию согласование левое) одиночным знаком # или перед первым из нескольких подряд стоящих знаков #, при этом правая структурная скобка сдвигается до позиции исключённого символа.

Первое предложение Cor предназначено для случаев, когда в обрабатываемой строке было несколько подряд стоящих знаков #, и поэтому после применения второго предложения внутри структурных скобок остались ещё символы, подлежащие удалению.

Третье предложение завершает обработку строки, убирая вспомогательные структурные скобки. Заметим, что в данном решении порядок предложений изменить нельзя.

Поскольку структурные скобки являются жёсткими элементами, которые проектируются однозначно на соответствующие структурные скобки обрабатываемой строки, их использование ускоряет процесс отождествления при поиске применимого правила функции Cor, и поэтому рассмотренное решение более эффективно.

Далее приводится изменение поля зрения рефал-машины в процессе обработки полученной функцией строки
'Children### is playe#ing'.

№ шага	Содержимое поля зрения рефал-машины перед началом шага
1	`<Correct 'Children### is playe#ing'>`
2	`<Cor ()'Children### is playe#ing'>`
3	`<Cor ('Childre')'## is playe#ing'>`
4	`<Cor ('Childr')'# is playe#ing'>`
5	`<Cor ('Child')' is playe#ing'>`
6	`<Cor ('Child is play')'ing'>`
7	`'Child is playing'`

Задача 3.

Определить функцию `Count`, подсчитывающую число вхождений в заданную строку символов `'A'`. Например, результатом вычисления `<Count 'CATS AND DOGS ARE NOT FRIENDS'>` будет число 3.

Приводимое ниже решение реализует рекурсивный алгоритм подсчёта символов: если найти первое вхождение в текст символа `'A'`, то общее число вхождений символа `'A'` на 1 больше количества его вхождений в оставшуюся часть текста (для подсчёта этого количества используется рекурсивный вызов функции). Если же в тексте нет символов `'A'`, то результатом функции является число 0.

```
* Функция Count подсчитывает количество
* символов A в строке
Count    e1  'A'  e2  =  <Add (/1/)<Count  e2>>
         e1  =  /0/
```

В процессе рекурсивной обработки исходного текста в поле зрения рефал-машины будут накапливаться вложенные рекурсивные вызовы функции `Add` – до тех пор, пока аргумент функции `Count` содержит символы `'A'`. Когда же символов `'A'` уже нет, функция `Count` выдаст результат `/0/`, и начнётся последовательное вычисление накопленных рекурсивных вызовов. Например, для текста `'CATS AND DOGS ARE NOT FRIENDS'` изменение поля зрения рефал-машины будет происходить следующим образом:

№ шага	Содержимое поля зрения рефал-машины перед началом шага
1	`<Count 'CATS AND DOGS ARE NOT FRIENDS'>`
2	`<Add (/1/)<Count 'TS AND DOGS ARE NOT FRIENDS'>>`
3	`<Add (/1/)<Add (/1/)` `<Count 'ND DOGS ARE NOT FRIENDS'>>>`

4	<Add(/1/)<Add(/1/)<Add(/1/) <Count 'RE NOT FRIENDS'>>>>
5	<Add (/1/)<Add (/1/)<Add (/1/) /0/>>>
6	<Add (/1/)<Add (/1/) /1/>>
7	<Add (/1/) /2/>
8	/3/

Как видно из этого примера, максимальная глубина вложенности функциональных вызовов (функциональных термов) на единицу больше количества символов `'A'` в исходном тексте.

Напомним, что на каждом шаге работы рефал-машины ищется ведущий функциональный терм (самый левый из самых вложенных функциональных вызовов), и происходит его замена на правую часть применяемого предложения. Ясно, что с ростом глубины вложенности функциональных вызовов все больше и больше времени рефал-машина затрачивает на поиск очередного ведущего функционального терма. Каким образом можно избежать этого?

Опишем соответствующий приём программирования, заключающийся в использовании *накопителя* – выражения, в котором накапливается нужный результат. Для хранения накапливаемого результата в Рефале обычно используют структурные скобки. Заметим, что в предыдущей задаче в структурных скобках накапливалась уже просмотренная часть обрабатываемого текста.

Для получения решения рассматриваемой задачи заведём функцию `CountR`: в её аргумент будет входить ещё не просмотренная часть текста, а перед ней в структурных скобках будет храниться подсчитанное к текущему моменту количество встреченных символов `'A'`. По окончании просмотра текста в этом накопителе окажется нужное нам значение. До начала процесса просмотра текста необходимо установить в накопителе начальное значение – число ноль, и это реализует главная функция `CountB` при обращении к вспомогательной функции `CountR`, которая и выполняет нужную работу:

```
* Главная функция CountB: инициализация накопителя
* и вызов функции подсчёта
CountB   e1  =  <CountR  (/0/)e1>
* Вспомогательная функция подсчёта символов A
CountR   (s1)e2 'A' e3 = <CountR (<Add (/1/)s1>)e3>
         (s1)e2 = s1
```

Для этого решения с накопителем характерно то, что в поле зрения рефал-машины не происходит накопления функциональных вызовов – это

видно при обработке той же самой строки текста 'CATS AND DOG ARE NOT FRIENDS' в поле зрения рефал-машины:

№ шага	Содержимое поля зрения рефал-машины перед началом шага
1	<CountB 'CATS AND DOGS ARE NOT FRIENDS'>
2	<CountR (/0/)'CATS AND DOGS ARE NOT FRIENDS'>
3	<CountR (<Add (/1/)/0/>) 'TS AND DOGS ARE NOT FRIENDS'>
4	<CountR (/1/)'TS AND DOGS ARE NOT FRIENDS'>
5	<CountR (<Add (/1/)/1/>) 'ND DOGS ARE NOT FRIENDS'>
6	<CountR (/2/)'ND DOGS ARE NOT FRIENDS'>
7	<CountR (<Add (/1/)/2/>)'RE NOT FRIENDS'>
8	<CountR (/3/)'RE NOT FRIENDS'>
9	/3/

Как видно, глубина вложенности и количество функциональных термов не превосходит двух и не зависит от количества символов 'A' в исходной строке.

Подчеркнём, что использование накопителя – один из основных приёмов эффективного программирования на Рефале.

4.2. Структурирование текста

Ясно, что использование структурных скобок (являющихся собственными знаками языка Рефал) позволяет упростить и сделать эффективной обработку текстов, в том числе текстов со сложной структурой. Однако в исходных текстах такие скобки изначально отсутствуют. В то же время обрабатываемый текст, например, алгебраические и логические выражения, часто содержит символы круглых скобок, задающих его структуру, учесть которую необходимо при его обработке. Таким образом, возникает важная вспомогательная задача замены символов круглых скобок на структурные скобки Рефала.

Задача 4.

В тексте, рассматриваемом как последовательность символов, заменить литеры скобок '(' и ')' на соответствующие структурные скобки, например, текст '5/((A+B)^2-(A-B)^3)+8*A' преобразовать в '5/'**(**'A+B'**)**'^2-'**(**'A-B'**)**'^3'**)**'+8*A'. Если в исходном тексте баланс символьных скобок нарушен, то выдать сообщение об ошибке.

Для решения задачи необходимо последовательно находить соответствующие пары символов открывающей и закрывающей скобки и

заменять их структурными скобками. Особенность решения состоит в том, что поиск надо начинать с самых внутренних скобок и сначала надо находить закрывающую скобку, а потом – парную ей открывающую, а не наоборот, поскольку иначе поиск парной скобки сложнее. Действительно, любой закрывающей скобке соответствует ближайшая слева от неё свободная (не имеющая на данный момент пару) открывающая скобка, в то же время открывающей скобке в общем случае соответствует вовсе не ближайшая к ней справа закрывающая скобка.

Баланс скобок может быть нарушен в двух случаях: если была найдена закрывающая скобка, а перед ней не нашлось соответствующей ей открывающей скобки, и если была найдена открывающая скобка, а нужной закрывающей скобки не обнаружено. Будем считать, что вспомогательная функция `Error` обрабатывает ситуации, когда нарушен баланс скобок, и производит корректное завершение работы программы.

На языке Рефал-2 решение задачи замены символов скобок на структурные скобки состоит из двух взаимосвязанных функций: функция `RBrac` выделяет очередную закрывающую скобку, а `LBrac` – парную ей открывающую.

```
* Рефал-2: преобразование пар скобок в структурные
* Функция RBrac выделения закрывающей скобки
RBrac  e1  ')'  e2  =  <RBrac <LBrac  e1> e2>
  e1 '(' e2 = <Prout  'Баланс скобок нарушен'><Error>
    e1  =  e1
* Функция LBrac поиска парной открывающей скобки
* и перевода пары скобок в структурные
LBrac   R  e1  '('  e2  =  e1(e2)
       e1  =  <Prout  'Баланс скобок нарушен'><Error>
```

Заметим, что первое предложение функции `LBrac` содержит знак правого согласования, что означает проведение отождествления справа налево, а значит, нахождение самой правой (и ближайшей к закрывающей) открывающей скобки.

В первом предложении функции `RBrac` знак согласования не указан, по умолчанию будет применено левое согласование, и поэтому найдена первая слева самая внутренняя закрывающая скобка. Соответствующая ей открывающая скобка ищется в части текста `e1`, расположенной слева от найденной закрывающей скобки, при помощи функции `LBrac`, которая и преобразует найденную пару символьных скобок в структурные скобки. После обращения к `LBrac` функция `RBrac` рекурсивно продолжает работу – для поиска новой закрывающей скобки, и поскольку уже найденные пары скобок преобразованы в структурные, они в дальнейшем не учитываются.

Заметим, что второе предложение функции RBrac необходимо для обработки случаев, когда в обрабатываемом тексте нет закрывающей, но есть открывающая скобка, а второе предложение LBrac – когда нет открывающей, но есть закрывающая скобка.

На языке Рефал-5 функция RBrac будет выглядеть таким же образом. В то же время, поскольку в Рефале-5 отсутствует возможность правого отождествления, первое предложение функции LBrac необходимо изменить. Заметим, что выделения самого правого символа '(' можно добиться, задав для отождествляемого выражения в образце предложения дополнительное условие, что выражение e2 не должно содержать символов '('. Для проверки этого условия понадобится ввести вспомогательную функцию Check. В итоге получаем следующее решение на языке Рефал-5:

```
* Рефал-5: преобразование пар скобок в структурные
* Функция RBrac выделения закрывающей скобки
RBrac  { e.1  ')'  e.2 = <RBrac <LBrac  e.1> e.2> ;
  e.1 '(' e.2=<Prout 'Баланс скобок нарушен'><Error>;
     e.1  =  e.1   }
* Функция LBrac поиска парной открывающей скобки
* и перевода пары скобок в структурные
LBrac { e.1 '(' e.2 , <Check e.2> : True = e.1(e.2) ;
       e.1 = <Prout 'Баланс скобок нарушен'><Error> }
* Вспомогательная функция проверки, что выражение-
* аргумент не содержит открывающей скобки
Check  {  e.A  '('  e.B  =  False  ;
          e.A  =  True  }
```

Покажем изменение содержимого поля зрения рефал-машины при применении написанных функций языка Рефал-2 для обработки текста арифметического выражения '5/((A+B)^2-(A-B)^3)+8*A':

№ шага	Содержимое поля зрения рефал-машины перед началом шага
1	<RBrac '5/((A+B)^2-(A-B)^3)+8*A'>
2	<RBrac<LBrac '5/((A+B'> '^2-(A-B)^3)+8*A'>
3	<RBrac '5/('**(**'A+B'**)**'^2-(A-B)^3)+8*A'>
4	<RBrac<LBrac '5/('**(**'A+B'**)**'^2-(A-B'> '^3)+8*A'>
5	<RBrac '5/('**(**'A+B'**)**'^2-'**(**'A-B'**)**'^3)+8*A'>
6	<RBrac<LBrac '5/('**(**'A+B'**)**'^2-'**(**'A-B'**)**'^3'> '+8*A'>
7	<RBrac '5/'**((**'A+B'**)**'^2-'**(**'A-B'**)**'^3'**)**'+8*A'>
8	'5/'**((**'A+B'**)**'^2-'**(**'A-B'**)**'^3'**)**'+8*A'

Задача 5.

Исходный текст содержит символы скобок трёх видов: круглые, квадратные и фигурные. Необходимо заменить круглые скобки структурными, а выражения в парных квадратных и фигурных скобках заключить в структурные скобки, при этом оставляя на своём месте исходные символы квадратных и фигурных скобок. Например, текст `'5/([A+B]^2-{A-B}^3)+8*A'` следует преобразовать в выражение `'5/'(('[A+B]')'^2-'('{A-B}')'^3')'+8*A'`.

По сути, эта задача – обобщение и усложнение предыдущей, но основная идея обработки выражения аналогична: нужно сначала найти первую от начала текста закрывающую скобку любого вида, после чего выделить ближайшую к ней слева открывающую скобку любого вида. Для найденной пары скобок проверить, являются ли они парой скобок одного вида, и если да, то заменить их на структурные (если была пара круглых скобок) или заключить в структурные скобки (для пар скобок других видов). Если же скобки разного вида, то сообщить об отсутствии баланса скобок. При такой обработке используется свойство правильных многоскобочных выражений: выражения в разных скобках не могут пересекаться друг с другом, а могут только вкладываться друг в друга.

```
* Рефал-2: структурирование выражения, содержащего
* скобки разного вида
* Функция RBrac выделения первой закрывающей
* скобки любого вида
RBrac e1 s(')]}')A e2  =  <RBrac <LBrac  e1 sA> e2>
      e1 s('([{')B e2 = +
            <Prout 'Баланс скобок нарушен'><Error>
      e1  =  e1
* Функция LBrac поиска ближайшей слева открывающей
* скобки любого вида
LBrac   R  e1  s('([{')B  e2  =  e1 <EqBr  sB  e2>
       e1  =  <Prout  'Баланс скобок нарушен'><Error>
* Функция EqBr проверки, является ли найденная
* пара скобок скобками одного вида, и расстановки
* структурных скобок
EqBr  '(' e1 ')'  =  (e1)
      '[' e1 ']'  =  ('[' e1 ']')
      '{' e1 '}'  =  ('{' e1 '}')
      e1  =  <Prout  'Баланс скобок нарушен'><Error>
```

Покажем изменение содержимого поля зрения при обработке рассмотренными функциями текста `'5/([A+B]^2-{A-B}^3)+8*A'`:

№ шага	Содержимое поля зрения рефал-машины перед началом шага
1	`<RBrac '5/([A+B]^2-{A-B}^3)+8*A'>`
2	`<RBrac<LBrac '5/([A+B]'> '^2-{A-B}^3)+8*A'>`
3	`<RBrac'5/('<EqBr '[A+B]'> '^2-{A-B}^3)+8*A'>`
4	`<RBrac '5/('(`**`'[A+B]'`**`)'^2-{A-B}^3)+8*A'>`
5	`<RBrac<LBrac '5/('(`**`'[A+B]'`**`)'^2-{A-B}'> '^3)+8*A'>`
6	`<RBrac'5/('(`**`'[A+B]'`**`)'^2-'<EqBr '{A-B}'>'^3)+8*A'>`
7	`<RBrac'5/('(`**`'[A+B]'`**`)'^2-'(`**`'{A-B}'`**`)'^3)+8*A'>`
8	`<RBrac<LBrac '5/('(`**`'[A+B]'`**`)'^2-'(`**`'{A-B}'`**`)'^3)'> '+8*A'>`
9	`<RBrac '5/'<EqBr '('(`**`'[A+B]'`**`)'^2-'(`**`'{A-B}'`**`)'^3)'> '+8*A'>`
10	`<RBrac '5/'((`**`'[A+B]'`**`)'^2-'(`**`'{A-B}'`**`)'^3'`**`)'+8*A'>`
11	`'5/'((`**`'[A+B]'`**`)'^2-'(`**`'{A-B}'`**`)'^3'`**`)'+8*A'`

В рассмотренных функциях языка Рефал-2, решающих задачу преобразования скобок, использовались переменные со спецификатором: переменная `s(')]}')A` для поиска закрывающей скобки и переменная `s('([{')B` – для поиска открывающей скобки. Поскольку в языке Рефал-5 спецификаторы и правое согласование отсутствуют, потребуется использовать условные конструкции и дополнительную функцию `CheckInM` для проверки, что между найденной парой скобок нет других скобок.

Аргумент функции `CheckInM` имеет вид (*множество*)*выражение*, её результатом будет `True`, если никакой символ из заданного *множества* символов не входит в *выражение*, и `False` иначе.

```
* Рефал-5: структурирование выражения, содержащего
* скобки разного вида
* Функция RBrac выделения первой закрывающей скобки
* любого вида
RBrac { e.1 s.A e.2 , ')]}' : e.3 s.A e.4 =
                      <RBrac<LBrac e.1 s.A> e.2> ;
        e.1 s.B e.2 , '([{' : e.3 s.B e.4 =
             <Prout 'Баланс скобок нарушен'><Error> ;
        e.1 = e.1   }
* Функция LBrac поиска ближайшей слева (парной)
* открывающей скобки любого вида
LBrac  { e.1 s.B e2 , '([{' : e.3 s.B e.4 ,
             <CheckInM ('([{')e.2> : True
                      = e.1<EqBr s.B e.2>;
       e.1 = <Prout 'Баланс скобок нарушен'><Error>  }
```

```
* Функция EqBr проверки, является ли выделенная
* пара скобок скобками одного вида,
* и расстановки структурных скобок
EqBr { '(' e.1 ')' = (e.1) ;
       '[' e.1 ']' = ('[' e.1 ']') ;
       '{' e.1 '}' = ('{' e.1 '}') ;
      e.1 = <Prout 'Баланс скобок нарушен'><Error>  }
* Вспомогательная функция CheckInM: возвращает True,
* если никакой символ из заданного множества
* не входит в выражение, иначе возвращает False
CheckInM  { (e.1 s.X e.2) e.3 s.X e.4 = False  ;
            e.1 = True  }
```

4.3. Обработка структурированного текста

В этом подразделе рассматриваются задачи преобразования символьных выражений, в которых в результате предшествующей обработки появились структурные скобки, т.е. эти выражения были предварительно структурированы.

Задача 6.

Задано рефал-выражение, которое можно рассматривать как запись двоичного дерева с узлами, помеченными буквами или цифрами. Структура дерева фиксирована структурными скобками и описывается следующими БНФ-правилами:

дерево ::= *узел* **(***дерево* `','` *дерево***)** | *узел*

узел ::= *буква* | *цифра*

Необходимо заменить в узлах заданного дерева все символы `'A'` на символы `'B'`, а символы `'0'` – на символы `'1'`.

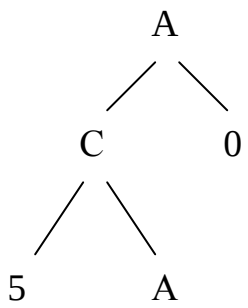

Рисунок 2

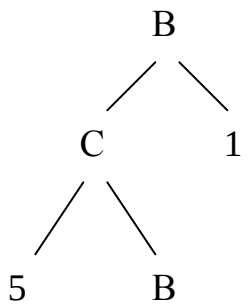

Рисунок 3

К примеру, дерево, изображённое на рисунке 2 и записывающееся как выражение `'A'('C'('5,A')',0')`, преобразуется в дерево, показанное на рисунке 3.

В случае, когда синтаксис обрабатываемого выражения описывается с помощью БНФ-правил, рефал-программу можно строить по следующей общей методике:

I. Каждому нетерминалу (структурной единице обрабатываемого выражения) соответствует обрабатывающая его рефал-функция. В нашей задаче нетерминалу *дерево* соответствует функция `Tree`, а нетерминалу *узел* – функция `Node`.

II. Если БНФ-правило для рассматриваемого нетерминала содержит несколько альтернатив, то каждой альтернативе соответствует обычно одно или несколько предложений рефал-функции. В нашей задаче двум альтернативам в правиле для нетерминала *дерево* будут соответствовать два предложения функции `Tree`.

III. При построении очередного предложения функции его левая часть получается из соответствующей альтернативы БНФ-правила заменой входящих в него нетерминалов подходящими рефал-переменными. В правой части предложения записываются необходимые преобразования над значениями этих переменных.

Определим типы рефал-переменных для нашей задачи. Нетерминалу *узел* соответствует `s`-переменная, т.к. узел – это *буква* или *цифра*, т.е. символ. Нетерминалу *дерево* нельзя поставить в соответствие `s`-переменную или `w`-переменную (в Рефале-5 – `t`-переменную), поскольку первая альтернатива в правиле для этого нетерминала описывает более сложное выражение, чем символ или структурный терм. Это выражение состоит из символа (изображающего *узел*), за которым расположен терм **(***дерево* `','` *дерево***)**. Поэтому нетерминалу *дерево* соответствует `e`-переменная в Рефале-5 и `v`-переменная в Рефале-2 (дерево не может быть пустым).

Согласно рассмотренной методике получаем следующее определение функций языка Рефал-2:

```
* Функция Tree обработки (анализа) дерева
Tree   sA (v1 ',' v2) = <Node sA>  +
                        (<Tree v1> ',' <Tree v2>)
       sA  =  <Node  sA>
* Функция замены символов в узлах дерева
Node   'A'  =  'B'
       '0'  =  '1'
        s1  =  s1
```

Функция `Tree` осуществляет проход по структуре дерева, вызывая при этом функцию `Node`, которая осуществляет необходимую замену символов дерева. Правая часть первого предложения функции `Tree` содержит 2 рекурсивных вызова для обработки соответственно левого и правого поддерева. Поскольку сама структура дерева не меняется, то выражение в правой части этого предложения отличается от выражения в левой части только функциональными вызовами.

Рассмотренное определение функции написано в предположении, что её аргумент синтаксически правилен (относительно заданных БНФ-правил), в ином случае в функцию следует добавить третье предложение, фиксирующее синтаксическую ошибку:

```
E1 = <Prout 'Ошибка в описании дерева: 'E1> +
             <Error>
```

Рассмотрим изменение поля зрения рефал-машины при обработке приведённого выше примера дерева, задаваемого выражением `'A'('C'('5,A')',0')`:

№ шага	Содержимое поля зрения рефал-машины перед началом шага
1	`<Tree 'A'('C'('5,A')',0')>`
2	`<Node 'A'>(<Tree 'C'('5,A')>','<Tree '0'>)`
3	`'B'(<Tree 'C'('5,A')>','<Tree '0'>)`
4	`'B'(<Node 'C'>(<Tree '5'>','<Tree 'A'>)','<Tree '0'>)`
5	`'B'('C'(<Tree '5'>','<Tree 'A'>)','<Tree '0'>)`
6	`'B'('C'(<Node '5'>','<Tree 'A'>)','<Tree '0'>)`
7	`'B'('C'('5,'<Tree 'A'>)','<Tree '0'>)`
8	`'B'('C'('5,'<Node 'A'>)','<Tree '0'>)`
9	`'B'('C'('5,B')','<Tree '0'>)`
10	`'B'('C'('5,B')','<Node '0'>)`
11	`'B'('C'('5,B')',1')`

Таким образом, на шаге 11 в поле зрения рефал-машины получено преобразованное дерево.

Задача 7.

Преобразовать заданное арифметическое выражение из обычной инфиксной записи в ПОЛИЗ (*польскую инверсную запись*, при которой операция записывается после своих операндов). Арифметическое выражение состоит из чисел (уже преобразованных во внутреннее

рефальское представление – символ-число для Рефала-2 или макроцифра для Рефала-5), знаков операций сложения, вычитания, умножения и деления, а также круглых скобок. Например, для выражения в инфиксной записи `5*(2-3)+8/(2*4-7)-11*6` необходимо получить его ПОЛИЗ `5 2 3 - * 8 2 4 * 7 - / + 11 6 * -`.

Предполагается, что скобки в заданном арифметическом выражении уже преобразованы в структурные.

Структура выражения задаётся БНФ-правилами:

выражение ::= *выражение знак-add-sub слагаемое* | *слагаемое*

слагаемое ::= *слагаемое знак-mul-div множитель* | *множитель*

множитель ::= *число* | (*выражение*)

знак-add-sub ::= `'+'` | `'-'`

знак-mul-div ::= `'*'` | `'/'`

При описании нетерминалов *выражение* и *слагаемое* использована левая рекурсия. При синтаксическом разборе сверху вниз такое описание соответствует общепринятому порядку вычисления арифметических операций одного приоритета – слева направо. При таком порядке отсутствующие в выражении скобки, например, `7+12-23+5` восстанавливаются следующим образом: `((7+12)-23)+5`, и самый левый знак операции находится в глубине всех скобок, в то время как самый правый знак операции остался на верхнем уровне выражения. Если же необходимо выполнять вычисление операций справа налево, то при описании соответствующих нетерминалов следует использовать правую рекурсию. Приведённые БНФ-правила учитывают не только порядок вычисления операций одного приоритета, но и приоритеты операций – операции умножения и деления имеют больший приоритет, чем операции сложения и вычитания.

Для преобразования арифметического выражения в ПОЛИЗ необходимо на всех его уровнях после выделения в нём очередного знака операции и соответствующих операндов преобразовать их в ПОЛИЗ, используя рекурсию, а знак операции записать после результатов преобразования операндов.

Согласно методике построения рефал-программы, изложенной в задаче 6, для каждого из нетерминалов БНФ-правил *выражение, слагаемое* и *множитель* организуем отдельную функцию – соответственно `PolizExp`, `PolizSum` и `PolizMult`. Эти функции будут переводить в ПОЛИЗ соответствующую структурную единицу выражения. Количество рефал-предложений в теле каждой функции будет совпадать с количеством альтернатив в БНФ-правиле нетерминала, соответствующего этой функции. Левые части рефал-предложений строятся также согласно

методике, но с учётом левой рекурсии в описании нетерминалов *выражение* и *слагаемое* – это означает, что следует выделять самый правый знак арифметической операции на рассматриваемом уровне обрабатываемого выражения. Заметим, что для нетерминалов *знак-add-sub* и *знак-mul-div* отдельные функции не заводятся (необходимые для их обработки действия будут выполнять обозначенные выше функции).

Определим теперь соответствие типов рефал-переменных описанным нетерминалам. Нетерминалам *знак-add-sub* и *знак-mul-div* (символы операций) соответствуют s-переменные, в языке Рефал-2 – соответственно со спецификатором `'+-'` или `'*/'`. Нетерминалу *множитель* соответствует w-переменная в языке Рефал-2 и t-переменная в языке Рефал-5, поскольку множитель является либо числом, либо выражением в структурных скобках (т.е. термом). Нетерминалам *выражение* и *слагаемое* соответствует v-переменная языка Рефал-2 (поскольку они не могут быть пустыми) и e-переменная языка Рефал-5.

Чтобы в функции `PolizExp` выделить самый правый знак сложения или вычитания, в первом её предложении следует использовать правое согласование. В то же время в функции `PolizSum` нет необходимости использовать правое согласование – действующее по умолчанию левое согласование обеспечит выделение последнего знака умножения или деления, поскольку слагаемое состоит из множителей, а *множитель* является термом (однозначно проектирующимся жёстким элементом выражения).

Функция `PolizMult` написана с учётом того, что ПОЛИЗ числа есть само число, а при переводе в ПОЛИЗ арифметического выражения в скобках эти скобки убираются.

Таким образом, на языке Рефал-2 рассматриваемую задачу решают следующие функции:

```
* Рефал-2: перевод в ПОЛИЗ арифметического выражения
PolizExp  R v1 s('+-')2 v3 = <PolizExp v1> +
                              <PolizSum v3>s2
            v1 = <PolizSum  v1>
* Перевод в ПОЛИЗ слагаемого
PolizSum   v1 s('*/')2 w3 = <PolizSum v1> +
                               <PolizMult w3>s2
           w1 = <PolizMult w1>
* Перевод в ПОЛИЗ множителя
PolizMult  s(N)1 = s1
           (v1) = <PolizExp v1>
```

Отметим, что порядок следования рефал-предложений существенен лишь в функции PolizExp, поскольку образец её второго предложения может быть синтаксически отождествлён с теми же объектными выражениями, для обработки которых предназначено первое предложение этой функции. Рефал-предложения в телах функций PolizSum и PolizMult можно менять местами, поскольку множества выражений, с которыми могут отождествляться образцы этих предложений, не пересекаются. В частности, в функции PolizSum образец второго предложения отождествляется только с термом, в то время как образец первого предложения отождествляется только с выражениями, состоящими как минимум из трёх термов.

Рассмотрим изменение поля зрения рефал-машины при переводе в ПОЛИЗ выражения /5/'*'(/2/'-'/3/)'+'/8/'/'/6/'-'/11/, с помощью рассмотренных функций языка Рефал-2:

№ шага	Содержимое поля зрения рефал-машины перед началом шага
1	<PolizExp /5/'*'(/2/'-'/3/)'+'/8/'/'/6/'-'/11/>
2	<PolizExp /5/'*'(/2/'-'/3/)'+'/8/'/'/6/> <PolizSum /11/> '-'
3	<PolizExp /5/'*'(/2/'-'/3/)><PolizSum /8/'/'/6/>'+' <PolizSum /11/> '-'
4	<PolizSum /5/'*'(/2/'-'/3/)><PolizSum /8/'/'/6/>'+' <PolizSum /11/> '-'
5	<PolizSum /5/><PolizMult (/2/'-'/3/)>'*'<PolizSum /8/'/'/6/>'+' <PolizSum /11/> '-'
6	<PolizMult /5/><PolizMult (/2/'-'/3/)>'*'<PolizSum /8/'/'/6/>'+' <PolizSum /11/> '-'
7	/5/<PolizMult (/2/'-'/3/)>'*'<PolizSum /8/'/'/6/>'+' <PolizSum /11/> '-'
8	/5/<PolizExp /2/'-'/3/>'*'<PolizSum /8/'/'/6/>'+' <PolizSum /11/> '-'
9	/5/<PolizExp /2/><PolizSum /3/>'-*'<PolizSum /8/'/'/6/>'+' <PolizSum /11/> '-'
10	/5/<PolizSum /2/><PolizSum /3/>'-*'<PolizSum /8/'/'/6/>'+' <PolizSum /11/> '-'
11	/5/<PolizMult /2/><PolizSum /3/>'-*'<PolizSum /8/'/'/6/>'+' <PolizSum /11/> '-'
12	/5/ /2/<PolizSum /3/>'-*'<PolizSum /8/'/'/6/>'+' <PolizSum /11/> '-'
13	/5/ /2/<PolizMult /3/>'-*'<PolizSum /8/'/'/6/>'+' <PolizSum /11/> '-'

14	/5/ /2/ /3/ '-*'<PolizSum /8/''/6/>'+' <PolizSum /11/> '-'
15	/5/ /2/ /3/ '-*'<PolizSum /8/><PolizMult /6/>'/+' <PolizSum /11/> '-'
16	/5/ /2/ /3/ '-*'<PolizMult /8/><PolizMult /6/>'/+' <PolizSum /11/> '-'
17	/5/ /2/ /3/ '-*' /8/<PolizMult /6/>'/+' <PolizSum /11/> '-'
18	/5/ /2/ /3/ '-*' /8/ /6/ '/+' <PolizSum /11/> '-'
19	/5/ /2/ /3/ '-*' /8/ /6/ '/+' <PolizMult /11/> '-'
20	/5/ /2/ /3/ '-*' /8/ /6/ '/+' /11/ '-'

Для решения рассматриваемой задачи перевода выражения в ПОЛИЗ на основе языка Рефал-5 опять же, как и в задаче 5, потребуется введение дополнительных условий в левых частях предложений и использование функции `CheckInM`:

```
* Рефал-5: перевод в ПОЛИЗ арифметического выражения
PolizExp { e.1 s.2 e.3 , '+-' : e.A s.2 e.B ,
             <CheckInM ('+-')e.3> : True  =
                 <PolizExp e.1><PolizSum e.3> s.2 ;
           e.1 = <PolizTerm e.1>   }
* Перевод в ПОЛИЗ слагаемого
PolizSum {  e.1 s.2 t.3 , '*/' : e.A s.2 e.B  =
                 <PolizSum e.1><PolizMult t.3> s.2 ;
           t.1 = <PolizMult t.1>   }
* Перевод в ПОЛИЗ множителя
PolizMult    {  s.1 , <Type  s.1> : 'N' e.2 = s.1 ;
          (e.1) = <PolizExp e.1>  }
* Функция проверки, что никакой символ из заданного
* множества не входит в выражение
CheckInM  { (e.1 s.X e.2) e.3 s.X e.4 = False ;
             e.1 = True  }
```

Отметим, что из-за отсутствия в языке Рефал-5 правого отождествления и спецификаций переменных получившееся решение (как и решение задачи 5) состоит, по сравнению с решением на языке Рефал-2, из большего числа функций с более длинными предложениями.

5. Задания практикума

Все практические задания предполагают программирование типичных задач анализа и преобразования символьных данных: записей математических выражений, программ или их фрагментов.

5.1. Дифференцирование выражения

Требуется составить рефал-программу, выполняющую дифференцирование заданного выражения, включающего алгебраические операции и известные математические функции. Дифференцирование производится по одной из входящих в это выражение переменных.

Входные данные программы имеют вид **D***переменная***:** *выражение*, после буквы **D** указывается имя переменной дифференцирования, а после двоеточия – дифференцируемое выражение, например:

```
Dy:sin(x+2*y)*y^-4+(8*a-b)*sqrt(sin(y-2)+cos(y+9))
```

Синтаксис дифференцируемого выражения описывается с помощью следующих БНФ-правил:

выражение ::= *слагаемое* | *выражение знак_суммирования слагаемое*

знак_суммирования ::= **+** | **–**

слагаемое ::= *множитель* | *слагаемое знак_умножения множитель*

знак_умножения ::= ***** | **/**

множитель ::= *целое_без_знака* | *переменная* | *переменная***^***степень*
| *имя_функции*(*выражение*) | (*выражение*) |
(*выражение*)**^***степень*

переменная ::= *идентификатор*

степень ::= *знак целое_без_знака*

знак ::= *пусто* | **+** | **–**

имя_функции ::= `sin | cos | ln | exp | sqrt | arcsin | arccos`

Рефал-программа должна выполнять следующие действия:

- Проверку синтаксической правильности заданного выражения согласно приведённым БНФ;
- дифференцирование введённого выражения по заданной переменной;
- упрощение результата дифференцирования (вычисление арифметических операций над числами, удаление нулевых слагаемых и

единичных множителей, единичных степеней, лишних скобок, а также приведение подобных слагаемых и сокращение дробей);

- вывод на экран результирующего выражения.

В случае ввода синтаксически неверного выражения в качестве ответа должно быть выдано диагностическое сообщение. Должны быть выявлены и диагностированы следующие виды синтаксических ошибок:

- нарушен баланс открывающих и закрывающих скобок;
- пропущен знак операции или её операнд;
- употребление неизвестной функции;
- неправильная запись выражения со степенью;
- неверная запись множителя – числа, идентификатора, обращения к функции.

5.2. Решение системы линейных уравнений

Дана система из `n` линейных уравнений с `m` неизвестными, `n` и `m` ≥ 1, например (`m`=2, `n`=2):

```
3*x+2*x-16/(1+7)*y=11
x+2*y=7
```

Коэффициентами при переменных уравнения могут быть как числа, так и составленные из них арифметические выражения (с использованием операций умножения, сложения, вычитания, деления и круглых скобок).

Необходимо составить рефал-программу, решающую эту систему *методом исключения переменных*.

В общем случае решение системы включает в себя следующие шаги:

1) Все уравнения системы преобразуются к каноническому виду:
   `a*x+b*y+c*z+...+d=0`
   где `a`, `b`, `c`, `d` – числа.
2) В одном из уравнений некоторая переменная выражается через другие переменные:
   `x=-(b*y+c*z+...+d)/a = -b/a*y-c/a*z-...-d/a`.
3) Полученное для этой переменной выражение подставляется в остальные уравнения системы вместо всех вхождений указанной переменной.

В результате выполнения этих трёх шагов получается формула для вычисления значения одной из переменных системы по значениям других переменных, а также новая равносильная система уравнений, содержащая

на одно уравнение и на одну переменную меньше, чем предыдущая. Далее шаги 1-3 повторяются для этой новой системы уравнений, и так продолжается до тех пор, пока не останется одно уравнение или пока не обнаружится противоречие.

Противоречие (неверное равенство) возникает, когда система не имеет решения (например, в ходе преобразований получено уравнение `6-4=0`). В этом случае рефал-программа должна в качестве ответа выдать сообщение

```
Система уравнений не имеет решения
```

Если в ходе преобразований одно из уравнений превратилось в тождество (`0=0`), то исходная система уравнений была избыточна, а это уравнение можно исключить из решаемой системы.

Если в ходе преобразований системы получено одно уравнение с одной переменной, то из этого уравнения вычисляется значение этой переменной. Значения остальных переменных вычисляются по полученным в ходе преобразований формулам. В этом случае рефал-программа должна выдать решение системы уравнений в виде строк вида *имя_переменной=значение*, например, для системы уравнений, приведённой выше в качестве примера, ответом будет:

```
x=3
y=2
```

Если же в результате преобразований системы уравнений остаётся одно уравнение, но оно содержит более одной переменной (например, `x+y+2=0`), то это означает, что исходная система имеет бесконечное множество решений. В этом случае рефал-программа должна вывести на печать имена *свободных* переменных (которые могут принимать любые значения) и формулы для вычисления остальных переменных системы по значениям этих свободных переменных. Например, для заданной системы уравнений:

```
x+2*y-z=2
2*x-y+3*z=-6
```

рефал-программа может выдать результат в виде:

```
x=-z-2
y=z+2
z - свободная переменная
```

5.3. Определение равносильности логических формул

Составить программу, проверяющую равносильность (эквивалентность) двух заданных формул алгебры логики. В формулах используются логические константы TRUE и FALSE, логические переменные и логические операции: отрицания (¬), конъюнкции (∧), дизъюнкции (∨), импликации (→) и эквиваленции (≡), а также круглые скобки. В качестве имен переменных могут быть взяты произвольные идентификаторы. Например, логической формулой является запись a1 → ¬ (b1 ∧¬ a2 ∨ c) ≡ (c ∧ b2).

Порядок выполнения операций в формуле алгебры логики определяется согласно общепринятому приоритету логических операций и записанным скобкам. Операция эквиваленции ≡ имеет меньший приоритет, чем операция импликации →, а импликация – меньший, чем операции отрицания, конъюнкции и дизъюнкции.

В логической формуле могут быть опущены незначащие скобки. Пара скобок считается *незначащей* (избыточной), если после её удаления получается формула, равносильная исходной.

Две логические формулы называются *равносильными*, если для любого набора значений входящих в них переменных значения этих формул совпадают (т.е. они реализуют одну и ту же логическую функцию). Например, равносильны формулы x → y и ¬x ∨ y.

Логические формулы и реализуемые ими функции могут содержать фиктивные (несущественные) переменные. Переменная x_k является *фиктивной* для функции $f(x_1, \dots, x_n)$, $n \geq 1$, $k \leq n$, если для любого набора логических значений $a_1, \dots, a_{k-1}, a_{k+1}, \dots, a_n$

$f(a_1,\dots,a_{k-1},0,a_{k+1},\dots,a_n) \equiv f(a_1,\dots,a_{k-1},1,a_{k+1},\dots,a_n)$

Например, фиктивной является переменная x для функции, реализуемой формулой (¬ x ∨ x) ≡ y ∧ z.

Программа проверки равносильности формул выполняет:

- ввод исходных логических формул с клавиатуры или из файла (направление ввода определяется командами пользователя или выясняется в диалоге с ним);
- проверку синтаксической правильности введённых формул и выдачу диагностических сообщений в случае обнаружения синтаксических ошибок;
- проверку равносильности двух синтаксически правильных логических формул;

- нахождение для каждой из двух формул всех фиктивных переменных и их вывод.

В ходе выполнения задания необходимо описать синтаксис формул алгебры логики в виде БНФ-правил.

Рефал-программа должна выявлять и диагностировать следующие виды синтаксических ошибок:

- нарушение баланса открывающихся и закрывающихся скобок;
- недопустимая операция;
- пропуск операции или операнда;
- неверная запись логической константы или переменной.

5.4. Распознавание вхождения логической формулы

Рассматриваются формулы алгебры логики, в которых используются логические константы `TRUE` и `FALSE`, логические переменные и логические операции: отрицания (¬), конъюнкции (∧), дизъюнкции (∨), импликации (→), а также круглые скобки. В качестве имён переменных могут быть взяты латинские буквы. Например, логической формулой является запись `a → ¬ (b ∧ ¬c ∨ d)` .

Порядок выполнения операций в формуле алгебры логики определяется согласно общепринятому приоритету логических операций и записанным скобкам (импликация → имеет наименьший приоритет).

В логической формуле могут быть опущены незначащие скобки. Пара скобок считается *незначащей* (избыточной), если после её удаления получается формула, равносильная исходной (две логические формулы называются *равносильными*, если для любого набора значений входящих в них переменных значения этих формул совпадают).

Говорят, что имеет место *прямое вхождение* логической формулы β в логическую формулу α (т.е. она является *подформулой* α), если она в точности совпадает с ней или же является операндом одной из применённых в ней логических операций. Например, формула `p ∧ q` – подформула формулы `p ∧ q ∨ ¬ r`, а формула `q ∨ ¬ r` – нет.

Будем говорить, что имеется *непрямое вхождение* формулы β в логическую формулу α, если существует прямое вхождение в формулу α некоторой эквивалентной формулы β', получающейся из β несколькими перестановками операндов бинарных операций конъюнкции и дизъюнкции в формуле β. Например, формула `q ∧ p` имеет непрямое вхождение в формулу `p ∧ q ∨ ¬ r`.

Необходимо составить рефал-программу, которая проверяет, содержит ли заданная логическая формула α хотя бы одно вхождение (прямое или непрямое) логической формулы β. Программа осуществляет:

- ввод исходных логических формул с клавиатуры или из файла;
- проверку синтаксической правильности введённых формул и выдачу диагностических сообщений в случае обнаруженных ошибок;
- распознавание прямого вхождения формулы β в α, при успешном распознавании выводится формула α , в которой помечена (тем или иным способом) подформула β;
- проверку на непрямое вхождение формулы β в α, в случае успешной проверки выводится эквивалентная β' и помечается её местоположение в формуле α;
- анализ, можно ли переименовать в формуле β переменные таким образом, чтобы результирующая формула стала подформулой α – если это возможно, то выводится результирующая подформула и список сделанных в формуле β переименований переменных. Например, формула `x ∧ y` входит в формулу `p ∧ q ∨ ¬ r` при следующем переименовании переменных: `x` ↔ `p`, `y` ↔ `q`.

Для выполнения задания необходимо описать синтаксис формул алгебры логики в виде БНФ-правил и использовать их при написании рефал-программы.

5.5. Вычисление выражения языка C

Составить рефал-программу, вводящую выражение, записанное на языке программирования C, и вычисляющую его значение. Пример такого выражения:

```
x=2,y=3,z=x+=2*y-3*x?++y>4?(8-4)*x:25:x--&&y/3?x<<3:5
```

Выражение состоит из целых констант, имён переменных, круглых скобок и может содержать следующие знаки операций:

- ✓ постфиксные и префиксные `++` и `--`;
- ✓ унарные и бинарные `+` и `-`;
- ✓ арифметические `*`, `/` и `%`;
- ✓ побитовые `<<` и `>>`;
- ✓ логические `!`, `&&`, `||`;
- ✓ операции отношения `!=`, `==`, `>`, `<`, `>=`, `<=`;
- ✓ тернарная операция `?:`;

- ✓ операции присваивания =, +=, -=, *=, /=, %=, >>=, <<= ;
- ✓ операция запятая (,).

Приоритет и порядок вычисления этих операций соответствуют принятым в языке С [11] правилам – см. таблицу 1 (в ней операции располагаются по строкам в порядке убывания приоритета).

Синтаксис выражения языка С описывается следующими БНФ-правилами:

выражение ::= *выражение_присваивания* |
выражение , *выражение_присваивания*

выражение_присваивания ::= *условное_выражение* |
идентификатор операция_присваивания выражение_присваивания

операция_присваивания ::= = | += | -= | *= | /= | %= | <<= | >>=

условное_выражение ::= *логическое_ИЛИ_выражение* |
логическое_ИЛИ_выражение ? *выражение* : *условное_выражение*

логическое_ИЛИ_выражение ::= *логическое_И_выражение* |
логическое_ИЛИ_выражение || *логическое_И_выражение*

. . .

слагаемое ::= *унарное_выражение* |
слагаемое мульт_знак унарное_выражение

мульт_знак ::= * | / | %

унарное_выражение ::= *постфиксное_выражение* |
унар_знак унарное_выражение

унар_знак ::= ! | ++ | -- | + | -

постфиксное_выражение ::= *первичное_выражение* |
постфиксное_выражение1

постфиксное_выражение1::= *идентификатор* |
постфиксное_выражение1 постф_знак

постф_знак ::= ++ | --

первичное_выражение ::= *идентификатор* | *целое_без_знака* |
(*выражение*)

При выполнении задания необходимо дописать недостающие БНФ-правила, и согласно получившемуся набору синтаксических правил составить рефал-программу, вычисляющую выражение языка С.

Таблица 1. Приоритет и порядок вычисления операций языка С

Приоритет операций (в порядке убывания)	порядок вычисления
постфиксные ++ и --	слева направо
! префиксные ++ и -- унарные + и -	справа налево
* / %	слева направо
бинарные + и -	слева направо
<< >>	слева направо
< <= > >=	слева направо
== !=	слева направо
&&	слева направо
||	слева направо
? :	справа налево
= += -= *= /= %= <<= >>=	справа налево
,	слева направо

5.6. Интерпретация паскаль-программы

Рассматривается задача интерпретации программы, записанной на подмножестве языка Паскаль. Синтаксис подмножества задаётся следующими БНФ-правилами. Использованные в правилах фигурные скобки означают повторение заключённой в них конструкции 0, 1 или произвольное количество раз.

программа ::= `program` *имя*`;` *раздел_описаний раздел_операторов*
имя ::= *идентификатор*
раздел_описаний ::= *раздел_констант раздел_переменных*
раздел_констант ::= *пусто* | `const` *имя*=*конст*`;` {*имя*=*конст*`;`}
раздел_переменных ::= `var` *секция* {`;` *секция*}`;`
секция ::= *имя* {`,` *имя*} `:` *тип*
тип ::= `integer` | `boolean`

конст ::= *знак цифра* {*цифра*}
знак ::= *пусто* | + | −
конст._бз ::= *цифра* {*цифра*}
логич._значение::= `true` | `false`

раздел_операторов ::= `begin` *оператор* {`;` *оператор*} `end.`
оператор ::= *оператор_присваивания* | *оператор_вывода* |
составной_оператор | *оператор_цикла_с_постусловием* |
оператор_цикла_с_предусловием | *оператор_цикла_for* |
оператор_выбора | *условный_оператор*

оператор_присваивания ::= *имя*:=*выражение*
выражение ::= *простое_выражение* |
простое_выражение операция_отношения простое_выражение
операция_отношения ::= > | < | <> | <= | >= | =
простое_выражение ::= *слагаемое* |
простое_выражение операция_сложения слагаемое
операция_сложения ::= + | - | or
слагаемое ::= *множитель* |
слагаемое операция_умножения множитель
множитель ::= *имя* | (*выражение*) | *конст._бз* | *логич._значение* |
not *множитель*
операция_умножения ::= * | div | mod | and

оператор_вывода ::= writeln(*список_элементов*)
список_элементов ::= *элемент*{,*элемент*}
элемент ::= *выражение*

составной_оператор ::= begin *оператор*{; *оператор*} end
оператор_цикла_с_постусловием ::=
repeat *оператор*{; *оператор*} until *выражение*
оператор_цикла_с_предусловием ::= while *выражение* do *оператор*
оператор_цикла_for ::=
for *имя*:=*выражение* to *выражение* do *оператор*

оператор_выбора ::=
case *выражение* of *вариант* {; *вариант*} endcase
вариант ::= *метка_варианта* : *оператор*
метка_варианта ::= *конст* | *логическое_значение*

условный_оператор ::=
if *выражение* then *оператор* else *оператор*

Семантика описанных конструкций эквивалентна семантике соответствующих конструкций стандарта языка Паскаль [12].

Требуется написать на языке Рефал интерпретатор паскаль-программ, который выполняет:

- ввод паскаль-программы из файла;
- лексический и синтаксический анализ введённой программы с выдачей диагностических сообщений в случаях обнаружения в ней ошибок;
- интерпретацию (выполнение) синтаксически правильной паскаль-программы с выдачей сообщений об ошибках, обнаруженных в ходе выполнения паскаль-программы.

В ходе лексического и синтаксического анализа должен быть проверен синтаксис конструкций, а также выполнен контроль контекстных условий (в том числе контроль типов). Перечень обнаруживаемых ошибок включает:

- нарушение баланса открывающих и закрывающих скобок в выражениях;
- неверная запись или пропуск операций и операндов в выражении;
- нарушения синтаксиса записи операторов;
- в том числе – баланса операторных скобок `begin` и `end`, а также `case` и `endcase`;
- присваивание значения константе;
- неописанный или дважды описанный идентификатор;
- несоответствие типов в операторе присваивания и в выражении;
- неверный тип выражения в условном операторе и операторе цикла.

В процессе интерпретации следует выявлять такие ошибки, как переменная без значения, деление на ноль, зацикливание. Требуется также контролировать изменение параметра цикла в цикле `for`, поскольку присваивание переменной-параметру цикла в теле цикла запрещено.

В случае обнаружения ошибки в ходе интерпретации рефал-программа должна выдать соответствующее диагностическое сообщение, после чего по возможности продолжить дальнейшее её выполнение.

5.7. Трансляция паскаль-программы в язык С

Составить рефал-программу, преобразующую текст программы, записанной на подмножестве языка Паскаль [12] в текст эквивалентной программы, написанной на языке С [11]. В качестве подмножества Паскаля можно взять язык, описанный в предыдущем варианте задания, дополнив его процедурами и функциями.

В функции программы-транслятора входит:

- ввод паскаль-программы из текстового файла;
- лексический и синтаксический анализ введённой программы с выдачей сообщений о найденных ошибках;
- преобразование синтаксически правильной паскаль-программы в эквивалентную программу на языке С;
- выполнение некоторых преобразований полученной С-программы, упрощающих её или оптимизирующих проводимые вычисления (например, преобразующих выражение присваивания `i+=1` в `++i`);

- вывод (печать) результирующей программы с учетом принятых для этого языка правил структурирования вложенных конструкций.

Для выполнения задания необходимо дополнить необходимыми БНФ-правилами описание подмножества Паскаля, приведённое в предыдущем варианте задания.

5.8. Методические указания к вариантам

Во всех вариантах рекомендуется выделить следующие предварительные этапы обработки исходных символьных выражений – *лексический* и *синтаксический* анализ.

Задача лексического анализа – выделение лексем исходного выражения и, возможно, перевод их во внутреннее представление, в котором:

- числовые константы (последовательности цифр) преобразованы в символы-цифры или макроцифры;
- имена переменных (последовательности букв и цифр, начинающиеся с буквы) заключены в структурные скобки или преобразованы в символ-метки;
- имена функций (`sin`, `cos` и др. в варианте дифференцирования выражения) или служебные имена (`begin`, `end`, `case`, `integer` и др. в вариантах интерпретации и трансляции паскаль-программы) преобразованы в соответствующие символы-метки (`/sin/`, `/cos/`, `/begin/` и т.п.);
- знаки операций, состоящие из нескольких символов, заменены на соответствующие им символы-метки (например, в варианте вычисления выражения языка С: знаки операции `++` можно заменить на символ-метку `/pp/`, а знаки `*=` на символ-метку `/mulassign/`);
- унарные знаки + и − заменены на другие символы – для того, чтобы отличать их от знаков бинарных операций + и −.

Основная задача синтаксического анализа – выявление структуры обрабатываемого выражения и перевод его во *внутреннее представление*, удобное для дальнейших преобразований. Для этого:

- необходимо заменить в исходном выражении символы круглых скобок на структурные скобки;
- в варианте вычисления выражения языка С в ходе расстановки структурных скобок целесообразно считать знаки тернарной операции `?` и `:` особым видом парных скобок и заменить каждую пару таких

символов на пару структурных скобок и символов-меток, например, `/question/` и `/two-spot/`, или же заключить в структурные скобки каждую пару символов `?:` вместе со стоящим между ними выражением;

- в вариантах интерпретации и трансляции паскаль-программы после замены обычных круглых скобок на структурные следует зафиксировать вложенность операторов паскаль-программы расстановкой структурных скобок вокруг фрагментов операторов, начинающихся и заканчивающихся соответственно символами-метками `/begin/` и `/end/`, `/then/` и `/else/`, `/case/` и `/endcase/`, `/repeat/` и `/until/` (введёнными на этапе лексического анализа).

Таким образом, в ходе лексического и синтаксического анализа входное символьное выражение будет преобразовано во внутреннее представление, упрощающее его последующую обработку (дифференцирование, вычисление, интерпретацию, трансляцию и т.п. – в зависимости от варианта задания).

Для хранения значений переменных в ходе вычислений целесообразно использовать копилку.

6. Литература

1. Турчин В.Ф. Алгоритмический язык рекурсивных функций (РЕФАЛ). М: ИПМ АН СССР, Препринт № 4, 1968.
2. Романенко С.А., Турчин В.Ф. РЕФАЛ-компилятор. // Труды 2-й Всесоюзной конференции по программированию. ВЦ СОАН. Новосибирск, 1970.
3. Турчин В.Ф. Базисный РЕФАЛ. Описание языка и основные приемы программирования (метод. рекомендации) . Фонд алгоритмов и программ в отрасли "Строительство", vol. 5, N 33. ЦНИПИАСС. М: 1974.
4. Климов Анд.В., Климов Арк.В., Красовский А.Г., Романенко С.А., Травкина Е.В., Турчин В.Ф., Хорошевский В.Ф., Щенков И.Б. Базисный РЕФАЛ и его реализация на вычислительных машинах (метод. рекомендации). Фонд алгоритмов и программ для ЭВМ (в отрасли "Строительство"), специальный раздел, vol. 5, N 40. М: 1977.
5. Климов А.В., Романенко С.А. Метавычислитель для языка Рефал. Основные понятия и примеры. М: ИПМ АН СССР, препринт № 71, 1987.
6. Романенко С.А. Метаалгоритмический язык Рефал и тенденции его развития // Искусственный интеллект в 3-х кн., Кн. 3. Программные и аппаратные средства: Справочник / под ред.Захарова В.Н., Хорошевского В.Ф., – М: Радио и связь, 1990, стр. 47–55.
7. Марков А.А. Нагорный Н.М. Теория алгоритмов. – М.: Фазис, 1996.
8. Романенко С.А. Реализация Рефала-2. М: ИПМ АН СССР, препринт № 71, 1987.
9. Turchin V. REFAL-5, Programming Guide and Reference Manual – New England Publishing Co., Holyoke, 1989.
10. Турчин В.Ф. РЕФАЛ-5. Руководство по программированию и справочник. http://www.refal.net/rf5_frm.htm
11. Керниган Б., Ритчи Д. Язык программирования Си. – М: Финансы и статистика, 1992.
12. Йенсен К., Вирт Н. Паскаль. Руководство для пользователя и описание языка. – М: Финансы и статистика, 1982.

Приложение 1. Синтаксис языка Рефал-2

Рефал-программа ::= *имя_программы* `START`
рефал-предложения_и_директивы
`END`
имя_программы ::= *идентификатор*

рефал-предложения_и_директивы ::= *единица_программы* |
единица_программы
рефал-предложения_и_директивы
единица_программы ::= *директива* | *описание_спецификатора* |
описание_функции

директива ::= `ENTRY`-*директива* | `EXTRN`-*директива* | `EMPTY`-*директива*
`ENTRY`-*директива* ::= `ENTRY Go`

`EXTRN`-*директива* ::= `EXTRN` *список_объявлений_функций*
список_объявлений_функций ::= *объявление_функции* |
объявление_функции `,` *список_объявлений_функций*
объявление_функции ::= *идентификатор* |
идентификатор `(`*внешний_идентификатор*`)`

`EMPTY`-*директива* ::= `EMPTY` *список_идентификаторов*
список_идентификаторов ::= *идентификатор* |
идентификатор `,` *список_идентификаторов*

описание_спецификатора ::= *имя_спецификатора* `S` *набор_ограничений*
имя_спецификатора ::= *идентификатор*
набор_ограничений ::= *цепочка_элементов_спецификации* |
цепочка_элементов_спецификации
`(`*цепочка_элементов_спецификации*`)` *набор_ограничений*
цепочка_элементов_спецификации ::= *пусто* |
элемент_спецификации *цепочка_элементов_спецификации*
элемент_спецификации ::= *символ-литера* | *символ-число* | *символ-метка* |
стандартное_множество | *спецификатор*

спецификатор ::= `:`*имя_спецификатора*`:`

символ-литера ::= `'`*символ*`'`
символ-число ::= `/`*последовательность_цифр*`/`
последовательность_цифр ::= *цифра* |
цифра *последовательность_цифр*
символ-метка ::= `/`*идентификатор*`/`

стандартное_множество ::= `S | F | N | O | L | D | B | W`

описание_функции ::= *имя_функции послед-ть_рефал-предложений* |
`Go` = *начальное_состояние_поля_зрения*
имя_функции ::= *идентификатор*
начальное_состояние_поля_зрения ::= *рабочее_выражение*

послед-ть_рефал-предложений ::= *пусто* |
рефал-предложение послед-ть_рефал-предложений
рефал-предложение ::=
знак_отождествления выражение-образец = выражение
знак_отождествления ::= *пусто* | `L | R`

выражение-образец ::= *пусто* | *терм1 выражение-образец*
терм1 ::= *символ-литера* | *символ-число* | *символ-метка* |
переменная | (*выражение-образец*)

выражение ::= *пусто* | *терм выражение*
терм ::= *символ-литера* | *символ-число* | *символ-метка* |
переменная | (*выражение*) | <*имя_функции выражение*>

рабочее_выражение ::= *пусто* | *терм2 рабочее_выражение*
терм2 ::= *символ-литера* | *символ-число* | *символ-метка* |
(*рабочее_выражение*) | <*имя_функции рабочее_выражение*>

переменная ::= *тип индекс* | *тип спецификация индекс*
тип ::= `s | S | w | W | e | E | v | V`
индекс ::= *буква* | *цифра*
спецификация ::= *спецификатор* | (*набор_ограничений*)

пусто ::=

Приложение 2. Синтаксис языка Рефал-5

программа ::= *определение_функции* |
определение_функции программа |
определение_функции `;` *программа* |
external-директива `;` *программа* |
программа external- директива `;`

определение_функции ::= *имя_функции* `{` *блок* `}` |
`$ENTRY` *имя_функции* `{` *блок* `}`

external- директива ::= `$EXTERNAL` *список_имён_функций* |
`$EXTERN` *список_имён_функций* |
`$EXTRN` *список_имён_функций*
список_имён_функций ::= *имя_функции* |
имя_функции `,` *список_имён_функций*
имя_функции ::= *идентификатор*

блок ::= *предложение* | *предложение* `;` | *предложение* `;` *блок*

предложение ::= *образец последовательность_условий* = *выражение* |
образец последовательность_условий `,` *присоединённый_блок*
последовательность_условий ::=
`,` *условие последовательность_условий* | *пусто*
условие ::= *аргумент* `:` *образец*
аргумент ::= *выражение*

образец ::= *пусто* | *терм_образца образец*
терм_образца ::= *символ-литера* | *макроцифра* | *идентификатор* |
переменная | `(`*образец*`)`

присоединённый_блок ::= *аргумент* `:` `{` *блок* `}`

выражение ::= *пусто* | *терм выражение*
терм ::= *символ-литера* | *макроцифра* | *идентификатор* |
переменная | `(`*выражение*`)` | `<`*имя_функции выражение*`>`

переменная ::= *признак_типа буква* | *признак_типа цифра* |
признак_типа`.`*идентификатор* | *признак_типа*`.`*макроцифра*
указатель_типа ::= `s` | `t` | `e`
символ-литера ::= `'`*символ*`'`
макроцифра ::= *последовательность_цифр*