Язык программирования Рефал Плюс

Рутен Гурин, Сергей Романенко

НОУ Институт программных систем -
Университет города Переславля
Москва, 2006 (Версия от 21 ноября)

Оглавление

Введение

Рефал Плюс представляет собой один из диалектов языка программирования Рефал.

Рефал (алгоритмический язык рекурсивных функций) был создан В.Ф.Турчиным в качестве языка, предназначенного для описания семантики других алгоритмических языков [Тур 1966], [Тур 1971]. Впоследствии, после появления достаточно эффективных реализаций [КРТ 1972], [БзР 1977], [Ром 1987а] Рефал нашел применения в таких областях как компьютерная алгебра, конструирование компиляторов и интерпретаторов, искусственный интеллект и др.

Основным типом данных в Рефале являются *объектные выражения*, которые представляют собой произвольные деревья, изображаемые в линейной записи как последовательности *символов* и *скобок*, правильно построенные относительно скобок. Символы представляют собой элементарные объекты (такие как литеры, слова, числа и ссылки на объекты).

Основным средством для анализа структуры объектных выражений и для доступа к их компонентам является *сопоставление с образцом*. Образцы в Рефале могут содержать символы, скобки и *переменные*. Если объектное выражение имеет структуру, соответствующую образцу, переменные, входящие в образец, получают в качестве значений фрагменты объектного выражения. Значения переменных впоследствии могут использоваться для построения новых объектных выражений.

Рефал-программа представляет собой набор определений *функций*. Каждая функция получает в качестве аргумента некоторое объектное выражение и вырабатывает в качестве результата некоторое объектное выражение. Функции могут произвольным образом вызывать друг друга. Основным средством организации управления в программах является *рекурсия*, т.е. такая организация вызовов функций при которой некоторые функции вызывают сами себя (либо непосредственно, либо через другие функции).

Рефал Плюс возник в результате обобщения опыта, накопленного при разработке, реализации и использовании Базисного Рефала [БзР 1977], Рефала-2 [КлР 1986], [КлР 1987], [Ром 1987а], Рефала-4 [Ром 1987б], [Ром 1987в], Рефала-5 [Тур 1989], и языков FLAC [Кис 1987] и RL [Ром 1987г], [Ром 1988а], [Ром 1988б].

По сравнению с другими диалектами Рефала, Рефал Плюс обладает следующими особенностями.

- Более развитая система модульности.

Каждый модуль разделен на две части: интерфейс модуля и реализацию модуля. Интерфейс модуля содержит ту часть модуля, которая "видна" из других модулей, а реализация модуля содержит те части, которые скрыты от других модулей.

Интерфейс модуля может содержать любые объявления, что дает возможность экспортировать из модуля не только объявления функций, но и объявления констант, каналов ввода-вывода, ящиков и таблиц. При экспорте функций экспортируются не только их имена, но и описания форматов их аргументов и результатов, что дает возможность проверять корректность их вызовов еще на стадии компиляции.

При компиляции модуля используются только интерфейсы других модулей, но не их реализации, что позволяет проводить компиляцию модуля даже в том случае, если еще отсутствуют реализации других модулей. Кроме того, внесение изменений в реализацию модуля не требует перекомпиляции зависящих от него модулей. Таким образом, допускаются любые (в том числе и циклические) связи между модулями.

- Статические объявления динамических объектов.

Все объекты, которые могут порождаются динамически, во время исполнения программы (каналы ввода-вывода, ящики, векторы и таблицы) могут объявляться и статически. В этом случае им присваиваются символические имена, которые используются в дальнейшем в тексте программы.

- Объявления функций

Каждая функция в Рефале Плюс может быть объявлена *откатной* или *безоткатной*. Если функция - откатная, она может выдавать в качестве результата особое значение "неуспех". Например, все функции-предикаты выдают в качестве результата либо пустое выражение, либо "неуспех". Если же функция - безоткатная, она никогда не выдает "неуспех".

Предыдущие диалекты Рефала позволяли определять только безоткатные функции.

Другая особенность Рефала Плюс - возможность определять функции, получающие несколько аргументов и выдающие несколько результатов. Количество и типы аргументов функции мы будем называть ее *арностью* (arity), а количество и типы результатов - ее *коарностью* (coarity).

Арность и коарность функции определяются заданием ее входного и выходного *формата*. Эти форматы представляют собой образцы, которые содержат символы, скобки и переменные. Входной формат налагает синтаксические ограничения на внешний вид вызовов функции, а выходной формат - на контексты, в которых может стоять вызов функции. Если из входного формата удалить символы и скобки, получаем последовательность переменных, описывающих арность функции. Если же то же самое сделать с выходным форматом функции, получаем ее коарность.

Явное объявление форматов функций позволяет обнаруживать многие ошибки еще на стадии компиляции и дает возможность уменьшить накладные расходы при исполнении вызовов функций.

В предыдущих диалектах Рефала считалось, что каждая функция получает ровно один аргумент и выдает ровно один результат.

- Перехват неуспехов и аварий.

Вычисление любой конструкции в Рефале Плюс может завершаться либо *успехом*, либо *неуспехом*, либо *аварией*.

Если вычисление закончилось успехом, результатом является некоторое объектное выражение (которое всегда соответствует некоторому заранее ожидаемому формату результата и в реализации может фактически представляться несколькими выражениями).

Если вычисление закончилось неуспехом, результатом является сигнал неуспеха и, может быть, некоторая дополнительная информация о месте, в котором он возник.

Если вычисление закончилось аварией, результатом является сигнал аварии и сообщение об ошибке - некоторое объектное выражение.

Рефал Плюс содержит конструкции, которые позволяют перехватывать и анализировать неуспехи и аварии.

- Ввод-вывод объектных выражений.

Рефал Плюс предоставляет набор функций, позволяющих выводить и вводить цепочки литер, а также изображения объектных выражений. При этом происходит автоматическое превращение литер слов и чисел в их изображения и обратно.

- Работа с ящиками, векторами и таблицами.

Рефал Плюс дает возможность работать с динамически создаваемыми объектами: ящиками, векторами, строками и таблицами. Средства работы с ящиками те же, что и в Рефале-2, в то время как векторы, строки и таблицы являются особенностью Рефала Плюс.

Каждый ящик представляет собой объект, который может содержать произвольное объектное выражение.

Каждый вектор представляет собой объект, который может содержать произвольную конечную последовательность объектных выражений.

Каждая строка представляет собой объект, который может содержать произвольную конечную последовательность литер.

Доступ к ящикам, векторам и строкам производится через символы-ссылки указывающие на эти объекты. Имеются функции для создания новых ящиков, векторов и строк и для извлечения и изменения их содержимого. Возможно выборочное извлечение и изменение отдельных компонент векторов и строк.

Каждая таблица представляет собой объект, который содержит конечный набор *ключей*, с каждым из которых связано его *значение*. И ключи, и таблицы являются некоторыми объектными выражениями. Доступ к таблице производится через символы-ссылки указывающие на эту таблицу. Имеются функции для создания и копирования таблиц, для получения по ключу его значения и для получения всех ключей, содержащихся в таблице. По существу каждая таблица представляет собой изображение функции с конечной областью определения.

- "Векторное" представление объектных выражений.

Реализация Рефала Плюс основана на новом "векторном" представлении объектных выражений [АбР 1988], позволяющем свести копирование выражения к копированию пары указателей на его концы.

Дешевизна операции копирования позволяет применять функциональный стиль программировании на Рефале, в то время как при использовании предыдущих реализаций приходилось тщательно избегать копирования, что вынуждало, по существу, придерживаться императивного стиля.

Освобождение памяти, занятой уже ненужными выражениями, производится с помощью сборки мусора.

Глава 1

Программирование на Рефале Плюс

В данной главе мы даем неформальный обзор основных изобразительных средств языка Рефал Плюс, приводим примеры программ и обсуждаем некоторые приемы программирования на Рефале Плюс. Полное описание Рефала Плюс содержится в главе II “Синтаксис и семантика Рефала Плюс”, куда и следует обращаться за недостающими тонкостями и подробностями. Если в примерах программ встречаются вызовы неизвестных функций, следует обращаться за их описанием к главе III “Библиотека функций” и к разделу “Алфавитный указатель функций”.

1.1 Первый пример

По сложившейся традиции, начнем с рассмотрения простейшей программы на Рефале Плюс:

```
$use StdIo;              // Импорт функций ввода-вывода
                         // из модуля StdIo
Main                     // Определение главной функции
  = <PrintLn "Hello!">;  // Печать и перевод строки
```

Эта программа состоит из двух директив. Первая директива

```
$use StdIo;
```

говорит о том, что в программе будут использоваться библиотечные функции ввода-вывода, которые должны быть импортированы из модуля `StdIo`. Вторая директива является определением функции `Main`. По принятому в Рефале Плюс соглашению, работа программы всегда начинается с вызова функции `Main`.

Функция `Main` всегда имеет пустой аргумент. В данной программе она вызывает библиотечную функцию `PrintLn` с аргументом `"Hello!"`, в результате чего на стандартное устройство вывода выводится цепочка литер

```
Hello!
```

а вслед за ней - литера "конец строки", после чего работа программы заканчивается.

Литеры `//` обозначают начало комментария, т.е. все, что следует за ними до конца строки - игнорируется.

1.2 Структуры данных

1.2.1 Объектные выражения

Данные, обрабатываемые программами написанными на языке Рефал Плюс, являются так называемыми *объектными выражениями*.

Ниже, в качестве примера, приведены 3 объектных выражения

```
"Петр" "Иванович" 33 "года"
("Маша" 17) ("Клава" 24) ("Эльвира" 6)
("мой" "дом") "имеет" ("большие" ("светлые" "окна"))
```

Первое, что бросается в глаза, это использование *скобок*. Если в этих примерах изменить количество или расположение скобок, то изменится структура и, видимо, подразумеваемый смысл этих выражений.

Кроме скобок вышеприведенные выражения содержат *символы*. Вот примеры символов:

```
"John" "Джон" "джоН" "ку-ку" 1988 -99999999999999
```

В общем случае *объектные выражения* состоят из *символов* и *скобок*. Всякое объектное выражение – это последовательность *объектных термов* (которая может быть пустой). Объектный терм – это либо символ, либо объектное выражение, заключенное в круглые скобки "(" и ")". Таким образом, всякое объектное выражение – это последовательность символов и скобок, в которой скобки "правильно расставлены".

В памяти компьютера выражения хранятся в виде структуризованных объектов, однако при выполнении операций ввода/вывода (печать, запись в файл или чтение из файла и т.п.) они должны быть представлены в виде линейных последовательностей *литер* (печатных знаков).

Рефал-система обеспечивает ввод и вывод объектных выражений и все необходимые при этом преобразования. При вводе выражений, входной поток литер разбивается на *лексемы*. Каждая лексема изображает

символ или скобку. Лексемы отделяются друг от друга пробелами (при этом переход на следующую строку считается эквивалентным пробелу).

Если рядом оказываются две скобки или скобка и символ, пробелы между ними могут опускаться. Во всех остальных случаях лексемы обязательно должны быть разделены хотя бы одним пробелом.

В текстах Рефал-программ могут появляться в виде констант следующие символы: символы-литеры, символы-слова и символы-числа.

Символ-литера – это печатный знак. Последовательность символов-литер изображается в виде последовательности соответствующих литер, заключенной в апострофы.

Примеры литер:

```
'A'   'a'   '*'   '2'   ' '
```

Цепочка из трех литер `'A'`, `'B'` и `'C'` может быть записана, например, следующими способами (а также многими другими):

```
'A' 'B' 'C'
'ABC'
'AB''C'
```

Символ-слово представляет собой цепочку литер и изображается в виде этой цепочки литер, заключенной в двойные кавычки.

Если слово начинается с прописной латинской буквы или подчеркивания, и содержит только латинские буквы, цифры и подчеркивания, то окружающие его двойные кавычки можно опустить.

Примеры слов:

```
"Вася"
"Вот-Слово"
"вот-очень-очень-длинное-Слово"
X_25m3s__
"равно?"
```

Символ-число является целым числом. Изображается в виде непустой последовательности десятичных цифр, перед которой может стоять знак “+” или “-”. Например:

```
237
-99999999999999999999999999999999999999999999999
+13
```

Разрядность целых чисел произвольна.

1.2.2 Представление данных в виде объектных выражений

Объектные выражения особенно удобны для представления символьных (т.е. не чисто числовых) данных.

Пусть, например, мы хотим выбрать представление в виде объектных выражений для алгебраических формул. Нам нужно уметь представлять константы, переменные и формулы, которые получаются в результате применения бинарной операции к двум более простым формулам. Обозначим через $\lceil p \rceil$ объектное выражение, которое является представлением формулы p. Тогда мы можем считать, что представлением целых чисел являются соответствующие символы-числа, представлением переменных являются соответствующие символы-слова, а для бинарных операций имеют место соотношения

$$\begin{array}{lcl} \lceil p+q \rceil & = & (\texttt{"плюс"}\ \lceil p \rceil\ \lceil q \rceil\ ) \\ \lceil p-q \rceil & = & (\texttt{"минус"}\ \lceil p \rceil\ \lceil q \rceil\ ) \\ \lceil p*q \rceil & = & (\texttt{"умн"}\ \lceil p \rceil\ \lceil q \rceil\ ) \\ \lceil p/q \rceil & = & (\texttt{"дел"}\ \lceil p \rceil\ \lceil q \rceil\ ) \\ \lceil p^q \rceil & = & (\texttt{"степ"}\ \lceil p \rceil\ \lceil q \rceil\ ) \end{array}$$

Таким образом, формула

$$(X + Y^2) - 512$$

представляется в виде

```
("минус" ("плюс" X ("степ" Y 2)) 512)
```

В качестве другого примера рассмотрим представление шахматной позиции в виде объектного выражения.

Первым делом нужно обозначить название и цвет каждой фигуры. Например `("бел" "Король")`, `("черн" "пешка")`. Затем нужно указать поле каждой фигуры. Например `("e" 2)`, `("h" 7)`. Теперь мы можем представить всю позицию в виде последовательности объектных термов, каждый из которых содержит имя и цвет фигуры, а также ее поле. Например

```
(("бел" "Король")   ("g" 5))
(("черн" "Король")  ("a" 7))
(("бел" "Пешка")    ("c" 6))
(("бел" "Конь")     ("g" 1))
(("черн" "Конь")    ("a" 8))
```

1.2.3 Объекты и значения

В самом широком смысле под *объектами* обычно понимают некоторые сущности, которые существуют во времени, развиваются, но при этом остаются сами собой.

Хорошим примером объекта может служить человек, который рождается, растет, развивается и умирает, все же оставаясь, в каком-то смысле, тем же самым человеком.

Другой знаменитый пример принадлежит Гераклиту (расцвет творческих сил которого приходится приблизительно на 504-501 гг. до н.э.). Гераклит учил, что нельзя дважды войти в одну и ту же реку, ибо "на входящих в ту же самую реку набегают все новые и новые воды" [Гер 500]. Таким образом, река тоже может служить хорошим примером объекта.

Под *значениями* в широком смысле обычно понимают некоторые сущности, которые не меняются, не развиваются и, в этом смысле, находятся вне времени.

Неизвестно, существуют ли значения в реальной жизни, но они являются излюбленным предметом изучения в математике. Примером значения, например, может служить число 25.

Конечно, значение можно считать частным, вырожденным случаем объекта (а именно, застывшим объектом, не способным к развитию), но мы все же обычно будем называть "объектами" только такие объекты, которые не являются значениями.

Иметь дело с объектами труднее, чем со значениями, ибо они могут изменяться. Поэтому объекты очень часто снабжают *именами*. Основным свойством имени является то, что оно однозначно связано с объектом (однозначно идентифицирует этот объект). В отличие от самих объектов, имена являются типичными значениями, поскольку они не меняются из-за того, что изменяется сам объект. Например, несмотря на то, что состояние реки Волга непрерывно изменяется, это никак не сказывается на слове "Волга". Другим примером (полного) имени могут служить паспортные данные человека: фамилия, имя, отчество, дата и место рождения и т.д.

В рамках языка Рефал термины "объект" и "значение" имеют более специальный смысл.

Любое *значение* в языке Рефал представляет собой некоторое объектное выражение.

Любой *объект* в языке Рефал представляет собой "контейнер", который может содержать объектные выражения и другую информацию.

Объекты создаются во время компиляции или в процессе работы Рефал-программы. При создании объекта одновременно с ним создается *символ-ссылка*, который мы будем называть *именем* объекта или *ссылкой* на этот объект. Основное свойство имени объекта заключается

в том, что оно должно *отличаться от всех других* символов-ссылок, существующих в момент создания объекта. Благодаря этому, между символами-ссылками и объектами существует однозначное соответствие: каждому символу-ссылке соответствует ровно один объект, а равным символам-ссылкам соответствует один и тот же объект.

Наглядно взаимосвязь между именем объекта R, объектом и содержимым объекта C можно изобразить следующим образом:

$R \rightarrow \boxed{\text{C}}$

Рефал Плюс имеет дело с объектами следующих типов.

Объекты-функции содержат скомпилированные определения функций. Они создаются во время компиляции программы.

Все остальные объекты могут создаваться как статически (т.е. во время компиляции программы), так и динамически (т.е. в процессе исполнения программы).

Объекты-ящики предназначены для хранения объектных выражений. Каждый ящик содержит ровно одно объектное выражение.

Объекты-таблицы предназначены для хранения конечных множеств упорядоченных пар. Каждая пара содержит два объектных выражения, первое из которых является ключом, а второе – значением, связанным с этим ключом. Все ключи в таблице должны быть попарно различны. Таким образом, каждому ключу в таблице однозначно соответствует связанное с ним значение.

Объекты-каналы дают возможность выполнять операции ввода-вывода.

Объекты-векторы предназначены для хранения конечных последовательностей объектных выражений.

Объекты-строки предназначены для хранения конечных последовательностей литер.

1.2.4 Сборка мусора

В языке Рефал Плюс имеется возможность порождать объекты во время работы программы, но не предусмотрено никаких специальных средств для явного уничтожения объектов. Таким образом, может случиться так, что в процессе выполнения Рефал-программы память будет заполняться все новыми и новыми объектами, хотя многие из них будут уже и не нужны. Конечно, теоретически это не создает никаких проблем, однако, поскольку Рефал-программы исполняются с помощью реальных компьютеров, память которых конечна, во всех реализациях языка Рефал предусмотрен механизм сборки мусора.

Сборка мусора автоматически запускается каждый раз, когда исчерпается свободная память. При этом обнаруживаются и удаляются все

объекты, которые заведомо не могут повлиять на дальнейшие вычисления, а именно те объекты, до которых невозможно добраться прямо или косвенно исходя из значений переменных, имеющихся в этот момент.

На рисунке 1.1 схематически изображены значения переменных, а также объекты и их содержимое. Белые кружочки изображают некоторые элементы выражений, которые не являются символами-ссылками, черные - символы-ссылки. Для удобства изложения все объекты на рисунке пронумерованы. Символы-ссылки обозначены цифрами.

Видно, что по ссылке из значения одной из переменных можно добраться до объекта 1, и из него – до объекта 4. По ссылке из значения другой переменной можно непосредственно добраться до объекта 2 и косвенно (через объект 2) до объектов 4, 5, 6, 3. Таким образом, нет способа извлечь информацию из объектов 7 и 8. Если в этот момент запустить сборку мусора, то объекты 7 и 8 будут уничтожены. Если теперь убрать ссылку на объект 1 из значения переменной, то станет недоступным и объект 1. Если же ссылку на объект 1 оставить, но убрать ссылку на объект 2 из значений переменных, то окажутся ненужными все объекты, кроме 1 и 4.

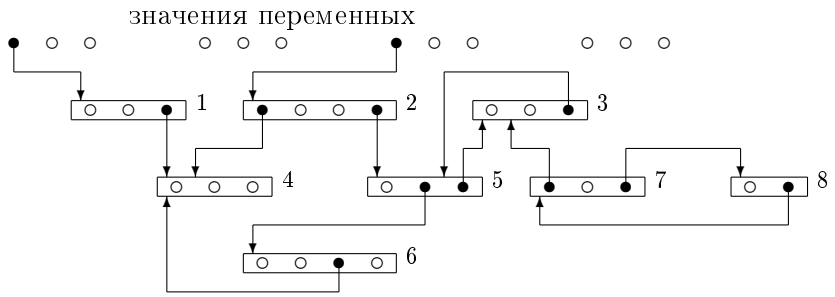


Рис. 1.1: Объекты и ссылки

1.3 Вычисление и анализ объектных выражений

1.3.1 Результатные выражения

Результатные выражения в языке Рефал являются аналогом общеизвестных арифметических выражений. Так, например, арифметическому выражению $X * Y + 3$ в Рефале Плюс соответствует следующее результатное выражение:

```
<Add <Mult sX sY> 3>
```

Угловые скобки обозначают вызов функции. Причем первый символслово после левой угловой скобки обозначает имя функции, а все остальное соответствует аргументам функции. Благодаря тому, что аргумент

функции всегда явно заключается в угловые "функциональные" скобки, отпадает необходимость использовать круглые скобки для группировки операций. Например, выражение $X * (A + B)$ на Рефале записывается как

```
<Mult sX <Add sA sB>>
```

в то время как выражение $X * A + B$ на Рефале записывается в виде

```
<Add <Mult sX sA> sB>
```

Результатные выражения, как и арифметические выражения в других языках, используются для получения одних значений из других. А именно, при вычислении результатного выражения все входящие в него переменные заменяются на их значения, после чего вычисляются вызовы всех функций. При этом, если вызовы функций вложены друг друга, то сначала вычисляются более внутренние вызовы.

Ясно, что результатное выражение можно вычислить только в том случае, если заданы значения для входящих в него переменных. Информацию о значениях переменных мы будем называть *средой*. Запись

$$\{V_1 = \mathcal{E}_1, \ldots, V_n = \mathcal{E}_n\}$$

будет изображать среду, в которой переменные $V_1, \ldots, V_n$ имеют значения $\mathcal{E}_1, \ldots, \mathcal{E}_n$ соответственно.

Ясно, что запись арифметических выражений в виде результатных выражений довольно громоздка, однако, как мы увидим в дальнейшем, у нее есть и определенные преимущества.

Дело в том, что выбор той или другой системы обозначений определяется природой тех объектов, с которыми мы собираемся иметь дело, а также набором операций, которые должны применяться к этим объектам.

Желательно выбирать такие обозначения, чтобы те операции, которые встречаются чаще всего, записывались бы как можно короче. Ясно, что самым кратким обозначением операции является отсутствие всякого обозначения, т.е. пустое место. В случае арифметических выражений мы имеем две основных операции: сложение и умножение, и одну из них можно обозначить пустотой. Так обычно и поступают, опуская знак умножения.

В случае языка Рефал, основным объектом работы являются не числа, а объектные выражения, для которых имеется две основных операции: конкатенация (сцепление) двух выражений и заключение выражения в скобки. Синтаксис Рефала построен таким образом, чтобы эти операции имели кратчайшие обозначения.

А именно, если у нас имеются два результатных выражения Re' и Re'', то запись

$$Re'\ Re''$$

тоже является результатным выражением, которое означает, что надо вычислить Re' и Re'', а полученные результаты сцепить вместе. Т.е. если результатом вычисления Re' и Re'' являются объектные выражения $\mathcal{E}'$ и $\mathcal{E}''$ соответственно, то результатом вычисления Re' Re'' является объектное выражение $\mathcal{E}'$ $\mathcal{E}''$.

Если у нас имеется результатное выражение Re, то запись

(*Re*)

тоже является результатным выражением, которое означает, что надо вычислить Re, а полученный результат заключить в круглые скобки. Т.е. если результатом вычисления Re является объектное выражение $\mathcal{E}$, то результатом вычисления (*Re*) является (*E*).

Например, результатом вычисления результатного выражения

```
sX '+' sY (eZ)
```

в среде `{sX = 25, sY = 36, eZ = A (B C) D}` является объектное выражение

```
25 '+' 36 (A (B C) D)
```

1.3.2 Переменные

Каждая переменная в Рефале Плюс должна начинаться с *признака типа* переменной. Признак типа указывает на множество допустимых значений переменной и должен быть одной из четырех букв: `s`, `t`, `v` или `e`. В соответствии с этим все переменные делятся на четыре категории: s-переменные, t-переменные, v-переменные и e-переменные.

Значениями s-переменных могут быть только символы, значениями t-переменных – только объектные термы, значениями v-переменных – только непустые объектные выражения. Что касается e-переменных, то их значениями могут быть произвольные объектные выражения.

В дальнейшем мы будем говорить, что некоторая переменная является ve-переменной, если она является либо v-переменной, либо e-переменной.

1.3.3 Образцы

В языке Рефал Плюс основным средством анализа объектных выражений являются *образцы*.

Образцы могут содержать символы, круглые скобки и переменные. Например:

```
A B C
tX (eY B)
```

Каждый образец изображает множество объектных выражений, которые могут быть получены из него путем замены переменных на произвольные значения, удовлетворяющие их типам. Например, образец `A eX` изображает множество объектных выражений, которые начинаются с символа `A`, а образец `sX sY` – множество объектных выражений, состоящих ровно из двух символов.

Если одна и та же переменная входит в образец несколько раз, то все ее вхождения должны иметь одно и то же значение. Например, образец `tX tX` изображает множество выражений, состоящих из двух одинаковых термов.

Если нам задано объектное выражение $\mathcal{E}$ и образец P, то всегда можно *сопоставить* $\mathcal{E}$ с P и установить, имеет ли $\mathcal{E}$ структуру, предписанную образцом P. Если да, то мы говорим, что $\mathcal{E}$ *успешно сопоставляется* с P, в противном случае мы говорим, что результатом сопоставления является *неуспех*.

В результате успешного сопоставления $\mathcal{E}$ с P переменные, входящие в P, связываются с соответствующими частями выражения $\mathcal{E}$. Таким образом, результатом сопоставления $\mathcal{E}$ с P является некоторая среда ρ. Например, если мы сопоставим `AAA BBB CCC` с образцом `eX sY`, результатом сопоставления будет среда `{eX = AAA BBB, sY = CCC}`.

Попробуем теперь сопоставить выражение `A B C` с образцом `e1 sX e2`. Мы обнаружим, что это можно сделать тремя различными способами, в результате чего получаются три разных среды:

```
{e1 = ,     sX = A, e2 = B C}
{e1 = A,    sX = B, e2 = C}
{e1 = A B,  sX = C, e2 = }
```

Что считать результатом сопоставления в подобных случаях? В Рефале Плюс эта проблема решается следующим образом. Считается, что правильными являются все варианты сопоставления, однако одни варианты “предшествуют” другим, т.е. имеют приоритет.

А именно, пусть ρ_1 и ρ_2 - два варианта сопоставления $\mathcal{E}$ с P. Рассмотрим все вхождения переменных в P. Если ρ_1 и ρ_2 не совпадают, они приписывают некоторым переменным различные значения. Найдем в P самое первое слева вхождение переменной которому ρ_1 и ρ_2 приписывают разные значения и сравним длину этих значений. Если значение, приписываемое средой ρ_1, короче, чем значение, приписываемое средой ρ_2, то считается, что ρ_1 “предшествует” ρ_2, т.е. имеет приоритет перед ρ_2, в противном случае считается, что ρ_2 “предшествует” ρ_1.

Например, сопоставим объектное выражение `(A1 A2 A3)(B1 B2)` с образцом `e1 (eX sA eY) e2`. В результате получится следующее множество вариантов сопоставления

```
{e1 = , eX = ,       sA = A1, eY = A2 A3, e2 = (B1 B2)}
```

```
{e1 = , eX = A1,    sA = A2, eY = A3,    e2 = (B1 B2)}
{e1 = , eX = A1 A2, sA = A3, eY = ,      e2 = (B1 B2)}
{e1 = (A1 A2 A3), eX = ,   sA = B1, eY = B2, e2 = }
{e1 = (A1 A2 A3), eX = B1, sA = B2, eY = ,   e2 = }
```

где варианты сопоставления перечислены в соответствии с их приоритетами, т.е. самый первый вариант находится на первом месте и т.д.

Описанный выше способ упорядочения вариантов сопоставления называется *сопоставлением слева направо*. Однако в Рефале Плюс имеется возможность упорядочивать варианты сопоставления и *справа налево*, при этом упорядочение происходит не по самому левому вхождению переменной, а по самому правому. Чтобы изменить порядок сопоставления, следует приписать к образцу слева ключевое слово `$r`. Например, если мы сопоставим объектное выражение `(A1 A2 A3)(B1 B2)` с образцом `$r e1 (eX sA eY) e2`, множество вариантов сопоставления будет упорядочено следующим образом:

```
{e1 = (A1 A2 A3), eX = B1, sA = B2, eY = ,   e2 = }
{e1 = (A1 A2 A3), eX = ,   sA = B1, eY = B2, e2 = }
{e1 = , eX = A1 A2, sA = A3, eY = ,      e2 = (B1 B2)}
{e1 = , eX = A1,    sA = A2, eY = A3,    e2 = (B1 B2)}
{e1 = , eX = ,      sA = A1, eY = A2 A3, e2 = (B1 B2)}
```

1.4 Функции определяемые в программе

1.4.1 Форматы функций

С чисто формальной точки зрения все функции в языке Рефал Плюс имеют ровно один аргумент и всегда выдают ровно один результат. Однако, во многих случаях, мы заранее знаем, какую структуру должны иметь аргумент и результат. Например, функция `Add` всегда должна получать на входе объектное выражение, состоящее из двух символов и всегда выдает на выходе объектное выражение, состоящее из одного символа.

Ограничения, накладываемые на структуру аргументов и результатов функций, описываются с помощью объявлений функций. Так, например, объявление функции `Add` имеет вид:

```
$func Add sX sY = sZ;
```

В общем случае объявление функции $\mathcal{F}$ имеет вид

`$func` $\mathcal{F}$ F_{in} `=` F_{out}`;`

где F_{in} входной формат функции, а F_{out} – ее выходной формат. Форматы функции могут содержать символы, круглые скобки и переменные.

Индексы переменных никакой информации не несут, используются в качестве комментариев и могут опускаться.

Входной и выходной форматы должны быть "жесткими", т.е. на одном уровне скобок может находиться не более чем одна ve-переменная. Например, формат `(e)(e)` является жестким, а формат `e A e` – не является.

Все аргументы и результаты функции должны иметь структуру предписанную ее объявлением. Объявление функции должно предшествовать ее первому использованию в каком-либо результатном выражении в программе. При этом, если функция определена в самой программе, ее объявление появляется в тексте программы явно. Если же функция определена в другом модуле, ее объявление должно быть импортировано в программу с помощью директивы `$use`.

Во время компиляции программы производится проверка того, что во всех вызовах функции структура ее аргумента соответствует ее входному формату. Например, результатное выражение

```
<Add 2 <Add sX sY>>
```

построено правильно. По отношению ко внутреннему вызову функции это очевидно, но для того, чтобы проверить корректность наружного вызова, уже требуется привлечь информацию о структуре результата функции `Add`. А именно, заменяем `<Add sX sY>` на выходной формат функции `Add`, в результате чего получается `<Add 2 s>`.

Отсюда видно, что аргумент наружного вызова соответствует входному формату функции `Add`. С другой стороны, если мы напишем в программе результатное выражение

```
<Add 2 <Add sX sY> 3>
```

оно будет расцениваться компилятором как ошибочное, поскольку аргумент наружного вызова функции `Add` состоит из трех символов, а не из двух, как предписано ее входным форматом.

Таким образом, информация о входных и выходных форматах функций позволяет находить многие ошибки в программе еще на стадии компиляции.

1.4.2 Определения функций

Рефал-программа состоит из *определений функций*, каждое из которых имеет следующий вид:

$\mathcal{F}$ `\{` Snt_1`;` Snt_2`;` ... Snt_n`;` `};`

или

$\mathcal{F}$ `{` Snt_1`;` Snt_2`;` ... Snt_n`;` `};`

где $\mathcal{F}$ – имя функции, а Snt_1, Snt_2, ..., Snt_n – *предложения*. (Тонкое различие между "\{" и "{" для нас пока не существенно и будет рассмотрено позднее.)

Каждое предложение Snt_j имеет вид $P_j\ R_j$, где P_j – *входной образец* предложения, а R_j – *хвост* предложения.

Определение функции имеет следующий смысл. Предположим, что требуется вычислить вызов функции $\mathcal{F}$ вида

<$\mathcal{F}$ Re>

Тогда первым делом вычисляется результатное выражение Re. Предположим, что в результате этого получилось объектное выражение $\mathcal{E}$. Тогда делается попытка сопоставить $\mathcal{E}$ с входными образцами P_1, P_2, ..., P_n, пока не найдется такой образец P_j, для которого сопоставление проходит успешно, т.е. имеется хотя бы один вариант сопоставления $\mathcal{E}$ с P. После этого из всех вариантов сопоставления $\mathcal{E}$ с P выбирается "самый первый" вариант ρ, после чего в среде ρ вычисляется хвост R_j. Если при этом получится некоторое объектное выражение $\mathcal{E}'$, оно и считается результатом вызова функции.

Пока что, простоты ради, будем считать, что каждый из хвостов R_j имеет вид

$= Re_j$

где Re_j - результатное выражение. Хвосты такого вида называются *правыми частями*. Вычисление правой части $= Re_j$ сводится к вычислению результатного выражения Re_j, а то, что получится, считается результатом всего хвоста.

В качестве примера, рассмотрим функцию `Sumsq`, вычисляющую сумму квадратов двух чисел. В общепринятых обозначениях ее определение записывается в виде

$$SumSq(X, Y) = X * X + Y * Y$$

в то время как на Рефале ее определение принимает следующий вид:

```
$func SumSq sX sY = sZ;

SumSq
  {
  sX sY = <Add <Mult sX sX> <Mult sY sY>>;
  };
```

Обратите внимание, что объявление функции должно располагаться в программе раньше как ее определения, так и первого вызова, так как информация о функции, которая содержится в ее объявлении необходима для правильной компиляции ее определения и ее вызовов.

Если объявление функции имеет вид

$\texttt{\$func}\ \mathcal{F}\ F_{in}\ \texttt{=}\ F_{out}\texttt{;}$

то компилятор выполняет проверку, что все входные образцы P_1, P_2, ..., P_n являются "уточнениями" входного формата F_{in}, а все хвосты R_1, R_2,..., R_n заведомо вырабатывают результаты, удовлетворяющие выходному формату F_{out}.

1.4.3 Определения функций из одного предложения

Если определение функции содержит только одно предложение Snt, т.е. имеет вид

$\mathcal{F}\ \backslash\{\ Snt\texttt{; \}};$

его разрешается записывать в сокращенном виде следующим образом:

$\mathcal{F}\ Snt\texttt{;}$

Например, определение функции Sumsq может быть записано следующим образом:

```
Sumsq  sX sY = <Add <Mult sX sX> <Mult sY sY>>;
```

1.4.4 Локальные переменные

Определим функцию `SqSub1`, вычитающую из аргумента единицу и возводящую полученное число в квадрат:

$SqSub1(X) = (X - 1) * (X - 1)$

Простейшее определение на Рефале имеет вид:

```
$func SqSub1 sX = sZ;

SqSub1  sX = <Mult <Sub sX 1> <Sub sX 1>>;
```

Очевидный недостаток такого определения в том, что дважды производятся одни и те же вычисления: а именно, дважды вычисляется выражение `<Sub sX 1>`. Этого можно избежать посредством введения вспомогательной функции `Sq`:

```
$func SqSub1 sX = sZ;
$func Sq     sY = sZ;

SqSub1  sX = <Sq <Sub sX 1>>;
Sq      sY = <Mult sY sY>;
```

Единственное назначение функции `Sq` – подождать, пока закончится вычитание единицы, "поймать" полученный результат и затем продолжить вычисления. Ясно, что введение вспомогательных функций, как правило, загромождает программу и делает ее более трудной для восприятия. Поэтому в Рефале Плюс имеется возможность вводить новые переменные для обозначения промежуточных результатов вычислений.

А именно, мы можем записать определение функции `Sq-Sub1` следующим образом:

```
$func SqSub1 sX = sZ;

SqSub1  sX =
  <Sub sX 1> :: sY,
    <Mult sY sY>;
```

где `<Sub sX 1> :: sY` означает, что нужно вычислить `<Sub sX 1>`, и запомнить результат в переменной `sY`. Затем нужно вычислить `<Mult sY sY>`, причем при этом разрешается использовать значение переменной `sY`. А то, что получится, считается результатом всей правой части предложения.

1.4.5 Лирико-синтаксическое отступление: тропы, хвосты и источники

В этом месте нам придется на время отвлечься от темы "локальные переменные" и обсудить некоторые особенности синтаксиса Рефала Плюс. Эти особенности связаны не с сутью Рефала Плюс, а с тем, что при его разработке была сделана попытка добиться максимальной краткости при записи программ. Однако, к сожалению, это привело к усложнению системы понятий, связанных с синтаксисом Рефала Плюс.

По-сути, все конструкции Рефала Плюс, используемые в определениях функций, занимаются либо анализом структуры объектных выражений, либо что-то вычисляют и выдают результат. Для анализа данных служат *образцы*, а для выработки результатов - конструкции, которые называются *тропами*.

Термин *тропа* был использован для того, чтобы подчеркнуть, что выработка результата – это, в общем случае, процесс сложный, который может потребовать выполнения некоторой последовательности шагов: вычисление постепенно, как бы шаг за шагом, продвигается вдоль тропы. А результатное выражение – это простейший пример тропы, когда результат получается "за один шаг".

Конструкция

```
= <Sub sX 1> :: sY,
    <Mult sY sY>;
```

является более сложным примером тропы, когда вычисление результата выполняется в два этапа. Сначала вычисляется промежуточный результат, а потом он используется при вычислении результатного выражения `<Mult sY sY>`.

При этом, запятая `,` не обозначает никаких действий. Ее роль – чисто синтаксическая. Если ее убрать, то переменная `sY` "прилипнет" к следующему за ней результатному выражению

```
= <Sub sX 1> :: sY
    <Mult sY sY>;
```

и станет непонятно, где заканчивается часть тропы, ответственная за присваивание, и где начинается выработка результата.

В связи с этим, в описании Рефала Плюс приходится использовать два дополнительных синтаксических понятия: *хвост* и *источник*. Хвосты и источники – это "хорошие" тропы, обладающие некоторыми полезными синтаксическими свойствами.

Хвосты – это тропы, которые начинаются с какого-то *ключевого слова*, благодаря чему они однозначно отделяются от того, что стоит перед ними. Сразу же следует отметить, что "ключевые слова" в Рефале Плюс могут состоять не только из букв, они могут содержать и другие литеры. Например, тропы

```
= A B C
, A B C
```

являются хвостами. (Тонкая разница между `=` и `,` для нас пока несущественна, поскольку она проявляется только при обработке "неуспехов".)

Источники – это тропы, которые не содержат запятых на верхнем уровне фигурных скобок. Например:

```
A B C
\{ <Sub sX 1> :: sY, <Mult sY sY>; }
```

являются источниками. Слово *источник*, было использовано потому, что в конструкции присваивания "источник" является источником значений для переменных.

В дальнейшем, при описании Рефала Плюс, мы будем обозначать тропы через Q, хвосты – через R, а источники – через S.

Если некоторая тропа Q не является хвостом, её (без изменения смысла) всегда можно превратить в хвост, приписав спереди запятую: `,` Q.

Если некоторая тропа Q не является источником, её (без изменения смысла) всегда можно превратить в источник, заключив в фигурные скобки: `\{` Q`;}`.

1.4.6 Локальные переменные (продолжение)

Теперь мы можем вернуться к вопросу о локальных переменных.

А именно, присваивание локальным переменным делается с помощью тропы вида

$$S :: He\ R$$

где S – источник, R – хвост, а He – так называемое "жесткое выражение". Жесткое выражение состоит из символов, скобок и переменных и должно удовлетворять следующим ограничениям. Во-первых, He не может содержать два вхождения одной и той же переменной, во-вторых, на каждом уровне скобок может находиться не более одной ve-переменной. Легко видеть, что жесткое выражение He является в то же время и форматным выражением, и во время компиляции программы выполняется проверка, что S заведомо вырабатывает результаты, удовлетворяющие формату He.

Тропа $S :: He\ R$ вычисляется следующим образом. Первым делом вычисляется источник S. Если в результате получается объектное выражение $\mathcal{E}$, то переменные из He связываются с соответствующими частями $\mathcal{E}$, после чего вычисляется хвост R, и полученный результат считается результатом всей конструкции.

Обратите внимание, что при вычислении тропы $S :: He\ R$, источник S вычисляется в той среде, в которой находится вся конструкция. После этого среда пополняется значениями переменных из He, после чего хвост R вычисляется уже в новой, исправленной среде. Таким образом, результатом вычисления тропы

```
100 :: sX, <Add sX 1> :: sX = sX
```

является `101`.

В тех случаях, когда жесткое выражение He – пустое, тропа вида $S :: R$ может быть записана в сокращенном виде как $S\ R$.

Такая конструкция называется условием и обычно применяется в тех случаях, когда требуется вычислить S не для получения какого-то объектного выражения, а ради побочных эффектов, возникающих при вычислении S. Например, в результате вычисления тропы

```
<PrintLn "A">, <PrintLn "B">, <PrintLn "C"> =
```

будет напечатано три строчки, первая из которых будет содержать литеру `A`, вторая – литеру `B`, а третья – литеру `C`.

Если же в присваивании $S :: He\ R$ хвост R состоит из одной запятой, его можно опустить, в результате чего получается конструкция $S :: He$.

1.4.7 Рекурсия

Определение функции может содержать как вызовы библиотечных функций, так и вызовы функций, определяемых в программе. В частности, функция может вызывать и саму себя (либо непосредственно, либо через другие определяемые функции). В этом случае будем говорить, что определение функции является *рекурсивным*.

Необходимость в рекурсивных определениях обычно возникает в тех случаях, когда функция должна быть определена для бесконечного множества аргументов и при этом размер аргумента может быть произвольно большим.

Рассмотрим, например, следующую задачу. Пусть требуется определить функцию `Reverse`, которая "переворачивает" выражение, т.е. переставляет термы аргумента в обратном порядке. А именно, если на вход поступает объектное выражение вида

$$\mathcal{T}_1\ \mathcal{T}_2\ \ldots\ \mathcal{T}_n$$

где $\mathcal{T}_1$, $\mathcal{T}_2$, ..., $\mathcal{T}_n$ – некоторые объектные термы, то на выходе должно получиться объектное выражение

$$\mathcal{T}_n\ \ldots\ \mathcal{T}_2\ \mathcal{T}_1$$

Если бы длина входного выражения была ограничена, например, если бы мы знали, что n<=3, можно было бы разобрать четыре разных случая и написать следующее определение функции:

```
$func Reverse e.Exp = e.Exp;

Reverse
  {
  = ;
  t1 = t1;
  t1 t2 = t2 t1;
  t1 t2 t3 = t3 t2 t1;
  };
```

Однако, если длина входного выражения не ограничена сверху, получается, что нужно разобрать в программе бесконечное число случаев, что приводит к программе бесконечно больших размеров.

Можно, однако, преодолеть это затруднение с помощью рекурсии. В данном случае можно рассуждать следующим образом. Рассмотрим входное выражение

$$\mathcal{T}_1\ \mathcal{T}_2\ \ldots\ \mathcal{T}_n$$

Если $n = 0$, то результатом должно быть пустое выражение. Если же $n >= 1$, то можно свести исходную задачу к более простой: отбросить первый терм входного выражения, получив выражение длины $n - 1$,

$$\mathcal{T}_2 \ \ldots \ \mathcal{T}_n$$

и перевернуть это более короткое выражение. Получится

$$\mathcal{T}_n \ \ldots \ \mathcal{T}_2$$

Теперь ясно, что если к этому выражению приписать справа $\mathcal{T}_1$, мы как раз и получим желаемый результат

$$\mathcal{T}_n \ \ldots \ \mathcal{T}_2 \ \mathcal{T}_1$$

Эти рассуждения приводят нас к следующему рекурсивному определению функции `Reverse`

```
Reverse
  {
  = ;
  t.X e.Rest = <Reverse e.Rest> t.X;
  };
```

Интересно отметить, что для задачи переворачивания выражения существует и другое решение, которое ничем не хуже первого: чтобы свести задачу к более простой, можно было бы отбросить не первый терм, а последний. Тогда получилось бы следующее решение

```
Reverse
  {
  = ;
  e.Rest t.X = t.X <Reverse e.Rest>;
  };
```

Нетрудно заметить также, что в действительности сущность решения состоит в том, что мы разбиваем исходное выражение $\mathcal{E}$ на два непустых выражения $\mathcal{E}_1$ и $\mathcal{E}_2$ меньшего размера, таких, что

$$\mathcal{E} = \mathcal{E}_1 \ \mathcal{E}_2$$

Теперь мы можем перевернуть каждое из выражений $\mathcal{E}_1$ и $\mathcal{E}_2$ по-отдельности. Пусть при этом получатся выражения $\mathcal{E}'_1$ и $\mathcal{E}'_2$ соответственно. Тогда ясно, что

$$\mathcal{E}'_2 \ \mathcal{E}'_1$$

равно результату обращения исходного выражения $\mathcal{E}$.

Если Рефал реализован на многопроцессорной машине таким образом, что обращение выражений $\mathcal{E}_1$ и $\mathcal{E}_2$ можно выполнять одновременно, оказывается, что выгодно, чтобы $\mathcal{E}_1$ и $\mathcal{E}_2$ были приблизительно одинаковой длины. Тогда получаем следующий вариант описания функции, в котором используются библиотечные функции из модулей `Access` и `Arithm`:

```
$func Reverse e.Exp = e.Exp;

Reverse
  {
  = ;
  t1 = t1;
  eX,
    <Length e.X> :: sLen,
    <Div sLen 2> :: sDiv,
    = <Reverse <Middle sDiv 0 eX>>
      <Reverse <Left   0 sDiv eX>>;
  };
```

1.5 Неуспехи и аварии

1.5.1 Неуспех при вычислении результатных выражений и троп

До сих пор мы считали, что результатом вычисления тропы Q является некоторое объектное выражение $\mathcal{E}$. В действительности, однако, это вычисление может закончиться *неуспехом* или *аварией*.

В случае неуспеха результатом вычисления является особое значение, "неуспех", которое не является объектным выражением. Простейший способ породить неуспех – это вычислить хвост вида

```
$fail
```

В случае аварии результатом вычисления является особое значение, которое не является объектным выражением и имеет вид `$error(`$\mathcal{E}$`)`, где $\mathcal{E}$ – некоторое объектное выражение, которое является сообщением об ошибке. Обычно $\mathcal{E}$ начинается с символа-слова, являющегося именем функции, во время работы которой возникла авария. Например, если мы попытаемся вычислить вызов функции

```
<Div 10 0>
```

возникнет авария "деление на ноль", а результатом вычисления этого вызова будет

```
$error(Div "Divide by zero")
```

Значения вида `$error(`$\mathcal{E}$`)` обладают следующим свойством. Пусть нам требуется вычислить некоторую конструкцию. Тогда, если при вычислении некоторой составной части этой конструкции получается результат `$error(`$\mathcal{E}$`)`, дальнейшее вычисление конструкции прекращается и результатом вычисления всей конструкции считается `$error(`$\mathcal{E}$`)`. Единственным исключением из этого правила является конструкция `$trap`, специально предназначенная для "перехвата" аварий.

Все тонкости, связанные с обработкой аварий, подробно описаны в главах 2 и 3, поэтому, чтобы не загромождать изложение, в данной главе мы будем их по возможности опускать.

1.5.2 Перестройки

Рассмотрим следующую задачу: пусть нам задано некоторое объектное выражение $\mathcal{E}$, которое заведомо содержит на нулевом уровне не менее двух символов-литер `'+'`. Требуется разбить это выражение $\mathcal{E}$ на три части $\mathcal{E}_x$, $\mathcal{E}_a$ и $\mathcal{E}_y$, такие, что $\mathcal{E} = \mathcal{E}_x$ `'+'` $\mathcal{E}_a$ `'+'` $\mathcal{E}_y$, и при этом $\mathcal{E}_x$ и $\mathcal{E}_y$ не содержат `'+'` на нулевом уровне скобок. Функцию, решающую данную задачу назовем `PlusPlus`. Тогда, например

```
<PlusPlus 'AAA+BBB+CCC+DDD+EEE'> =>
    ('AAA')('BBB+CCC+DDD')('EEE')
```

Таким образом, требуется найти в $\mathcal{E}$ самый левый `'+'` и самый правый `'+'`. Самый левый `'+'` легко найти с помощью сопоставления $\mathcal{E}$ с образцом `$l eX '+' eP`, а самый правый – с помощью сопоставления с образцом `$r eQ '+' eY`. Внутри одного образца сопоставление выполняется либо слева направо, либо справа налево, поэтому мы не можем объединить эти два образца в один, и вынуждены выполнять сопоставление в два приема. Таким образом, мы можем решить задачу следующим образом:

```
$func PlusPlus    eZ       = (eX)(eA)(eY);
$func PlusPlusAux (eX)(eP) = (eX)(eA)(eY);

PlusPlus     $l eX '+' eP = <PlusPlusAux (eX)(eP)>;
PlusPlusAux  $r (eX)(eA '+' eY) = (eX)(eA)(eY);
```

Мы видим, что из-за необходимости выполнять анализ выражения в два этапа, пришлось ввести вспомогательную функцию. Однако, без этого можно было бы обойтись.

Перестройками называются тропы вида

$S : Snt$

где S – источник, а Snt – предложение вида $P\ R$, состоящее из образца P и хвоста R.

Если хвост R состоит из одной запятой, его разрешается опускать, в результате чего перестройка принимает вид $S : P$.

Вычисление перестроек производится следующим образом. Сначала вычисляется S. Если получится неуспех, то результатом всей перестройки считается неуспех. Если же в результате вычисления S получится объектное выражение $\mathcal{E}$, то $\mathcal{E}$ сопоставляется с образцом P. При этом, на рассматриваемые варианты сопоставления накладывается дополнительное ограничение: если в той среде, в которой вычисляется перестройка, значения некоторых переменных уже определены, то рассматриваются только те варианты сопоставления, для которых эти переменные сохраняют прежние значения.

Например, если переменная `sX` уже имеет значение `1`, то в результате сопоставления объектного выражения `1 2 1 2` с образцом `eA sX eB` получатся только два варианта сопоставления

```
{eA = , sX = 1, eB = 2 1 2}
{eA = 1 2, sX = 1, eB = 2}
```

а не четыре.

Пусть теперь ρ_1, ρ_2, ..., ρ_n – все получившиеся варианты сопоставления, перечисленные в соответствии с порядком, введенным ранее для вариантов сопоставления. Далее делается попытка вычислить R, используя значения переменных из первого варианта сопоставления. Если результатом вычисления R является объектное выражение $\mathcal{E}$, оно и считается результатом перестройки. Если же вычисление хвоста R завершилось неуспехом , то этот неуспех "перехватывается", т.е. самый первый вариант сопоставления отбрасывается и делается попытка точно так же вычислить R для оставшихся вариантов сопоставления.

Если для всех вариантов сопоставления результатом вычисления хвоста R является неуспех, то и результатом всей перестройки является неуспех.

Например, вычисление тропы

```
'ABC' : $r e1 sX e2, <Print sX> $fail
```

приведет к тому, что будет напечатана последовательность литер `'CBA'`, после чего вычисление тропы завершится с результатом "неуспех".

Теперь мы можем дать новое определение функции `PlusPlus`, не использующее вспомогательную функцию `PlusPlusAux`:

```
$func PlusPlus  eZ = (eX)(eA)(eY);

PlusPlus
  $l eX '+' eP,
```

```
eP : $r eA '+' eY
  = (eX)(eA)(eY);
```

1.5.3 Перехват неуспехов

В языке Рефал Плюс имеется возможность проверить, закончилось ли вычисление некоторой тропы неуспехом и, в зависимости от этого, предпринять те или иные действия.

Отрицаниями условий называются тропы вида

`#` S R

где S – источник, а R – хвост. Если хвост R состоит из одной запятой, его разрешается опускать, в результате чего отрицание условия принимает вид `#` S.

С синтаксической точки зрения любое отрицание условия является хвостом.

Вычисление таких хвостов производится следующим образом. Первым делом делается попытка вычислить источник S. Если в результате этого получается пустое объектное выражение, результатом всего хвоста является неуспех. Если же результатом вычисления источника S является неуспех, то вычисляется хвост R, и то, что получится, считается результатом всей конструкции.

Распутьями называются тропы вида

`\{` Q_1`;` Q_2`;` $\ldots$ Q_n`; }`

где Q_1, Q_2, $\ldots$, Q_n – некоторые тропы. С синтаксической точки зрения любое распутье является источником.

Вычисление распутья происходит следующим образом. Первым делом делается попытка вычислить тропу Q_1. Если в результате этого получается объектное выражение $\mathcal{E}$, оно считается результатом всего распутья. Если же результатом вычисления тропы Q_1 является неуспех, то делается попытка вычислить `\{` $Q_2; \ldots, Q_n;$ `}`, и то, что получится, является результатом всего распутья. Таким образом, распутье "перехватывает" неуспехи.

Если результатом вычисления всех троп является неуспех, то и результатом всего распутья является неуспех.

Рассмотрим, например, тропу

```
1 :: sX,
 \{ sX : 0 = 1; sX : 1 = 0; }
```

Она вычисляется следующим образом. Первым делом выполняется определение локальной переменной `1 :: sX`, в результате чего `sX` получает значение `1`. После этого делается попытка вычислить первую

тропу распутья, т.е. `sX : 0 = 1`. При попытке выполнить сопоставление `sX : 0` выясняется, что сопоставление проходит неуспешно, поэтому результатом этой тропы является неуспех. Вследствие этого делается попытка вычислить следующую тропу `sX : 1 = 0`, в результате чего получается результат `0`, который и является результатом вычисления всей тропы.

В некоторых случаях, однако, полезным оказывается другой вариант распутья, имеющий следующий вид:

`{` Q_1`;` Q_2`;` ... Q_n`;` `}`

Его отличие от предыдущего варианта проявляется в том, что если результатом вычисления всех троп является неуспех, то результатом вычисления распутья является ошибка `$error(`$\mathcal{F}$ `"Unexpected fail")`, где $\mathcal{F}$ – имя функции, в которой находится распутье.

Таким образом, мы можем считать, что пара скобок `\{ ... }` "прозрачна" для неуспехов, в то время как пара скобок `{ ... }` для них "непрозрачна", ибо неуспех не может "выскочить" из распутья `{` Q_1`;` Q_2`;` ... Q_n`;` `}`.

Довольно часто встречаются тропы следующего вида:

S`:`Ve`, \{`Ve `:` Snt_1`;` Ve `:` Snt_2`;` ... Ve `:` Snt_n`;}`
S`:`Ve`, {`Ve `:` Snt_1`;` Ve `:` Snt_2`;` ... Ve `:` Snt_n`;}`

где Ve – некоторая e-переменная, которая больше нигде в определении функции не используется, а каждое из предложений Snt_j имеет вид P_j R_j. Для таких троп предусмотрена, соответственно, следующая сокращенная форма записи:

S `: \{`Snt_1`;` Snt_2`;` ... Snt_n`;}`
S `: {`Snt_1`;` Snt_2`;` ... Snt_n`;}`

называемая выбором, при этом конструкции `\{`Snt_1`;` Snt_2`;` ... Snt_n`;}` и `{`Snt_1`;` Snt_2`;` ... Snt_n`;}` называются *образцовыми распутьями*.

Хвосты R_j, состоящие из одной запятой, разрешается опускать, в результате чего соответствующие предложения P_j R_j принимают вид P_j.

Особенно следует отметить, что если образцовое распутье используется в определении функции на самом верхнем уровне, то хвосты, состоящие из одной запятой тоже могут быть опущены. Например, функция

```
A_B \{ A = ; B = ; };
```

возвращает пустое выражение, если на вход подать `A` или `B`, а в противном случае вырабатывает неуспех. Она эквивалентна функции

```
A_B \{ A , ; B , ; };
```

которая эквивалентна функции

```
A_B \{ A; B; };
```

Таким образом, распутье `{ sX : 0 = 1; sX : 1 = 0; }` более кратко записывается в виде выбора `sX : { 0 = 1; 1 = 0; }`, а распутье `\{ sX : A,; sX : B,; }` может быть сокращено до `sX : \{ A; B; }`.

С синтаксической точки зрения, все выборы являются источниками, что позволяет записывать, например, присваивания следующего вида:

```
sX : { 0 = 1; 1 = 0; } :: sY = <Add sX sY>
```

При этом сначала вычисляется источник `sX : {0 = 1; 1 = 0;}`, затем полученный результат присваивается переменной `sY`, после чего вычисляется хвост `= <Add sX sY>`.

1.5.4 Управление перехватом неуспехов

Как мы видели выше, Рефал Плюс предоставляет довольно богатый набор конструкций для перехвата неуспехов. Однако, иногда возникает желание породить неуспех такой силы, чтобы он смог преодолеть все ловушки, расставленные для его перехвата (или хотя бы часть из них). Для этого служат *заборы* и *отсечения*.

Заборами называются хвосты вида `\?` Q, а *отсечениями* – хвосты вида `\!` Q.

Заборы и отсечения являются пометками в программе, позволяющими управлять распространением неуспехов. А именно, всякое отсечение `\!` Q должно находиться внутри некоторого забора `\?` $\ldots$ `\!` Q $\ldots$. Вычисление отсечения `\!` Q производится следующим образом. Первым делом вычисляется тропа Q. Если это вычисление завершается с некоторым результатом X, этот результат считается результатом всей конструкции `\?`$\ldots$ `\!` Q $\ldots$. В частности, если X – неуспех, то неуспехом заканчивается и вычисление всей конструкции `\?` $\ldots$ `\!` Q $\ldots$.

Рассмотрим пример, иллюстрирующий использование заборов и отсечений. Предположим, что мы пытаемся вычислить следующую тропу:

```
eA : e1 '+' e2 '-' e3
  = (e1)(e2)(e3)
```

Если значение переменной `eA` содержит `'+'` а вслед за ним содержит `'-'` на нулевом уровне скобок, то в результате сопоставления с образцом будут найдены самый левый `'+'` и ближайший за ним `'-'`, после чего вычисление тропы успешно завершается. Теперь рассмотрим случай, когда `eA` содержит `'+'` на нулевом уровне, но не содержит на нулевом уровне ни одного `'-'`. Тогда первым делом будет найден первый `'+'`, а затем будет просмотрена вся оставшаяся часть выражения, чтобы найти `'-'`.

Поскольку это сделать не удастся, будет выполнено удлинение значения переменной `e1`, после чего снова будет выполнен поиск `'-'` в оставшейся части выражения. Между тем, этого можно было бы и не делать, ибо если `'-'` не нашелся первый раз, он заведомо не найдется и во второй раз.

Мы можем избежать такого перебора с помощью `\?` и `\!`. Для этого, первым делом разобьем исходную перестройку на две части:

```
eA : e1 '+' eX,
  eX : e2 '-' e3
    = (e1)(e2)(e3)
```

Теперь, если сопоставление значения `eX` с образцом `e2 '-' e3` завершается неуспешно, делается попытка испробовать следующий вариант сопоставления для `eA : e1 '+' eX`. Этого, однако, можно избежать добавив `\?` и `\!` следующим образом:

```
\? eA : e1 '+' eX
  \! eX : e2 '-' e3
    = (e1)(e2)(e3)
```

теперь, если внутренняя перестройка выдает неуспех, этот неуспех сразу же становится результатом всей тропы.

1.5.5 Смысл правых частей

Правые части, имеющие вид `=` Q , где Q – некоторая тропа, являются еще более мощным средством ограничения перебора, чем заборы и отсечения.

В общих чертах различие между ключевыми словами `,` и `=` заключается в следующем. Ключевое слово `=` предписывает, что надо окончательно зафиксировать значения переменных, имеющиеся к данному моменту и приступать к вычислению тропы Q. Если результатом вычисления Q будет неуспех, то надо считать, что неуспехом закончилось и вычисление всей конструкции, объемлющей `=` Q. При этом не следует пытаться подбирать для переменных какие-то другие значения.

А в случае хвоста `,` Q , если результатом вычисления Q является неуспех, это необязательно приводит к неуспеху всей конструкции, поскольку делается попытка подобрать для переменных другие значения.

Например, тропа

```
A B C : e sX e, sX : C , sX
```

вырабатывает значение `C`, поскольку для переменной `sX` в конце-концов подбирается "правильное" значение `C`. В то же время, тропа

```
A B C : e sX e = sX : C , sX
```

вырабатывает неуспех, поскольку переменная sX принимает значение A и после прохождения через = это значение фиксируется. Затем терпит неудачу перестройка sX : C, но попытки подбора нового значения для sX уже не происходит.

Такого интуитивного понимания вполне достаточно для практического написания программ. Поэтому большинство читателей может смело пропустить остаток данного раздела. Ибо сомнения возникают только при рассмотрении достаточно сложных троп, наподобие следующей:

```
A B : e sX e,
  sX : { A = B; B = A; } :: sY,
    sY : A, Ok
```

Здесь в начале тропы стоит перестройка A B : e sX e, которая сначала присваивает переменной sX значение A. Затем делается попытка вычислить то, что находится справа от перестройки. А именно, значение переменной sX подается на вход образцового распутья { A = B; B = A; }, вырабатывающего значение B, если на него подано A, и значение A, если на него подано B. Затем значение, выданное образцовым распутьем, присваивается переменной sY. Дальше идет перестройка, которая сопоставляет значение sY с символом A. Поскольку первоначально переменной sX присваивается значение A, переменной sY присваивается значение B, и перестройка sY : A терпит неудачу.

Возникает интересный вопрос: что происходит дальше? Ведь при вычислении значения образцового распутья { A = B; B = A; } вычисление проходит через ключевое слово =, которое "фиксирует значения переменных, имеющиеся к данному моменту". Означает ли это, что когда перестройка sY : A терпит неудачу, перестройка A B : e sX e уже не должна пытаться подобрать новое значение? Согласно подходу, принятому в Рефале Плюс, в данном случае = не отменяет перебор для перестройки A B : e sX e. Поэтому, для sX подбирается новое значение B, которое подается на образцовое распутье, и переменная sY принимает новое значение B. После этого перестройка sY : A срабатывает успешно, и, в качестве результата всей тропы, выдается значение Ok.

В пользу такого подхода имеется следующее соображение. Образцовое { A = B; B = A; } фактически представляет собой замаскированный вызов безымянной функции. Если определить эту функцию в явном виде (назвав, например, AB)

```
$func AB s = s;
AB { A = B; B = A; };
```

всю тропу можно переписать в виде

```
A B : e sX e,
  <AB sX> :: sY,
    sY : A, Ok
```

А если бы считалось, что область действия = не имеет ограничений, то не было бы возможности легко выносить части троп в отдельные функции, или, наоборот, заменять вызовы функций на образцовые распутья из определений этих функций.

Для того, чтобы можно было описать, какие ограничения накладываются на области действия ключевого слова =, требуется ввести несколько дополнительных понятий.

Пусть какая-то конструкция входит в качестве составной части в некоторую другую конструкцию. Мы будем говорить, что эта внутренняя конструкция является *поставщиком*, если результат ее работы, согласно семантике языка, считается результатом работы объемлющей конструкции. Например, в тропе S :: He R хвост R является поставщиком, ибо результат вычисления R считается результатом вычисления всей конструкции.

Если некоторая конструкция не является поставщиком для объемлющей конструкции, мы будем говорить, что она является *ассистентом*. Например, в тропе S :: He R источник S является ассистентом, ибо его результат не считается результатом всей конструкции (хотя и может косвенно использоваться в R через значения переменных).

Теперь рассмотрим различные конструкции, и для каждой из них укажем в явном виде, какие их части являются поставщиками, а какие – ассистентами.

Если определение функции имеет вид $\mathcal{F}$ $Palt$, то образцовое распутье $Palt$ является, по определению, ассистентом. Это выглядит странно, поскольку $Palt$ вырабатывает значение функции. Но здесь – случай особый, поскольку речь идет о самом верхнем уровне в определении функции.

Если тропа имеет один из следующих видов: S R , S :: He R , S : P R или #S R, то источник S является ассистентом.

Если источник имеет вид S : $Palt$, то источник S является ассистентом.

Рассмотрим теперь некоторую конструкцию вместе со всеми объемлющими ее ассистентами. Среди этих ассистентов всегда существует наименьший, и мы будем называть его *хозяином* рассматриваемой конструкции. В частности, всякий ассистент является своим собственным хозяином.

Теперь мы можем описать семантику правых частей.

Пусть хозяином правой части =Q является некоторая объемлющая конструкция ... =Q ... . Тогда первым делом вычисляется тропа Q. Если это вычисление завершилось с некоторым результатом X, то X считается результатом всего хозяина ... =Q ... .

В частности, если X – неуспех, то и значением хозяина ... =Q ... будет неуспех, даже если внутри него расставлены ловушки, предназначенные для перехвата неуспехов.

Например, результатом вычисления тропы

```
\{ A B C : $l e sX e, sX : B, sX} :: sY, sY
```

будет символ `B`, ибо `sX` принимает значение `A`, сопоставление которого с символом `B` терпит неуспех, в результате чего `sX` принимает новое значение `B`, и вычисление успешно завершается. Но если мы заменим запятую на равенство, мы получим тропу

```
\{ A B C : $l e sX e = sX : B, sX} :: sY, sY
```

вычисление которой терпит неудачу.

На использование заборов, отсечений, и правых частей накладываются некоторые ограничения. И отсечение `\!`Q , и соответствующий ему забор `\?`$\ldots$`\!`$Q\ldots$, должны иметь общего хозяина. На пути от `\!` к соответствующему ему `\?` не должно быть ни одного `=` .

1.5.6 Откатные и безоткатные функции

Все функции, определяемые и используемые в программе на Рефале Плюс делятся на две категории: *откатные* и *безоткатные*.

Если функция $\mathcal{F}$ – безоткатная, то вычисление вызова `<`$\mathcal{F}$ Re`>` не может закончиться получением неуспеха. Если же функция $\mathcal{F}$ – откатная, то вычисление вызова `<`$\mathcal{F}$ Re`>` вообще говоря может закончиться получением неуспеха в качестве результата.

До сих пор мы считали, что объявления функций имеют вид

`$func` $\mathcal{F}$ F_{in} `=` F_{out}`;`

однако это верно только в том случае, если функция $\mathcal{F}$ – безоткатная. Если же $\mathcal{F}$ – откатная, ее объявление должно иметь вид

`$func?` $\mathcal{F}$ F_{in} `=` F_{out}`;`

Теперь мы можем более точно описать семантику определений функций. Пусть определение функции $\mathcal{F}$ имеет вид

$\mathcal{F}$ $Palt$

где $Palt$ – образцовое распутье вида `\{`P_1 R_1`;` P_2 R_2`;`$\ldots$$P_n$ R_n`;}` или `{`P_1 R_1`;` P_2 R_2`;`$\ldots$$P_n$ R_n`;}`, и пусть требуется вычислить ее вызов `<`$\mathcal{F}$ Re`>`. Тогда прежде всего вычисляется результатное выражение Re. Если при этом получается неуспех, то вызов функции $\mathcal{F}$ не выполняется и считается, что результатом вычисления `<`$\mathcal{F}$ Re`>` является неуспех. Предположим теперь, что в результате вычисления Re получилось объектное выражение $\mathcal{E}$. Тогда выполняется вызов функции, а именно, вычисляется источник

$\mathcal{E}$ `:` $Palt$

в пустой среде, т.е. в среде, в которой ни одна переменная не имеет значения. Пусть в результате этого получилось значение X. Если X – объектное выражение, то X считается результатом вызова <$\mathcal{F}$ *Re*>. Если же X является неуспехом, то дальнейшие действия зависят от того, является ли функция $\mathcal{F}$ откатной.

Если $\mathcal{F}$ – откатная, и X является неуспехом, то результатом вызова <$\mathcal{F}$ *Re*> считается неуспех.

Если $\mathcal{F}$ – безоткатная, и X является неуспехом, то этот неуспех "перехватывается" и преобразуется в ошибку `$error(`$\mathcal{F}$ `"Unexpected fail")`.

1.6 Логические условия

1.6.1 Проверки и предикаты

В некоторых случаях в процессе работы программы требуется проверить справедливость некоторого условия и, в зависимости от результатов проверки, направить вычисления либо по одному, либо по другому пути. При программировании на Рефале Плюс мы будем придерживаться следующего соглашения. Тропа Q будет называться *условием*, если результатом ее вычисления является либо пустое объектное выражение, либо неуспех. Если результат – пустое выражение, мы считаем, что условие выполнено, а если результат – неуспех, мы считаем, что условие не выполнено. Таким образом, пустой результат означает, что результат проверки – "истина", а неуспех означает, что результат проверки – "ложь".

Конечно, не следует забывать, что вычисление тропы Q может вообще не закончиться или же привести к возникновению ситуации ошибки, но мы будем считать, что это свидетельствует либо о том, что программа неправильна, либо о том, что программе были заданы недопустимые исходные данные.

Некоторые функции, входящие в библиотеку функций, используются только для проверки условий. Такие функции называются *предикатами*. В Рефале Плюс функции-предикаты выдают пустое выражение, если их аргументы удовлетворяют условию и выдают неуспех в случае, если их аргументы не удовлетворяют условию. Например, функция `Lt` проверяет, что ее первый аргумент меньше второго. А именно, результатом вычисления вызова вида <`Lt` ($\mathcal{E}_1$)($\mathcal{E}_2$)> является пустое выражение, если объектное выражение $\mathcal{E}_1$ "меньше", чем объектное выражение $\mathcal{E}_2$. Если же это не так – результатом является неуспех.

Если в программе определяется некоторая функция-предикат, ее объявление должно иметь следующий вид:

`$func?` $\mathcal{F}$ F_{in}`=` `;`

Дальше мы рассмотрим различные способы использования и комбинирования условий.

1.6.2 Ветвления

Пусть у нас имеется условие, заданное источником S и две тропы Q' и Q''. Рассмотрим тропу

```
\? {S \! Q'; \! Q'';}
```

Тогда, если результат вычисления S – пустое выражение, то вычисляется тропа Q', и то, что получится, является результатом всей конструкции. Если же результат вычисления источника S – неуспех, то вычисляется тропа Q'' и то, что получится, является результатом всей конструкции.

Следует обратить внимание на использование отсечений `\!`. Они существенны в том случае, если вычисление Q' или Q'' заканчивается неуспехом. Предположим, что они убраны, в результате чего тропа приобретает следующий вид:

```
{S, Q'; Q'';}
```

Тогда, если условие S выполнено, производится вычисление тропы Q'. Предположим, однако, что вычисление тропы Q' заканчивается неуспехом. Тогда этот неуспех не будет выдан в качестве результата всей конструкции, а будет перехвачен, что приведет к вычислению тропы Q''. В данном случае это совсем не то, что нам было нужно. Таким образом, первое отсечение необходимо, чтобы предотвратить "перескок" на следующую тропу в распутье.

Теперь рассмотрим случай, когда условие S не выполнено, т.е. результатом вычисления S является неуспех. В этом случае неуспех перехватывается и начинается вычисление тропы Q''. Предположим, что вычисление тропы Q'' завершается неуспехом. В этом случае неуспех перехватывается и делается попытка вычислить следующую тропу в распутье. Но, поскольку следующей тропы нет, возникает ситуация ошибки, а это опять таки не то, что мы хотели!

Если ветвление имеет вид

```
\? {S \! = Q'; \! =Q'';}
```

его можно записать более кратко следующим образом:

```
{ S = Q'; = Q'';}
```

(что мы и будем часто делать в дальнейшем).

Рассмотрим следующий пример. Определим функцию `MinE`, которая сравнивает два объектных выражения $\mathcal{E}_1$ и $\mathcal{E}_2$ и выдает $\mathcal{E}_1$, если $\mathcal{E}_1$ предшествует $\mathcal{E}_2$. Если же $\mathcal{E}_1$ равно $\mathcal{E}_2$ или $\mathcal{E}_2$ предшествует $\mathcal{E}_1$, то `MinE` выдает $\mathcal{E}_2$.

```
$func MinE (eX)(eY) = e.MinXY;

MinE  (eX)(eY) =
  {
  <Lt (eX)(eY)>
    = eX;
    = eY;
  };
```

Теперь рассмотрим случай, когда условие задано тропой Q и нужно вычислить тропу Q', если условие выполнено, либо тропу Q'', если условие не выполнено. Этого можно достичь, заключив Q в фигурные скобки и превратив его в источник `\{ `Q`; }`. После этого ветвление может быть записано следующим образом.

`\? { \{`Q`;} \! `Q'`; \! `Q''`;}`

1.6.3 Логические связки

В некоторых случаях приходится вычислять сложные логические условия. При этом удобно выражать сложные условия через более простые с помощью логических связок “и”, “или” и “не”. Хотя Рефал Плюс и не содержит логических связок в явном виде, они легко изображаются с помощью имеющихся в нем конструкций.

- Логическое “И”

Пусть у нас имеются два условия и требуется проверить, что они оба выполнены.

Если оба условия заданны тропами: Q' и Q'' достаточно вычислить тропу

`\{ `Q'`;}, `Q''

Если первое условие задано источником S, второе - тропой Q, достаточно вычислить тропу S`, `Q.

И наконец, если оба условия заданы результатными выражениями Re' и Re'', достаточно вычислить результатное выражение $Re'\ Re''$.

- Логическое “ИЛИ”

Пусть у нас имеются два условия и требуется проверить, что хотя бы одно из них выполнено.

Если оба условия заданны тропами: Q' и Q'' достаточно вычислить тропу

`\{ `Q'`; `Q''`; }`

- Логическое “НЕ”

Пусть у нас имеется условие заданное тропой Q и требуется проверить, что оно не выполнено. Этого можно достичь, вычислив тропу

```
# \{Q;}
```

которая является сокращенной записью для тропы `# \{Q;},` . Если же условие задано источником S, достаточно вычислить тропу

```
# S
```

которая является сокращенной записью для тропы `# S ,` .

И в том, и в другом случае мы пользуемся возможностью опускать хвосты, состоящие из одной запятой.

1.6.4 Пример: программа дифференцирования

Опишем функцию, которая будет выполнять символьное дифференцирование формул [Хен 1983]. Чтобы не загромождать изложение мы будем рассматривать только простые формулы, в которые входят целые числа, переменные, а также операции $+$ и $*$, хотя обобщить рассмотрение на случай более сложных формул не составило бы большого труда.

Будем обозначать через x и y произвольные переменные, через i – произвольное целое число, через e – произвольную формулу, а через $Dx(e)$ - результат дифференцирования e по x. Тогда правила дифференцирования можно записать в виде

$$\begin{array}{lcl} Dx(x) & = & 1 \\ Dx(y) & = & 0 \quad \texttt{(где y не равно x)} \\ Dx(i) & = & 0 \\ Dx(e1 + e2) & = & Dx(e1) + Dx(e2) \\ Dx(e1 * e2) & = & e1 * Dx(e2) + e2 * Dx(e1) \end{array}$$

Прежде чем писать программу дифференцирования, нужно выбрать для формул представление в виде объектных выражений. Обозначим через $\lceil e \rceil$ представление формулы e. Выберем следующее представление

$$\begin{array}{lcl} \lceil x \rceil & = & x \\ \lceil i \rceil & = & i \\ \lceil e_1 + e_2 \rceil & = & (Sum\ \lceil e_1 \rceil\ \lceil e_2 \rceil) \\ \lceil e_1 * e_2 \rceil & = & (Prod\ \lceil e_1 \rceil\ \lceil e_2 \rceil) \end{array}$$

Теперь не составляет труда определить функцию `Diff`, первым аргументом которой является переменная, по которой надо дифференцировать, а вторым аргументом – формула, подлежащая дифференцированию. Результатом работы является продифференцированная формула.

```
$func Diff sX tE = tE;

Diff  sX tE =
  tE :
  {
  sX = 1;
  sY = 0;
  (s.Oper t.E1 t.E2) =
    <Diff sX tE1> :: t.DxE1,
    <Diff sX tE2> :: t.DxE2,
    s.Oper :
    {
    Sum   = (Sum t.DxE1 t.DxE2);
    Prod  = (Sum (Prod t.E1 t.DxE2) (Prod t.E2 t.DxE1));
    };
  };
```

Это определение функции `Diff` страдает очевидным недостатком: формулы, которые получаются в результате дифференцирования, содержат слишком много частей, которые можно сократить. Например

$$DX(3 * (X * X) + 5) = (3 * ((X * 1) + (X * 1)) + (X * X) * 0) + 0$$

в соответствии с приведенными выше правилами дифференцирования. Между тем, после очевидных упрощений эту формулу можно привести к виду

$$3 * (X + X)$$

Для этого достаточно применить к формуле следующие правила преобразования:

$$\begin{array}{lll} 0 + e2 & ==> & e2 \\ e1 + 0 & ==> & e1 \\ 0 * e2 & ==> & 0 \\ e1 * 0 & ==> & 0 \\ 1 * e2 & ==> & e2 \\ e1 * 1 & ==> & e1 \end{array}$$

(Более сложные преобразования мы не рассматриваем, чтобы не загромождать изложения.)

Существует два способа реализовать эти упрощения. Можно сначала построить результат дифференцирования полностью, а затем анализировать его, пытаясь сделать упрощения. Другой способ – делать упрощения сразу же, в процессе дифференцирования. Мы поступим именно так.

Определим функции `Sum` и `Prod`. Эти функции будут получать на входе две формулы, и выдавать формулу, которая является их суммой или произведением соответственно. При этом будут сразу же делаться упрощающие преобразования.

```
$func Sum  t1 t2 = t;
$func Prod t1 t2 = t;

Sum
  {
  0  t2 = t2;
  t1 0  = t1;
  t1 t2 = (Sum t1 t2);
  };

Prod
  {
  0  t2 = 0;
  1  t2 = t2;
  t1 0  = 0;
  t1 1  = t1;
  t1 t2 = (Prod t1 t2);
  };
```

Теперь переделаем определение функции `Diff`, вставив в соответствующие места вызовы функций `Sum` и `Prod`.

```
Diff  sX tE =
  tE :
  {
  sX = 1;
  sY = 0;
  (s.Oper t.E1 t.E2) =
    <Diff sX tE1> :: t.DxE1,
    <Diff sX tE2> :: t.DxE2,
    s.Oper :
    {
    Sum   = <Sum t.DxE1 t.DxE2>;
    Prod  = <Sum <Prod t.E1 t.DxE2> <Prod t.E2 t.DxE1>>;
    };
  };
```

1.6.5 Пример: сравнение множеств

В качестве примера использования логических связок и рекурсии рассмотрим следующую задачу.

Как известно из теории множеств, два множества считаются равными, если они состоят из одних и тех же элементов. Предположим, что мы хотим запрограммировать на Рефале операцию сравнения конечных множеств на равенство. Для этого прежде всего следует придумать какой-то способ изображения множеств в виде объектных выражений. Сначала рассмотрим множества, элементами которых могут быть только символы (в смысле Рефала). Тогда множество символов $\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_n$ можно изобразить в виде объектного выражения

$$\mathcal{S}_1\ \mathcal{S}_2\ \ldots \mathcal{S}_n$$

При этом оказывается, что одному и тому же множеству могут соответствовать несколько его изображений. Например, множество `{John, Mary}` можно изобразить как в виде выражения `John Mary`, так и в виде выражения `Mary John` , или даже в виде `Mary John John Mary`. Таким образом, неравные выражения могут изображать равные множества.

Как известно, элементы множеств сами могут быть множествами. Поэтому мы теперь несколько усложним нашу задачу. Предположим, что рассматриваемые нами множества могут содержать в качестве элементов как символы, так и множества (которые, в свою очередь снова могут содержать множества и т.д.). Теперь мы должны как-то изобразить те элементы множества которые сами являются множествами.

Мы поступим следующим образом. Если элемент множества – символ $\mathcal{S}$, мы будем его изображать в виде символа $\mathcal{S}$. Если же элемент множества – множество X, мы будем изображать его в виде терма вида `(` $\lceil X \rceil$ `)`, где $\lceil X \rceil$ – изображение множества X. Таким образом, множество `{`A, `{`A, B`}, {`A`}}` можно изобразить, например, в виде выражения A `(`AB`) (`A`)` .

Теперь постараемся определить функцию-предикат `IsEqSet` , которая будет определять, являются ли два ее аргумента представлениями одного и того же множества. Для этого мы постараемся разложить понятие равенства множеств на более простые понятия.

А именно, два множества A и B равны в том и только том случае, если A является подмножеством B и B является подмножеством A. Далее, множество A является подмножеством множества B в том и только том случае, если для любого элемента X множества A верно, что X принадлежит B. В формализованном виде это можно записать следующим образом:

$$\begin{aligned} A = B &\Leftrightarrow A \subseteq B \wedge B \subseteq A \\ A \subseteq B &\Leftrightarrow (\forall x \in A)(x \in B) \end{aligned}$$

Поэтому мы определим не одну, а четыре функции-предиката: `IsEqSet` будет проверять, что ее аргументы представляют одно и то же

множество, `IsSubset` будет проверять, что множество, представляемое первым аргументом, является подмножеством множества, представляемого вторым аргументом, `IsEl` будет проверять, что ее первый аргумент представляет элемент, входящий в множество, представляемое вторым аргументом, а `IsEqEl` будет проверять, что ее аргументы представляют один и тот же элемент множества.

Интересно отметить, что если два элемента множества сами являются множествами, то их сравнение приводит к необходимости сравнивать два множества, поэтому функция `IsEqEl` вынуждена вызвать функцию `IsEqSet` . Таким образом, оказывается, что функция `IsEqSet` в конечном итоге рекурсивно определяется через себя.

```
$func? IsEqSet  (eA)(eB) = ;
$func? IsSubset (eA)(eB) = ;
$func? IsEl     tX  (eA) = ;
$func? IsEqEl   tX  tY   = ;

IsEqSet  (eA)(eB) =
  <IsSubset (eA)(eB)><IsSubset (eB)(eA)>;

IsSubset  (eA)(eB) =
  eA :
  {
  = ;
  tX eR = <IsEl tX (eB)><IsSubset (eR)(eB)>;
  };

IsEl  tX (eA) =
  eA : tY eR,
  \{ <IsEqEl tX tY>; <IsEl tX (eR)>; };

IsEqEl  tX tY =
  \{
  tX tY : s s
    = tX : tY;
  tX tY : (eA)(eB)
    = <IsEqSet (eA)(eB)>;
  };
```

1.7 Использование селекторов прямого доступа

Одним из типичных случаев, когда требуется прямой доступ к частям выражения, являются алгоритмы, основанные на методе “разделяй и властвуй”. Этот метод заключается в том, что исходная задача разби-

вается на подзадачи меньшего размера и решается для этих подзадач. Затем решение исходной задачи получается исходя из решений подзадач. Как правило, принцип "разделяй и властвуй" применяется совместно с принципом балансировки, гласящим, что исходную задачу следует разбивать на подзадачи примерно равного размера [АХУ 1979].

В качестве классического примера рассмотрим задачу сортировки (т.е. упорядочения в порядке неубывания) множества целых чисел.

Одним из способов решения этой задачи является сортировка слиянием [АХУ 1979], когда исходное множество чисел S разбивается на два непересекающихся множества S_1 и S_2 примерно равного размера. Затем S_1 и S_2 упорядочиваются независимо друг от друга. В результате получаются две упорядоченные последовательности чисел Q_1 и Q_2, которые соединяются (сливаются) в одну упорядоченную последовательность Q, которая и является решением исходной задачи.

Теперь определим функцию `MSort`, которая получает на входе последовательность целых чисел, разбивает ее на две примерно равные части, сортирует их (обратившись к себе рекурсивно), а затем сливает получившиеся последовательности, обратившись к функции `Merge`.

```
$func MSort eS = eS;
$func Merge (eX)(eY) = eZ;

MSort  eS =
  <Length eS> :: sLen,
  {
  <Le (sLen) (1)>
    = eS;
    = <Div sLen 2> :: sK,
      <Left 0 sK eS> :: eS1,
      <Middle sK 0 eS> :: eS2,
        <Merge ( <MSort eS1> )( <MSort eS2> )>;
  };
```

Теперь определим функцию `Merge`, которая получает на входе две упорядоченные последовательности и объединяет их в одну упорядоченную последовательность.

```
Merge  (eX)(eY) =
  {
  eX :
    = eY;
  eY :
    = eX;
  (eX)(eY) : (sA eXRest)(sB eYRest)
    = {
```

```
      <Le (sA)(sB)>
        = sA <Merge (eXRest)(eY)>;
        = sB <Merge (eX)(eYRest)>;
    };
  };
```

1.8 Функции выдающие несколько результатов

1.8.1 Сквозной просмотр выражений

Рассмотрим несколько примеров, демонстрирующих полезность функций выдающих несколько результатов.

Предположим, что требуется определить функцию `Nmb`, заменяющую все символы выражения на их порядковые номера. Например

```
<Nmb A (B A) C A>   =>   1 (2 3) 4 5
```

Основная трудность здесь заключается в том, что встретив пару скобок, функция не знает заранее, сколько символов находится внутри скобок, а эта информация нужна, чтобы продолжить обработку "хвоста" выражения, стоящего после скобок. Поэтому функция, нумерующая символы должна иметь два аргумента: выражение, подлежащее обработке, и номер, который следует приписать первому символу, который встретится. Эта же функция должна выдавать два результата: обработанное выражение и первый "неиспользованный" номер символа. Таким образом, мы приходим к следующему определению функции `Nmb` (использующему две вспомогательные функции `NmbExp` и `NmbTerm`):

```
$func Nmb      e.Exp    = e.Exp;
$func NmbExp  e.Exp sN = e.Exp sN;
$func NmbTerm t.Exp sN = t.Exp sN;

Nmb  e.Exp =
  <NmbExp e.Exp 1> :: e.Exp s,
    e.Exp;

NmbExp  e.Exp sN =
  e.Exp :
  {
  = sN;
  tX e.Rest =
    <NmbTerm tX sN> :: tX sN,
    <NmbExp e.Rest sN> :: e.Rest sN,
      tX e.Rest sN;
  };
```

```
NmbTerm  tX sN =
  tX :
  {
  s =
    sN <Add sN 1>;
  (eE) =
    <NmbExp eE sN> :: eE sN,
      (eE) sN;
  };
```

1.8.2 Быстрая сортировка

Следующий пример - так называемая *быстрая сортировка* [АХУ 1979], сущность которой заключается в следующем.

Пусть требуется отсортировать множество целых чисел S. Выберем из S произвольное число X и разобьем S на три подмножества S_1, S_2 и S_3, такие, что S_1 содержит числа, которые меньше X, S_2 содержит числа, равные X, а S_3 содержит числа, которые больше X. Отсортируем S_1, S_2 и S_3, пусть при этом получились упорядоченные последовательности Q_1, Q_2 и Q_3 (сортировка множества S_2 – тривиальна, ибо все его элементы равны X). Тогда мы можем соединить Q_1, Q_2 и Q_3 в новую последовательность $Q_1Q_2Q_3$, которая и будет решением исходной задачи.

Определим функцию `QSort`, которая будет сортировать поданную на вход последовательность описанным выше методом. Для разбиения исходного множества чисел на три части используется функция `Split`.

```
$func QSort  eS = eQ;
$func Split  sX eS = (eS1)(eS2)(eS3);
$func SplitAux  sX (eS1)(eS2)(eS3) eS = (eS1)(eS2)(eS3);

QSort  eS =
  {
  eS :
    = ;
  eS : t
    = eS;
  eS : sX e
    = <Split sX eS> :: (eS1)(eS2)(eS3),
        <QSort eS1> eS2 <QSort eS3>;
  };

Split  sX eS =
```

```
    <SplitAux sX ()()() eS>;

SplitAux  sX (eS1)(eS2)(eS3) eS =
  eS :
  {
  =
    (eS1)(eS2)(eS3);
  sY eRest =
    {
    <Lt (sY)(sX)>
      = <SplitAux sX (eS1 sY)(eS2)(eS3) eRest>;
    <Gt (sY)(sX)>
      = <SplitAux sX (eS1)(eS2)(eS3 sY) eRest>;
      = <SplitAux sX (eS1)(eS2 sY)(eS3) eRest>;
    };
  };
```

1.9 Итеративные циклы

Основным средством изображения циклов в Рефале Плюс являются рекурсивные определения функций. Во многих случаях, однако, это средство оказывается слишком универсальным. Поэтому в Рефале Плюс предусмотрена конструкция *поиска*, которая имеет следующий вид:

S'' `$iter` S' `::` $He\ R$

и синтаксически является тропой. При этом, источники S'' и S' являются суверенами, а хвост R – вассалом (что существенно при наличии правых частей вида `=` Q в S'', S' или R).

Если хвост R состоит из одной запятой, его разрешается опускать. Также, если жесткое выражение He – пустое, его разрешается опускать вместе с ключевым словом "`::`".

Тропа-поиск вводит новые локальные переменные (так же, как и тропа-присваивание S `::` $He\ R$). Начальные значения этих переменных получаются путем вычисления источника S''. Затем делается попытка вычислить хвост R. Если она завершается успешно, то полученный результат считается результатом всей конструкции. Если же вычисление хвоста R кончается неуспехом, то вычисляются новые значения для переменных, входящих в He (для этого вычисляется источник S', причем при этом используются старые значения переменных). Далее делается новая попытка вычислить хвост R и т.д.

Таким образом, тропа-поиск как бы пытается подобрать для переменных из He такие значения, при которых вычисление хвоста R успешно завершится.

Точный смысл конструкции поиска проще всего можно определить через смысл более простых конструкций: присваивания и распутья. А именно, поиск S'' `$iter` S' `::` $He\,R$ эквивалентен тропе

S'' `::` He, `\{` R; S' `$iter` S' `::` $He\,R$; `}`

Эта тропа опять содержит конструкцию поиска, которую можно "развернуть" с помощью точно такого же преобразования. Получаем

```
S'' :: He, \{ R;
    S' :: He, \{ R;
        S' $iter S' :: He R;
};}
```

повторив этот процесс бесконечное число раз, получаем, что исходная конструкция поиска эквивалентна бесконечной конструкции

```
S'' :: He, \{ R;
    S' :: He, \{ R;
        S' :: He, \{ R;
...
...};};}
```

Рассмотрим пример использования конструкции поиска – определение функции факториал:

```
$func Fact sN = sFact;

Fact
  {
  0 = 1;
  sN = <Mult sN <Fact <Sub sN 1>>>;
  };
```

Недостаток этого рекурсивного определения в том, что происходит накопление вызовов функции `Mult`, ожидающих пока рекурсия развернется до конца. Можно дать "более итеративное" определение функции `Fact` (при этом потребуется вспомогательная функция `FactAux`).

```
$func Fact sN = sFact;
$func FactAux sR sK = sFact;

Fact  sN =
  <FactAux 1 sN>;

FactAux  sR sK =
  {
```

```
  sK : 0
    = sR;
    = <FactAux <Mult sR sK> <Sub sK 1>>;
  };
```

То же самое изображается с помощью конструкции поиска следующим образом.

```
$func Fact sN = sFact;

Fact  sN =
  1 sN
    $iter <Mult sR sK> <Sub sK 1>
      :: sR sK,
  sK : 0,
    = sR;
```

1.10 Перебор с возвратом

1.10.1 Задача о ферзях

Рассмотрим классическую задачу о восьми ферзях [Хен 1983]. Как известно, в шахматах ферзи атакуют друг друга по горизонтали, по вертикали или по диагоналям шахматной доски. Задача состоит в том, чтобы найти такое размещение 8 ферзей на доске 8×8, чтобы никакие два ферзя не атаковали друг друга.

Мы будем решать несколько более общую задачу, считая, что требуется разместить n ферзей на доске размера $n \times n$.

Занумеруем вертикали и горизонтали доски числами от 1 до n. Будем говорить, что поле имеет координаты (i, j) или, другими словами, является полем (i, j), если оно находится на пересечении i-й вертикали и j-й горизонтали.

Заметим, что для полей любой диагонали, идущей слева направо и снизу вверх, сумма номеров столбцов и строк – постоянна, и что для полей любой диагонали, идущей слева направо и сверху вниз, разность номеров столбцов и строк – постоянна (рис. 1.2).

Таким образом, поля (I, J) и (I', J') лежат на одной диагонали, если $I + J = I' + J'$ или $I - J = I' - J'$. Это условие легко проверить. А именно, если вычисление тропы

```
\{
<Add sI sJ> :: sN1, <Add sI1 sJ1> :: sN2, sN1 : sN2;
<Sub sI sJ> :: sN1, <Sub sI1 sJ1> :: sN2, sN1 : sN2;
}
```

	1	2	3	4	5
1	2	3	4	5	6
2	3	4	5	6	7
3	4	5	6	7	8
4	5	6	7	8	9
5	6	7	8	9	10

строка + столбец

	1	2	3	4	5
1	0	-1	-2	-3	-4
2	1	0	-1	-2	-3
3	2	1	0	-1	-2
4	3	2	1	0	-1
5	4	3	2	1	0

строка − столбец

Рис. 1.2: Номера диагоналей

завершается успешно, поля (I, J) и (I', J') лежат на одной диагонали.

Теперь надо выбрать способ, которым мы будем кодировать расположение ферзей на доске.

Достаточно рассматривать только такие позиции, для которых на одной вертикали находится не более одного ферзя, ибо любые два ферзя, стоящие на одной вертикали атакуют друг друга, и такая позиция не может быть решением. С другой стороны, количество ферзей, которых следует расставить, равно количеству вертикалей, из чего следует что на каждой вертикали должен стоять ровно один ферзь. Поэтому мы будем изображать позицию в виде последовательности чисел

$$I_1\ I_2\ \ldots\ I_n$$

где число I_k означает, что на k-й вертикали ферзь находится на горизонтали с номером I_k. Например, позиция, изображенная на рис. 1.3

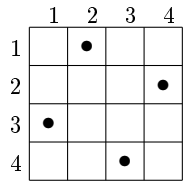


Рис. 1.3: Пример решения

кодируется выражением

```
3 1 4 2
```

Будем строить позицию постепенно, заполняя вертикали одну за другой. При этом всякий раз, пытаясь поставить на доску очередного ферзя, надо проверять, не атакован ли он одним из уже стоящих на доске ферзей. Предположим, что на доске уже стоит k ферзей на вертикалях $1, 2, \ldots, k$. Будем кодировать частично построенную позицию как последовательность чисел

$$I_1\ I_2\ \ldots\ I_k$$

где I_m – номер горизонтали для ферзя на m-й вертикали.

Теперь мы можем определить предикат `UnderAttack`, который выдает пустое выражение, если поле (I, J) атаковано ферзями, уже стоящими на доске, и выдает неуспех, если это поле не атаковано.

```
$func? UnderAttack     sI sJ ePos = ;

UnderAttack  sI sJ ePos =
  ePos : $r eRest e, eRest : e sJ1,
  <Length eRest> :: sI1,
  \{
  sI1 : sI;
  sJ1 : sJ;
  <Add sI sJ> :: sN1, <Add sI1 sJ1> :: sN2, sN1 : sN2;
  <Sub sI sJ> :: sN1, <Sub sI1 sJ1> :: sN2, sN1 : sN2;
  };
```

Отметим, что проверку $I' = I$ можно и убрать, поскольку в нашей программе функция `UnderAttack` будет вызываться только таким образом, что параметр I будет заведомо больше, чем номера столбцов у тех ферзей, которые уже поставлены на доску.

Теперь опишем функцию `NextQueen`, которая пытается добавить к частично построенной позиции нового ферзя, перебирая различные горизонтали. Если ферзя удается поставить, но этот ферзь не последний, делается попытка поставить следующего ферзя и т.д. Если очередного ферзя поставить не удается, происходит возврат назад и делается попытка переставить на новое место предыдущего ферзя.

```
$func? NextQueen sI sN ePos = ePos;

NextQueen  sI sN ePos =
  1 $iter \{ <Lt (sJ) (sN)> = <Add sJ 1>; }
    :: sJ,
  # <UnderAttack sI sJ ePos>,
  ePos sJ :: ePos,
  \? {
  sI : sN
    \!  ePos;
    \!  <NextQueen <Add sI 1> sN ePos>;
  };
```

Некоторые тонкости в определении функции `NextQueen` заслуживают пояснения.

Во-первых, конструкция поиска делает попытку вычислить свое тело, последовательно пробуя для переменной J значения 1, 2, ..., n и

увеличивая значение переменной J на 1 после каждой неудачной попытки вычислить свое тело.

Во-вторых, неудача при вычислении тела конструкции поиска может возникнуть по двум причинам: либо поле (I, J) атаковано ферзями, уже стоящими на доске, и тогда успешно завершится вызов функции `UnderAttack`, а значит потерпит неудачу отрицание этого условия, либо очередного ферзя можно поставить на поле (I, J), но не удается разместить следующих ферзей, и тогда терпит неудачу рекурсивный вызов функции `NextQueen`.

Теперь осталось определить последнюю функцию `Solution`, которая имеет один аргумент – размер доски, и выдает решение задачи, если оно существует, либо терпит неудачу, если решение не существует:

```
$func? Solution   sN = ePos;

Solution  sN =
  <NextQueen 1 sN >;
```

1.10.2 Задача о цепочках:

Рассмотрим следующую задачу [Вир 1977]. Требуется найти объектное выражение $\mathcal{E}$, обладающее следующими свойствами.

- $\mathcal{E}$ не содержит скобок, а всякий входящий в него символ является одним из символов 1, 2 или 3.
- $\mathcal{E}$ имеет заданную длину `len`.
- Не существует таких объектных выражений $\mathcal{E}_a$, $\mathcal{E}_b$ и $\mathcal{E}_c$, что $\mathcal{E}_c$ – не пусто и при этом

  $$\mathcal{E} = \mathcal{E}_a\ \mathcal{E}_c\ \mathcal{E}_c\ \mathcal{E}_b$$

  т.е. $\mathcal{E}$ не содержит двух смежных непустых совпадающих подвыражения.

Для построения требуемого выражения можно поступить следующим образом: начать с пустого выражения, а затем постепенно добавлять к нему числа, следя за тем, чтобы не получалось запрещенной комбинации. При этом, чтобы гарантировать, что цепочка не имеет вида $\mathcal{E}_a\ \mathcal{E}_c\ \mathcal{E}_c\ \mathcal{E}_b$, где $\mathcal{E}_c$ – непустое, достаточно после добавления каждого числа проверять, что не образовалось выражение вида

$$\mathcal{E}_a\ \mathcal{E}_c\ \mathcal{E}_c$$

Для распознавания выражений такого вида определим предикат `IsUnacceptable`.

```
$func? IsUnacceptable e.String = ;

IsUnacceptable  e.String =
  <Div <Length e.String> 2> :: s.Max,
  {
  s.Max : 0
    = $fail;
    = 1
      $iter \{ <Lt (sK) (s.Max)> = <Add sK 1>; }
      :: sK,
      <Right 0 sK <Middle 0 sK e.String>> :: eU,
      <Right 0 sK e.String> :: eV,
      eU : eV;
  };
```

Теперь определим функцию `Extend`, которая пытается добавить к выражению очередной символ, пока цепочка не достигнет заданной длины. Если цепочку продолжить невозможно, происходит возврат назад, и делается попытка изменить предыдущие символы.

```
$func? Extend       s.Len e.String = e.String;

Extend  s.Len e.String =
  {
  <Length e.String> : s.Len
    = e.String;
    = 1 $iter \{ <Lt (s.Digit) (3)> = <Add s.Digit 1>; }
        :: s.Digit,
      e.String s.Digit :: e.String,
      # <IsUnacceptable e.String>,
        <Extend s.Len e.String>;
  };
```

Теперь осталось определить начальную функцию `FindString`, которая получает в качестве параметра желаемую длину цепочки, и выдает искомую цепочку (если она существует), либо терпит неудачу (если она не существует).

```
$func? FindString  s.Len = e.String;

FindString  s.Len =
  <Extend s.Len >;
```

1.11 Пример: компилятор для простого императивного языка

В этом разделе мы рассматриваем методы написания компиляторов на Рефале Плюс. Эти методы демонстрируются на примере небольшого компилятора с простого императивного языка, позаимствованного из [Уор 1980].

Хотя этот компилятор носит чисто учебный характер, он все же значительно больше всех остальных программ, рассмотренных в этой книге, и состоит из нескольких модулей.

1.11.1 Входной язык компилятора

Программа на входном языке представляет собой конечную последовательность *лексем*. Каждая из лексем изображается с помощью цепочки литер, синтаксис которой может быть описан посредством следующей грамматики (см. главу II, раздел 1):

```
$    Лексема =
$         КлючевоеСлово | Идентификатор | Число.
$    КлючевоеСлово =
$         ";" | "(" | ")" | "+" | "-" | "*" | "-" |
$         ":=" | "<=" | '<>' | "<" | ">=" | ">" | "=".
$         "DO" | "ELSE" | "IF" | "READ" | "THEN" |
$         "WHILE" | "WRITE".
$    Идентификатор = Буква { Буква | Цифра }.
$    Число = Цифра { Цифра }.
```

Ключевые слова являются *зарезервированными*, т.е. не разрешается использовать идентификаторы, совпадающие по начертанию с ключевыми словами.

Прописные и строчные буквы в ключевых словах не различаются, т.е. считаются полностью эквивалентными.

Соседние лексемы разделяются между собой "межевыми" литерами: пробелами, табуляциями и литерами конца строки. В тех случаях, когда нет опасности "склеивания" соседних лексем в одну, разделять их не обязательно.

Не всякая последовательность лексем является синтаксически правильной программой. Поэтому, после того как входная последовательность литер разбита на лексемы, следует проверить, что полученная последовательность лексем имеет следующий синтаксис:

```
$    Программа = ПослОператоров.
$    ПослОператоров = Оператор { ";" Оператор }.
$    Оператор =
```

```
$          "IF" Условие "THEN" Оператор "ELSE" Оператор |
$          "WHILE" Условие "DO" Оператор |
$          "READ" ИмяПеременной |
$          "WRITE" Выражение |
$          "(" ПослОператоров ")".
$          ИмяПеременной ":=" Выражение |
$          Пусто.
$     Пусто = .
$     Условие = Выражение ОперОтношения Выражение.
$     ОперОтношения = "=" | "<=" | "<>" | "<" | ">=" | ">".
$     Выражение = Слагаемое { АддитивнОпер Слагаемое}.
$     Слагаемое = Множитель { МультиплОпер Множитель}.
$     Множитель = ИмяПеременной | Значение | "(" Выражение ")".
$     АддитивнОпер = "+" | "-".
$     МультиплОпер = "*" | "/".
$     ИмяПеременной = Идентификатор.
$     Значение = Целое.
```

Программа представляет собой последовательность операторов, которые выполняются слева направо. Каждый из операторов может использовать значения переменных и изменять значения переменных.

Условный оператор

`IF` $Cond$ `THEN` St_1 `ELSE` St_2

проверяет условие $Cond$. Затем, если оно выполнено, то выполняется оператор St_1, иначе – оператор St_2.

Оператор цикла

`WHILE` $Cond$ `DO` St

проверяет условие $Cond$. Если условие выполнено, то выполняется оператор St, а затем все повторяется сначала. Если же условие не выполнено, то выполнение оператора цикла завершается.

Оператор ввода

`READ` Var

читает из устройства ввода одно число и присваивает его переменной Var.

Оператор вывода

`WRITE` $Expr$

вычисляет арифметическое выражение $Expr$ и полученное число записывает на устройство вывода.

Составной оператор

(St_1; St_2; ... St_n)

означает, что следует последовательно выполнить операторы St_1, St_2, ..., St_n.

Оператор присваивания

Var := $Expr$

вычисляет выражение $Expr$ и полученное число присваивает переменной Var.

Пустой оператор не выполняет никаких действий.

Условия и арифметические выражения имеют общепринятый смысл. Операции умножения и деления имеют приоритет перед операциями сложения и вычитания.

Переменные, которые используются в программе, описывать не требуется. Перед началом работы программы их значения не определены.

Ниже приведен пример программы, которая вводит одно число, а затем вычисляет и выводит его факториал.

```
read value;
count:=1;
result:=1;
while count<value do
  (
  count:=count+1;
  result:=result*count
  );
write result
```

1.11.2 Выходной язык

Результатом работы компилятора является целевая программа в "машинном коде", имеющая следующий синтаксис:

```
$    Программа = { Директива }.
$    Директива =
$         Инструкция | "BLOCK" "," Значение ";".
$    Инструкция =
$         КодОперации "," Значение ";" |
$    КодОперации =
$         ADD  | SUB  |  MULT  | DIV  | LOAD  | STORE |
$         ADDC | SUBC |  MULTC | DIVC | LOADC |
$         JUMPEQ | JUMPNE | JUMPLT | JUMPGT | JUMPLE | JUMPGE
$         JUMP | READ | WRITE | HALT.
$    Значение = Число.
```

Программа представляет собой последовательность директив, а каждая директива является либо “инструкцией”, т.е. машинной командой, либо директивой резервирования памяти.

Мы считаем, что *оперативная память* машины состоит из *ячеек*, каждая из которых имеет *адрес* – целое положительное число (таким образом, ячейки пронумерованы с 1). Каждая ячейка может содержать либо одну инструкцию, либо одно целое число.

Выполнение программы всегда начинается с первой ячейки.

Помимо оперативной памяти в машине имеется *сумматор*, который может содержать одно число.

Директива

`BLOCK,`Int`;`

означает, что в этом месте программы следует оставить незанятыми Int ячеек памяти. Обычно эта директива находится в конце программы и используется для выделения ячеек памяти, предназначенных для хранения значений переменных.

Каждая инструкция имеет вид:

Op`,`$Value$`;`

где Op – название выполняемой операции, а $Value$ – *операнд* этой операции. Смысл операнда $Value$ определяется названием операции. Для некоторых операций $Value$ является адресом некоторой ячейки, которая используется при выполнении операции. Для других операций $Value$ представляет собой целое число, которое также используется при выполнении операции. Для некоторых операций, однако, операнд не требуется, и в этом случае $Value$ должно быть равно нулю.

Инструкция `LOAD,`$Addr$`;` загружает в сумматор содержимое ячейки с адресом $Addr$. Инструкция `STORE,`$Addr$`;` записывает содержимое сумматора в ячейку с адресом $Addr$.

Инструкция `LOADC,`Int`;` загружает число Int в сумматор.

Инструкции `ADD`, `SUB`, `MULT` и `DIV` имеют вид Op`,`$Addr$`;` и вычисляют соответственно сумму, разность, произведение и целую часть частного двух чисел. При этом первое число берется из сумматора, второе – из ячейки с адресом $Addr$, а результат записывается в сумматор.

Инструкции `ADDC`, `SUBC`, `MULTC` и `DIVC` имеют вид Op`,`Int`;` и вычисляют соответственно сумму, разность, произведение и целую часть частного двух чисел. При этом первое число берется из сумматора, второе число равно Int, а результат записывается в сумматор.

Инструкция `READ,`$Addr$`;` читает с устройства ввода очередное число и помещает его в ячейку с адресом $Addr$. Инструкция `WRITE,0;` записывает число, находящееся на сумматоре, на устройство вывода.

Инструкция `HALT,0;` прекращает работу программы. Инструкция `JUMP,`$Addr$`;` передает управление на ячейку с адресом $Addr$.

И наконец, последняя группа инструкций – это условные переходы `JUMPEQ`, `JUMPNE`, `JUMPLT`, `JUMPGT`, `JUMPLE` и `JUMPGE`, которые имеют вид *Op, Addr*;. Выполняются они следующим образом. Сначала содержимое сумматора сравнивается с нулем. Если проверяемое условие выполнено, то управление передается инструкции с адресом *Addr*, если не выполнено – следующей инструкции. Какую проверку выполнять определяют две последние буквы в имени инструкции. `EQ` означает проверку того, что содержимое сумматора равно 0, `NE` – что оно не равно 0, `LT` – что оно меньше 0, `GT` – что оно больше 0, `LE` – что оно меньше или равно 0, `GE` – что оно больше или равно 0.

Программа вычисления факториала, приведенная выше, будет переведена компилятором в следующую программу в машинном коде:

```
001  READ,21;        008  JUMPGE,16;     015  JUMP,6;
002  LOADC,1;        009  LOAD,19;        016  LOAD,20;
003  STORE,19;       010  ADDC,1;         017  WRITE,0;
004  LOADC,1;        011  STORE,19;       018  HALT,0;
005  STORE,20;       012  LOAD,20;        019  BLOCK,3;
006  LOAD,19;        013  MULT,19;
007  SUB,21;         014  STORE,20;
```

Слева от директив выписаны соответствующие им адреса.

1.11.3 Общая структура компилятора

Наш компилятор имеет “классическую” структуру и состоит из следующих частей.

Входной поток литер, который часто называют также *конкретной* программой, поступает в *сканер*, который разбивает его на лексемы.

Затем поток лексем попадает в *синтаксический анализатор*, который превращает этот поток в дерево синтаксического разбора, которое часто называют также программой, записанной в *абстрактном синтаксисе*, или *абстрактной* программой.

Далее абстрактная программа попадает в *генератор кода*, результатом работы которого является программа на языке *ассемблера*. Ассемблерная программа, по сути, является уже почти готовой программой в машинном коде, однако, вместо конкретных адресов ячеек она содержит *метки*, каждая из которых изображает некоторый еще неизвестный адрес.

Полученная ассемблерная программа затем поступает в *ассемблер*, который заменяет все метки на конкретные адреса, в результате чего получается окончательная программа в машинном коде.

Информация о соответствии между именами переменных и метками хранится в *словаре* переменных. Таким образом, в компиляторе име-

ется модуль, который обеспечивает работу со словарем. Этот модуль используется при генерации кода и при ассемблировании программы.

Может сложиться впечатление, что данный компилятор имеет неоправданно сложную структуру, не соответствующую простоте входного языка. И действительно, его вполне можно было бы упростить за счет объединения многих компонент. Например, можно было бы делать сканирование программы, синтаксический анализ и генерацию кода одновременно. Однако, не следует забывать, что в случае более сложного входного языка такой подход мог бы привести к запутанному и ненадежному компилятору, в котором было бы очень трудно разобраться. Наш же компилятор служит только в качестве учебного примера по программированию на Рефале Плюс и его назначение – продемонстрировать как Рефал Плюс позволяет применить классические методы построения компиляторов. Взяв этот пример за основу, читатель может попробовать развить его в двух направлениях. Во-первых, в сторону более реалистического и сложного входного языка, во-вторых – в сторону упрощения компилятора за счет устранения из него чрезмерной "научности" и общности подхода.

1.11.4 Модули компилятора и их интерфейсы

Компилятор состоит из следующих модулей:

```
Cmp     —  головная программа
CmpScn  —  сканер
CmpPrs  —  синтаксический анализатор
CmpGen  —  генератор кода и ассемблер
CmpDic  —  работа со словарем
```

Головной модуль не имеет интерфейсной части и содержит определение начальной функции `Main`. Прочие модули состоят из двух частей: интерфейса модуля и реализации модуля.

Модуль `CmpScn` имеет следующую интерфейсную часть:

```
//
// File CmpScn.rfi
//

$func  InitScanner  s.Channel = ;
$func  ReadToken    = s.TokenClass s.TokenInfo;
$func  TermScanner  = ;
```

Из него экспортируются три функции.

Функция `InitScanner` служит для инициализации сканера. Параметр `s.Channel` является ссылкой на канал, из которого сканер начинает

читать литеры. Прежде чем обращаться к `InitScanner`, следует открыть этот канал на чтение.

Функция `TermScanner` должна быть вызвана после того как вся исходная программа будет прочитана. Это дает возможность сканеру закончить свою деятельность и приготовиться к чтению другой программы.

Функция `ReadToken` выдает очередную лексему исходной программы в виде двух символов: первый символ указывает, к какому классу принадлежит лексема, а второй символ содержит дополнительную информацию о лексеме.

Модуль `CmpPrs` имеет следующую интерфейсную часть:

```
//
// File: Cmp.rfi
//

$func Parse  s.Channel = t.Program;
```

Из него экспортируется одна функция `Parse`, которая читает (обращаясь к сканеру) исходную программу из канала `s.Channel` и выдает абстрактную программу `t.Program`. Прежде чем обращаться к `Parse`, следует открыть канал `s.Channel` на чтение.

Если исходная программа содержит синтаксические ошибки, функция `Parse` выдает `$error(`$\mathcal{E}$`)`, где $\mathcal{E}$ – сообщение об ошибке.

Модуль `CmpGen` имеет следующую интерфейсную часть:

```
//
// File: CmpGen.rfi
//

$func GenCode    t.Program = t.Code;
$func WriteCode  t.Code = ;
```

Из него экспортируются две функции.

Функция `GenCode` получает абстрактную программу `t.Program` и выдает скомпилированную программу в машинном коде `t.Code`, во внутреннем представлении в виде синтаксического дерева.

Функция `WriteCode` получает программу в машинном коде в виде синтаксического дерева, и превращает ее в поток литер, который выводит на стандартное устройство вывода.

Модуль `CmpDic` имеет следующую интерфейсную часть:

```
//
// File: CmpDic.rfi
//
```

```
$func MakeDic      = s.Dic;
$func LookupDic    s.Key s.Dic = s.Ref;
$func AllocateDic  s.Dic s.StartAddr = s.FreeAddr;
```

Из него экспортируются четыре функции.

Функция `MakeDic` создает новый пустой словарь и выдает ссылку на него.

Функция `LookupDic` находит в словаре, на который указывает `s.Dic`, метку, соответствующую ключу `s.Key` и выдает ее. Если ключ `s.Key` ранее не был зарегистрирован в словаре, то создается новая уникальная метка, которая связывается с ключом `s.Key` и выдается в качестве значения.

Функция `AllocateDic` просматривает словарь, на который указывает `s.Dic` и связывает все зарегистрированные в нем метки с различными адресами. Если в словаре зарегистрировано N ключей, то соответствующим меткам присваиваются последовательные адреса начиная с `s.StartAddr`. В качестве результата функция выдает первый свободный адрес.

1.11.5 Головной модуль компилятора

Головной модуль компилятора связывает все части компилятора между собой. Предполагается, что имя файла, в котором находится исходная программа, передается компилятору через первый параметр в командной строке. Таким образом, компилятор вызывается следующим образом:

`Cmp` *FileName*

где *FileName* – имя файла. Это имя запрашивается с помощью библиотечной функции `Arg`.

```
//
// File Cmp.rf
//

$use Dos StdIo;
$use CmpPrs CmpGen;

$func Compile          e.FileName = ;

Main  =
  <Arg 1> :: e.FileName,
  <Compile e.FileName>;
```

```
Compile  e.FileName =
  <Channel> :: s.Chl,
  <OpenFile s.Chl e.FileName "r">,
  <Parse s.Chl> :: t.AProgram,
  <CloseChannel s.Chl>,
  <GenCode t.AProgram> :: t.Code,
  <WriteCode t.Code>;
```

1.11.6 Сканер

Результатом работы сканера является поток лексем, каждая из которых представлена двумя символами. Первый символ указывает на класс лексемы. В дальнейшем мы будем описывать синтаксис объектных выражений с помощью расширенной БНФ, нетерминалы которой записаны в виде переменных Рефала Плюс. При этом мы предполагаем, что значения нетерминалов должны соответствовать их типам. Таким образом, лексемы, поступающие из сканера имеют следующий синтаксис:

```
$    e.Tokens = { e.Token }.
$    e.Token =
$         Key s.Key | Name s.Name | Value s.Value |
$         Char s.Char.
$    s.Key  = s.Word.
$    s.Name = s.Word.
$    s.Value = s.Int.
```

Лексема вида `Key s.Key` изображает ключевое слово, где `s.Key` – символ-слово, печатное изображение которого соответствует ключевому слову. Лексема вида `Name s.Name` изображает имя переменной, где `s.Name` – символ-слово, печатное изображение которого соответствует идентификатору – имени переменной. Лексема вида `Value s.Value` изображает числовую константу, где `s.Value` – соответствующий символ-число. Лексема вида `Char s.Char` изображает неопознанную литеру `s.Char`.

Когда исходная программа прочитана до конца, сканер выдает в конце лексему `Key Eof`.

Модуль `CmpScn` имеет следующую реализацию:

```
//
// File: CmpScn.rf
//

$use StdIo Class Convert Box;
```

```
$func  ScanToken
          s.Chl e.Line = s.TokenKey s.TokenInfo (e.Line1);
$func  ScanIdRest
          (e.IdChars)  e.Chars = s.TokenKey s.Word (e.Rest);
$func  ScanIntRest
          (e.IntChars) e.Chars = s.TokenKey s.Int  (e.Rest);
$func? IsBlank           s.Char = ;
$func? IsOneCharToken  s.Char = ;
$func? CompoundToken  s.Char e.Line = s.Word e.Rest;
$func? IsKeyWord         s.Word = ;

// Ящики для хранения канала, из которого читаются лексемы,
// и остатка очередной строки.

$box ScanChl ScanLine;

InitScanner  s.Chl =              // Инициализация сканера.
  <Store &ScanChl s.Chl>,         // Прячем канал в ящик.
  <Store &ScanLine >;             // Текущая строка - пустая.

TermScanner  =                    // Завершение работы.
  <Store &ScanChl   >,            // Забываем канал
  <Store &ScanLine  >;            // и текущую строку.

ReadToken  =                      // Чтение лексемы.
  <Get &ScanChl> : s.Chl,
  <Get &ScanLine> :: e.Line,
  <ScanToken s.Chl e.Line>
        :: s.TokenKey s.TokenInfo (e.Line),
  <Store &ScanLine e.Line>,
    = s.TokenKey s.TokenInfo;

ScanToken  s.Chl e.Line =
  e.Line :
  {
  =                                   // Остаток строки -
    {                                 // пустой. Читаем
    <ReadLineCh s.Chl> :: e.Line      // следующую.
      = <ScanToken s.Chl e.Line>;
      = Key Eof ();                   // Конец файла.
    };
  s.Char e.Rest =                     // Изучаем очередную
    {                                 // литеру.
    <IsBlank s.Char>
```

```
      = <ScanToken s.Chl e.Rest>;
    <IsLetter s.Char>
      = <ScanIdRest (s.Char) e.Rest>;
    <IsDigit s.Char>
      = <ScanIntRest (s.Char) e.Rest>;
    <IsOneCharToken s.Char>
      = Key <ToWord s.Char> (e.Rest);
    <CompoundToken s.Char e.Rest> :: s.Word e.Rest
      = Key s.Word (e.Rest);
      = Char s.Char (e.Rest);          // Неопознанная литера.
    };
  };

// Отщепляем остаток идентификатора.

ScanIdRest  (e.IdChars) e.Rest =
  {
  e.Rest : s.Char e.Rest1,
              \{<IsLetter s.Char>; <IsDigit  s.Char>;}
    = <ScanIdRest (e.IdChars s.Char) e.Rest1>;
    = <ToWord <ToUpper e.IdChars>> : s.Word,
      {<IsKeyWord s.Word> = Key; = Name;} :: s.TokenKey,
      = s.TokenKey s.Word (e.Rest);
  };

// Отщепляем остаток целого числа.

ScanIntRest  (e.IntChars) e.Rest =
  {
  e.Rest : s.Char e.Rest1, <IsDigit  s.Char>
    = <ScanIntRest (e.IntChars s.Char) e.Rest1>;
    = Value <ToInt e.IntChars> (e.Rest);
  };

IsBlank  s.Char =                   // "Межевая" литера?
  ' \n\t' : e s.Char e;

IsOneCharToken  s.Char =          // Однолитерная лексема?
  ';()+-*/' : e s.Char e;

CompoundToken                     // Пытаемся отщепить
  \{                                // многолитерную лексему.
  ':=' e.Rest = ":=" e.Rest;
  '<=' e.Rest = "<=" e.Rest;
```

```
  '<>' e.Rest = "<>" e.Rest;
  '<'  e.Rest = "<"  e.Rest;
  '>=' e.Rest = ">=" e.Rest;
  '>'  e.Rest = ">"  e.Rest;
  '='  e.Rest = "="  e.Rest;
  };

IsKeyWord             // Идентификатор - ключевое слово?
  \{
  DO ; ELSE ; IF ; READ ; THEN ; WHILE ; WRITE ;
  };
```

1.11.7 Синтаксический анализатор

Задача синтаксического анализатора, находящегося в модуле `CmpPrs`, – преобразовать поток лексем в абстрактную программу, т.е. в дерево грамматического разбора.

Для выполнения синтаксического анализа, мы используем метод *нисходящего грамматического разбора*.

Рассмотрим, для примера, следующую грамматику:

```
$    Sentence = Subject Predicate.
$    Subject = "cats" | "dogs".
$    Predicate = "sleep" | "eat".
```

Пусть нам дана последовательность лексем

```
"dogs" "eat"
```

и мы хотим узнать, является ли она правильным предложением. Для этого мы должны узнать, может ли эта цепочка быть порождена нетерминалом `Sentence`("предложение"). Но из грамматики следует, что множество цепочек, порождаемых нетерминалом `Sentence`, совпадает с множеством цепочек, порождаемых цепочкой двух нетерминалов `Subject Predicate`. Таким образом, задача сводится к тому, чтобы проверить, нельзя ли от входной цепочки отщепить начало, которое порождается нетерминалом `Subject`("подлежащее"), а затем проверить, что остаток цепочки может быть порожден нетерминалом `Predicate`("сказуемое").

Как отщепить от цепочки начало, порождаемое нетерминалом `Subject`? Очевидно, что для этого достаточно проверить, что цепочка лексем начинается с лексемы `"cats"` или лексемы `"dogs"`.

Таким образом, мы приходим к следующему методу анализа.

Каждому нетерминалу A в грамматике мы ставим в соответствие функцию A, имеющую следующее объявление:

```
$func? A e.Token = e.Rest;
```

Эта функция проверяет, что входная цепочка символов `e.Token` начинается с цепочки символов, выводимой из нетерминала A. Если это действительно так, то функция A отбрасывает от `e.Token` ее начало, выводимое из нетерминала A и выдает то, что останется, в качестве результата. Если же от `e.Token` невозможно отщепить начальную цепочку, выводимую из нетерминала A, то функция A выдает в качестве результата неуспех.

Конечно, описанный метод синтаксического анализа подходит только в том случае, если для любого нетерминала A и любой входной цепочки Z существует не более чем один способ отщепить от X цепочку, выводимую из A. Во многих случаях, тем не менее, грамматика может быть преобразована к такому виду, чтобы это ограничение выполнялось. За более подробным обсуждением этого вопроса мы отсылаем читателя к [Вир 1985].

Руководствуясь вышеизложенными соображениями мы можем определить функцию `Sentence`, отщепляющую от цепочки символов начальную цепочку, выводимую из нетерминала `Sentence`, либо терпящую неуспех, если это невозможно.

```
$func? Sentence e.Token = e.Rest;
$func? Subject  e.Token = e.Rest;
$func? Predicate   e.Token = e.Rest;
$func? Token  s  e.Token = e.Rest;

Sentence  eZ =
  <Subject eZ> :: eZ,
  <Predicate  eZ> :: eZ,
    = eZ;

Subject  eZ =
  \{
  <Token "dogs" eZ> :: eZ = eZ;
  <Token "cats"  eZ> :: eZ = eZ;
  };

Predicate  eZ =
  \{
  <Token "sleep" eZ> :: eZ = eZ;
  <Token "eat" eZ> :: eZ = eZ;
  };

Token  s eZ =
  eZ : s eZ0
    = eZ0;
```

Функция `Token` введена для отщепления произвольных терминальных символов.

Теперь мы можем вернуться к рассмотрению модуля `CmpPrs`. Здесь мы сталкиваемся с двумя дополнительными проблемами.

Во-первых, исходная программа поступает из сканера не вся целиком, а постепенно, лексема за лексемой. Поэтому аргументом каждой из функций грамматического анализа является не вся цепочка лексем, а только одна лексема, которая была прочитана последней. Эта лексема является первой лексемой из еще не проанализированного остатка программы. Точно так же, функции анализа выдают не весь остаток программы, а только первую лексему из остатка программы. При этом каждая лексема представлена не одним символом, а двумя.

Во-вторых, синтаксический анализатор должен не только проверять правильность исходной программы, но еще и преобразовывать поток лексем в абстрактную программу, т.е. в дерево грамматического разбора. Поэтому функция анализа, соответствующая нетерминалу A, как правило имеет следующий формат:

```
$func A sC sI = sC sI tX;
```

где `sC sI` соответствует очередной лексеме, а `tX` – результат перевода отщепленной части программы в абстрактный синтаксис.

В третьих, если обнаруживается синтаксическая ошибка, синтаксический анализатор должен выдавать не просто неуспех, а ошибку вида `$error(`$\mathcal{E}$`)`, где $\mathcal{E}$ – сообщение, как-то описывающее ошибку. Поэтому функции анализа объявлены как безоткатные.

Абстрактная программа имеет следующий синтаксис:

```
$    t.Program = (Program t.Statement).
$    t.Statement =
$         (Assign s.Name t.Expr) |
$         (If t.Test t.Statement t.Statement) |
$         (While t.Test t.Statement) |
$         (Read s.Name) |
$         (Write t.Expr) |
$         (Seq t.Statement t.Statement)
$         (Skip).
$    t.Test = (Test s.CompOper t.Expr t.Expr).
$    t.Expr =
$         (Const s.Value) |
$         (Name s.Name) |
$         (Op t.Oper t.Expr t.Expr).
$    s.CompOper = Eq | Ne | Gt | Ge | Lt | Le.
$    s.Oper = Add | Sub | Div | Mult.
$    s.Name = s.Word.
```

```
$    s.Value = s.Int.
```

Таким образом, любая конструкция в абстрактном синтаксисе как правило принимает следующий вид:

$$(S_{keyWord}\ \mathcal{T}_1\ \mathcal{T}_2\ \ldots\ \mathcal{T}_n)$$

где ключевое слово $S_{keyWord}$ – это символ-слово, являющееся названием конструкции, а объектные термы $\mathcal{T}_1$, $\mathcal{T}_2$, ..., $\mathcal{T}_n$ – вложенные конструкции, тоже представленные в абстрактном синтаксисе. Соответствие между конструкциями в конкретном и абстрактном синтаксисе очевидно, и мы не будем останавливаться на этом вопросе более подробно.

Ниже приведена реализация модуля `CmpPrs`:

```
//
// File: CmpPrs.rf
//

$use CmpScn;

$func  Program        sC sI         = sC sI tX;
$func  StatementSeq   sC sI         = sC sI tX;
$func  RestStSeq      sC sI tX0     = sC sI tX;
$func  Statement      sC sI         = sC sI tX;
$func  Test           sC sI         = sC sI tX;
$func  Expr           sC sI         = sC sI tX;
$func  RestExpr       sC sI tX1     = sC sI tX;
$func  Term           sC sI         = sC sI tX;
$func  RestTerm       sC sI tX1     = sC sI tX;
$func  Factor         sC sI         = sC sI tX;
$func  CompOp         sC sI         = sC sI s.CompOper;
$func? AddOp          sC sI         = sC sI s.Oper;
$func? MultOp         sC sI         = sC sI s.Oper;
$func? Token          sX  sC sI     = sC sI ;
$func  Accept         sX  sC sI     = sC sI ;
$func? Name           sC sI         = sC sI s.Name;
$func? Value          sC sI         = sC sI s.Value;


Parse  s.Chl =
  <InitScanner s.Chl>,
  <Program <ReadToken>> :: sC sI t.Program,
  <TermScanner>,
  {                     // Проверяем, что остаток программы
  sC sI : Key Eof       // пуст.
```

```
    = t.Program;
    =  $error sC sI " instead of Eof after the program";
  };

Program  sC sI =                          // Программа.
  <StatementSeq sC sI> :: sC sI tX,
    = sC sI (Program tX);

StatementSeq  sC sI =                    // Последовательность
  <Statement sC sI> :: sC sI tX0,         // операторов.
    = <RestStSeq sC sI tX0>;

RestStSeq  sC sI tX0 =
  \? {
  <Token ";" sC sI> :: sC sI \!
    <StatementSeq sC sI> :: sC sI tX,
    = sC sI (Seq tX0 tX);
  \!
    = sC sI tX0;
  };

Statement  sC sI =                        // Оператор.
  \? {
  <Name sC sI> :: sC sI s.Name \!
    <Accept ":=" sC sI> :: sC sI,
    <Expr sC sI> :: sC sI t.Expr,
      = sC sI (Assign s.Name t.Expr);
  <Token "IF" sC sI> :: sC sI \!
    <Test sC sI> :: sC sI t.Test,
    <Accept "THEN" sC sI> :: sC sI,
    <Statement sC sI> :: sC sI t.Then,
    <Accept "ELSE" sC sI> :: sC sI,
    <Statement sC sI> :: sC sI t.Else,
      = sC sI (If t.Test t.Then t.Else);
  <Token "WHILE" sC sI> :: sC sI \!
    <Test sC sI> :: sC sI t.Test,
    <Accept "DO" sC sI> :: sC sI,
    <Statement sC sI> :: sC sI t.Do,
      = sC sI (While t.Test t.Do);
  <Token "READ" sC sI> :: sC sI \!
    <Name sC sI> :: sC sI s.Name,
      = sC sI (Read s.Name);
  <Token "WRITE" sC sI> :: sC sI \!
    <Expr sC sI> :: sC sI t.Expr,
```

```
      = sC sI (Write t.Expr);
  <Token "(" sC sI> :: sC sI \!
    <StatementSeq sC sI> :: sC sI t.Stmt,
    <Accept ")" sC sI> :: sC sI,
      = sC sI t.Stmt;
  \!
      = sC sI (Skip);
  };

Test  sC sI =                           // Условие.
  <Expr sC sI> :: sC sI t.Expr1,
  <CompOp sC sI> :: sC sI t.Op,
  <Expr sC sI> :: sC sI t.Expr2,
    = sC sI (Test t.Op t.Expr1 t.Expr2);

Expr  sC sI =                           // Выражение.
  <Term sC sI> :: sC sI t.X0,
    = <RestExpr sC sI t.X0>;

RestExpr  sC sI t.X1 =
  \? {
  <AddOp sC sI> :: sC sI s.Op \!
    <Term sC sI> :: sC sI t.X2,
      = <RestExpr sC sI (Op s.Op t.X1 t.X2)>;
  \!
      = sC sI t.X1;
  };

Term  sC sI =                           // Слагаемое.
  <Factor sC sI> :: sC sI t.X0,
    = <RestTerm sC sI t.X0>;

RestTerm  sC sI t.X1 =
  \? {
  <MultOp sC sI> :: sC sI s.Op \!
    <Factor sC sI> :: sC sI t.X2,
      = <RestTerm sC sI (Op s.Op t.X1 t.X2)>;
  \!
      = sC sI t.X1;
  };

Factor  sC sI =                     // Множитель.
  \? {
  <Name sC sI> :: sC sI s.Name \!
```

```
    = sC sI (Name s.Name);
  <Value sC sI> :: sC sI s.Value \!
    = sC sI (Const s.Value);
  <Token "(" sC sI> :: sC sI \!
    <Expr sC sI> :: sC sI t.Expr,
    <Accept ")" sC sI> :: sC sI,
      = sC sI t.Expr;
  \!
      $error "Invalid factor start: " sC sI;
  };

CompOp  sC sI =              // Операция отношения.
  {
  sC : Key,
  ("=" Eq) ("<>" Ne) ("<=" Le) ("<" Lt) (">=" Ge) (">" Gt)
  : e (sI s.Op) e
    = <ReadToken> s.Op;
    = $error "Invalid comparison operation: " sC sI;
  };

AddOp  Key sI =             // Аддитивная операция.
  ("+" Add) ("-" Sub) : e (sI s.Op) e
    = <ReadToken> s.Op;

MultOp  Key sI =            // Мультипликативная операция.
  ("*" Mult) ("/" Div) : e (sI s.Op) e
    = <ReadToken> s.Op;

// Эта функция пытается отщепить ключевое слово sI.
// Если это не удается, то результатом является неуспех.

Token  sI Key sI = <ReadToken>;

// Эта функция пытается отщепить ключевое слово sI.
// Если это не удается, то результатом является ошибка.

Accept
  {
  sI Key sI = <ReadToken>;
  sX sC  sI  = $error sC sI " instead of " Key sX;
  };

// Имя переменной.
```

```
Name  Name sI = <ReadToken> sI;

// Значение.

Value  Value sI = <ReadToken> sI;
```

1.11.8 Генератор кода

При генерации машинного кода порождается его промежуточное представление в виде объектного терма, имеющее следующий синтаксис:

```
$    t.Code =
$         (Seq { t.Code } ) |
$         (Instr s.Instr s.Operand) |
$         (Label s.Label) |
$         (Block s.Value).
$    s.Operand = s.Label | s.Value.
$    s.Label = s.Box.
$    s.Value = s.Int.
$
$    s.Instr =
$         Add  | Sub  | Div  | Mult  | Load  | Store |
$         Addc | Subc | Divc | Multc | Loadc |
$         Jumpeq | Jumpne | Jumplt | Jumpgt | Jumple | Jumpge
$         Jump | Read | Write | Halt |
```

Промежуточное представление может содержать метки, которые впоследствии, после ассемблирования, заменяются на адреса. Ассемблирование выполняется в два прохода. На первом проходе выясняются адреса всех инструкций и переменных, при этом адреса, соответствующие меткам, предварительно помещаются в ящики, на которые указывают метки. На втором проходе метки заменяются на их значения, т.е. ссылки на ящики заменяются не содержимое ящиков.

Модуль `CmpGen` имеет следующую реализацию:

```
//
// File: CmpGen.rf
//

$use StdIo Class Arithm Box;
$use CmpDic;

$func EncProgram     t.Program s.Dic = t.Code;
$func EncSt          t.St s.Dic = t.Code;
```

```
$func EncTest        t.Test s.Label s.Dic = t.TestC;
$func UnlessOp       s.Op = s.JumpIf;
$func EncExpr        t.Expr s.Dic = t.ExprC;
$func EncSubExpr     t.Expr sN s.Dic = t.ExprC;
$func LiteralOp      s.Op = s.OpCode;
$func MemoryOp       s.Op = s.OpCode;
$func Assemble       t.Code s.StartAddr = s.FreeAddr;
$func AssembleSeq    e.CodeSeq s.Addr = s.FreeAddr;
$func Dereference    t.Code = t.Target;
$func DereferenceSeq e.CodeSeq = e.CodeSeqD;
$func WriteCodeSeq   e.CodeSeq = ;

// Генерация кода из абстрактной программы.

GenCode  t.Program =
      // Создается пустой словарь.
  <MakeDic> :: s.Dic,
      // Компилируется абстрактная программа.
  <EncProgram t.Program s.Dic> :: t.Code,
      // Распределяется память под инструкции.
  <Assemble t.Code 1> :: s.FreeAddr,
      // Распределяется память под переменные.
  <AllocateDic s.Dic s.FreeAddr> :: s.EndAddr,
      // Метки заменяются на адреса.
  <Dereference t.Code> :: t.CodeD,
      // Генерируется директива BLOCK.
  <Sub s.EndAddr s.FreeAddr> :: s.BlockLength,
  (Seq t.CodeD (Block s.BlockLength)) :: t.Target,
    = t.Target;

// Компиляция программы.

EncProgram  (Program t.St) s.Dic =
  <EncSt t.St s.Dic> :: t.StC,
  <Box> :: s.L,
    = (Seq t.StC (Instr Halt 0) (Label s.L));

// Компиляция оператора.

EncSt  (s.KeyWord e.Info) s.Dic =
  (s.KeyWord e.Info) :
  {
  (Assign sX t.Expr) =
    <LookupDic sX s.Dic> :: s.Addr,
```

```
    <EncExpr t.Expr s.Dic> :: t.ExprC,
      = (Seq t.ExprC (Instr Store s.Addr));
  (If t.Test t.Then t.Else) =
    <Box> :: s.L1, <Box> :: s.L2,
    <EncTest t.Test s.L1 s.Dic> :: t.TestC,
    <EncSt t.Then s.Dic> :: t.ThenC,
    <EncSt t.Else s.Dic> :: t.ElseC,
      = (Seq
            t.TestC
            t.ThenC
            (Instr Jump s.L2)
          (Label s.L1)
            t.ElseC
          (Label s.L2)
        );
  (While t.Test t.Do) =
    <Box> :: s.L1, <Box> :: s.L2,
    <EncTest t.Test s.L2 s.Dic> :: t.TestC,
    <EncSt t.Do s.Dic> :: t.DoC,
      = (Seq
          (Label s.L1)
            t.TestC
            t.DoC
            (Instr Jump s.L1)
          (Label s.L2)
        );
  (Read s.X) =
    <LookupDic s.X s.Dic> :: s.Addr,
      = (Instr Read s.Addr);
  (Write t.Expr) =
    <EncExpr t.Expr s.Dic> :: t.ExprC,
      = (Seq t.ExprC (Instr Write 0));
  (Seq t.St1 t.St2) =
    <EncSt t.St1 s.Dic> :: t.StC1,
    <EncSt t.St2 s.Dic> :: t.StC2,
      = (Seq t.StC1 t.StC2);
  (Skip) =
      = (Seq );
  };

// Компиляция условия.

EncTest  (Test s.Op t.Arg1 t.Arg2) s.Label s.Dic =
  <EncExpr (Op Sub t.Arg1 t.Arg2) s.Dic> :: t.ExprC,
```

```
  <UnlessOp s.Op> :: s.JumpIf,
    = (Seq t.ExprC (Instr s.JumpIf s.Label));

UnlessOp                      // Генерация операции перехода.
  {
  Eq = Jumpne; Ne = Jumpeq;
  Lt = Jumpge; Gt = Jumple;
  Le = Jumpgt; Ge = Jumplt;
  };

// Компиляция арифметического выражения.
// При этом происходит выделение рабочих переменных
// для хранения результатов вычисления подвыражений.
// При компиляции бинарных операций выбирается такой
// порядок вычислений, который позволяет уменьшить
// количество используемых рабочих ячеек.

EncExpr  t.Expr s.Dic
  = <EncSubExpr t.Expr 0 s.Dic>;

EncSubExpr  (s.KeyWord e.Info) sN s.Dic =
  (s.KeyWord e.Info) :
  {
  (Const sC) =
      = (Instr Loadc sC);
  (Name sX) =
    <LookupDic sX s.Dic> :: s.Addr,
      = (Instr Load s.Addr);
  (Op s.Op t.Expr1 t.Expr2) =
    t.Expr2 :
    {
    (Const sC2) =
      <EncSubExpr t.Expr1 sN s.Dic> :: t.Expr1C,
      <LiteralOp s.Op> :: s.OpCode,
        = (Seq t.Expr1C (Instr s.OpCode sC2));
    (Name sX2) =
      <EncSubExpr t.Expr1 sN s.Dic> :: t.Expr1C,
      <MemoryOp s.Op> :: s.OpCode,
      <LookupDic sX2 s.Dic> :: s.Addr,
        = (Seq t.Expr1C (Instr s.OpCode s.Addr));
    (Op e) =
      <LookupDic sN s.Dic> :: s.Addr,
      <EncSubExpr t.Expr2 sN s.Dic> :: t.Expr2C,
      <Add sN 1> :: sN1,
```

```
      <EncSubExpr t.Expr1 sN1 s.Dic> :: t.Expr1C,
      <MemoryOp s.Op> :: s.OpCode,
        = (Seq
            t.Expr2C
            (Instr Store s.Addr)
            t.Expr1C
            (Instr s.OpCode s.Addr)
          );
    };
  };

LiteralOp                     // Генерация имен инструкций
  {                           // с непосредственным операндом.
  Add = Addc; Sub = Subc;
  Mult = Multc; Div = Divc;
  };

MemoryOp                      // Генерация имен инструкций
  {                           // с операндом-адресом.
  Add = Add; Sub = Sub;
  Mult = Mult; Div = Div;
  };

// Распределение памяти под инструкции.

Assemble  t.Code s.A0 =
  t.Code :
  {
  (Seq e.CodeSeq) =
      = <AssembleSeq e.CodeSeq s.A0>;
  (Instr s s) =
      = <Add s.A0 1>;
  (Label s.Label) =
    <Store s.Label s.A0>
      = s.A0;
  };

AssembleSeq  e.CodeSeq s.A0 =
  e.CodeSeq :
  {
  t.Code e.Rest =
    <Assemble t.Code s.A0> :: s.A1,
    = <AssembleSeq e.Rest s.A1>;
  =
```

```
    = s.A0;
  };

// Замена меток на их значения.

Dereference  t.Code =
  t.Code :
  {
  (Seq e.CodeSeq) =
    (Seq <DereferenceSeq e.CodeSeq>);
  (Instr s.Instr s.Value) =
    {
    <IsInt s.Value>
      = t.Code;
    <IsBox s.Value>
      = (Instr s.Instr <Get s.Value>);
    };
  (Label s.Label) =
    (Label <Get s.Label>);
  };

DereferenceSeq
  {
  t.Code e.CodeSeq =
      <Dereference t.Code><DereferenceSeq e.CodeSeq>;
  = ;
  };

// Преобразование машинного кода в поток литер
// и вывод на стандартное устройство вывода.

WriteCode
  {
  (Seq e.CodeSeq) =
      <WriteCodeSeq e.CodeSeq>;
  (Instr s.Instr s.Value) =
      <Print "   "><Print s.Instr><Print ",">
      <Print s.Value><Print ";\n">;
  (Label s.Label) =
      <Print s.Label><Print ":\n">;
  (Block s.Value) =
      <Print "   BLOCK,"><Print s.Value><Print ";\n">;
  };
```

```
WriteCodeSeq
  {
  t.Code e.CodeSeq =
      <WriteCode t.Code><WriteCodeSeq e.CodeSeq>;
  = ;
  };
```

1.11.9 Модуль работы со словарем

Словари реализованы в виде бинарных деревьев [АХУ 1979]. Каждый узел дерева представлен ящиком, который содержит три символа: ключ, значение ключа, ссылки на левое поддерево и ссылки на правое поддерево. Пустое дерево представляется ссылкой на пустой ящик.

Модуль `CmpDic` имеет следующую реализацию:

```
//
// File: CmpDic.rf
//

$use Box Compare Arithm;

// Создание нового пустого словаря.

MakeDic = <Box>;

// Поиск в словаре s.Dic метки, связанной с ключом
// s.Key. Если ключа s.Key еще нет в словаре, он заносится
// в словарь и связывается с новой уникальной меткой.

LookupDic  s.Key s.Dic =
  <Get s.Dic> :
  {
  =
    <Box> :: s.Ref,
    <Store s.Dic s.Key s.Ref <Box> <Box>>,
      = s.Ref;
  s.Key1 s.Ref1 s.DicL s.DicR =
    <Compare (s.Key)(s.Key1)> :
    {
    '<' = <LookupDic s.Key s.DicL>;
    '>' = <LookupDic s.Key s.DicR>;
    '=' = s.Ref1;
    };
  };
```

```
// Распределение памяти под метки, находящиеся в словаре.
// s.A - начальный адрес. Адреса, соответствующие меткам,
// заносятся в ящики, на которые указывают метки.

AllocateDic  s.Dic s.A =
  <Get s.Dic> :
  {
  = s.A;
  s.Key s.Ref s.DicL s.DicR =
    <AllocateDic s.DicL s.A> :: s.A,
    <Store s.Ref s.A>,
    <Add s.A 1> :: s.A,
      = <AllocateDic s.DicR s.A>;
  };

WriteDic  s.Dic =
  <Get s.Dic> :
  {
  = <Print "_">;
  s.Key s.Ref s.DicL s.DicR =
    <Print "(">, <WriteDic s.DicL>, <Print " ">,
    <Write s.Key>,
    <Get s.Ref> :
      {
      = ;
      e.Value = <Print "->"> <Write e.Value>;
      },
    <Print " ">,
    <WriteDic s.DicR>, <Print ")">;
  };
```

Глава 2

Синтаксис и семантика Рефала Плюс

2.1 Нотация для записи синтаксиса

Для описания синтаксиса используется расширенная форма Бекуса-Наура (РБНФ).

Синтаксические понятия (нетерминалы) обозначаются словами, выражающими их интуитивный смысл. Цепочки терминальных символов заключаются в двойные кавычки либо апострофы, с тем, чтобы их можно было отличить от нетерминалов.

Если конструкция A представляет собой конкатенацию конструкций B и C, т.е. состоит из конструкции B и следующей вслед за ней конструкции C, мы будем называть B и C *синтаксическими множителями* и описывать A следующей синтаксической формулой:

A `=` $B\,C$`.`

Если конструкция A является либо конструкцией B, либо конструкцией C, мы будем называть B и C *синтаксическими слагаемыми* и описывать A следующей синтаксической формулой:

A `=` B `|` C`.`

Для группировки синтаксических слагаемых и множителей можно использовать круглые скобки.

Если конструкция A представляет собой либо конструкцию B, либо является пустой цепочкой, это выражается в виде

A `=` `[`B`]`.

Если же A представляет собой произвольную (может быть пустую) последовательность из конструкций B, это выражается в виде

A `=` `{`B`}`.

Приведем несколько примеров того, как множества предложений описываются формулами в РБНФ.

```
(A|B)(C|D)      A C, A D, B C, B D
A[B]C           A B C, A C
A {B A}         A, A B A, A B A B A, A B A B A B A, ...
{A|B} C         C, A C, B C, A A C, A B C, B B C, B A C, ...
```

Чтобы отличать описания синтаксиса на РБНФ от окружающего их текста на русском языке мы будем помечать все строчки, в которых находится текст на РБНФ, литерой `$` в первой позиции.

Поскольку сама РБНФ представляет собой некоторый язык, ее синтаксис также может быть описан с помощью РБНФ следующим образом:

```
$    Синтаксис      = { СинтФормула }.
$    СинтФормула    = Идентификатор "=" СинтВыражение ".".
$    СинтВыражение  = СинтСлагаемое { "|" СинтСлагаемое }.
$    СинтСлагаемое  = СинтМножитель { СинтМножитель }.
$    СинтМножитель  = Идентификатор | Цепочка |
$         "(" СинтВыражение ")" | "[" СинтВыражение "]" |
$         "{" СинтВыражение "}".
```

2.2 Естественный метод описания семантики

Для описания процесса выполнения программ на Рефале Плюс будет использован метод, известный под названием “естественная семантика” или “структурная операционная семантика” [Плоткин 1983], [Апт 1983].

Название “естественная семантика” объясняется сходством этого метода описания с генценовским естественным выводом в математической логике. При таком способе описания, семантикой языка является неупорядоченное множество *утверждений* о программах и фрагментах программ.

Например, предположим, что в описываемом языке имеются выражения, которые могут содержать переменные и вычисление которых может приводить к побочным эффектам (например, из-за выполнения операций ввода-вывода). Тогда определение семантики языка может включать утверждения вида

$$\rho, \sigma' \vdash E \Rightarrow X, \sigma''$$

где E - выражение языка, ρ - среда, которая связывает переменные с их значениями в том контексте, в котором находится E, σ' и σ'' - глобальное состояние до начала вычисления E и после вычисления E, а X - результат вычисления E. Глобальное состояние может включать состояние памяти, состояние файлов и т.п.

Смысл таких утверждений неформально может быть истолкован следующим образом: если вычисление выражения E начинается в среде ρ и глобальном состоянии σ', оно может завершиться получением значения X в глобальном состоянии σ''.

Знак " $\vdash$ " (который можно читать как "влечет" или "вызывает") служит напоминанием о том, что результат вычисления выражения E зависит от среды ρ, в которой вычисляется выражение, ибо это выражение может содержать переменные.

Таким образом, задача описания семантики языка при таком подходе сводится к задаче описания множества истинных утверждений о программах и их фрагментах.

"Естественное" описание семантики представляет собой *неупорядоченное* множество правил вывода, позволяющих из одних истинных утверждений выводить другие.

Каждое правило вывода выглядит как дробь, в числителе которой находятся *посылки*, а в знаменателе – *заключение*. Если у правила нет ни одной посылки, оно называется *аксиомой*, и в этом случае горизонтальная черта может опускаться.

Кроме того, для каждого правила могут иметься дополнительные условия, которые накладывают какие-то ограничения на вид посылок и заключения. Эти ограничения мы будем записывать справа от правила или непосредственно под ним.

Например, предположим, что в описываемом языке имеется конструкция `if` E `then` E' `else` E'', смысл которой неформально может быть описан следующим образом.

Вычислить E. Если получится `true`, то вычислить E' и полученный результат считать результатом вычисления всей конструкции. Если же в результате вычисления E получилось `false`, то вычислить E'' и полученный результат считать результатом вычисления всей конструкции.

Недостатком такого описания является то, что мы забыли указать, в какой среде происходит вычисление каждого из выражений E, E' и E''. Поэтому переформулируем описание более точно.

Если результатом вычисления выражения E в среде ρ является `true`, а результатом вычисления выражения E' в среде ρ является X, то X является результатом вычисления конструкции `if` E `then` E' `else` E'' в среде ρ.

Если результатом вычисления выражения E в среде ρ является `false`, а результатом вычисления выражения E'' в среде ρ является X, то X является результатом вычисления конструкции `if` E `then` E' `else` E'' в среде ρ

Это многословное определение можно записать в более краткой и

наглядной форме в виде двух правил вывода:

$$\frac{\begin{array}{lll}\rho,\sigma & \vdash & E \Rightarrow \texttt{ true } ,\sigma' \\ \rho,\sigma' & \vdash & E' \Rightarrow X,\sigma''\end{array}}{\rho,\sigma \quad \vdash \quad \texttt{if } E \texttt{ then } E' \texttt{ else } E'' \Rightarrow X,\sigma''}$$

$$\frac{\begin{array}{lll}\rho,\sigma & \vdash & E \Rightarrow \texttt{ false } ,\sigma' \\ \rho,\sigma' & \vdash & E'' \Rightarrow X,\sigma''\end{array}}{\rho,\sigma \quad \vdash \quad \texttt{if } E \texttt{ then } E' \texttt{ else } E'' \Rightarrow X,\sigma''}$$

Обратите внимание, что формализованное описание семантики, в отличие от словесного, точно определяет как изменяется глобальное состояние в процессе вычисления.

2.3 Лексическая структура программы

Программа на языке Рефал Плюс представляет собой конечную последовательность литер. Анализ программы проводится в два этапа. Сначала производится сканирование программы, в результате чего поток литер разбивается на лексемы. Затем результат сканирования подвергается синтаксическому анализу, результатом которого является дерево синтаксического анализа. В связи с этим описание синтаксиса состоит из двух частей: сначала описывается лексическая структура программы, т.е. правила изображения лексем с помощью цепочек литер, а затем – синтаксическая структура, т.е. правила, по которым программы составляются из лексем.

```
$    Программа = { Межа Лексема } Межа.
$    Межа = РазделительЛексем | Комментарий.
$    РазделительЛексем = Пробел | Табуляция | КонецСтроки.
```

Программа представляет собой конечную последовательность лексем. В качестве разделителей лексем используются следующие литеры: пробел, табуляция и конец строки. Избыточные разделители лексем могут появляться как до, так и после лексемы, но не внутри лексем.

Если две соседние лексемы могут быть однозначно отделены одна от другой, разделители лексем между ними могут отсутствовать.

2.3.1 Комментарии

```
$    Комментарий = "//" ОкончаниеКомментария КонецСтроки
$         | "/*" ТелоКомментария "*/".
$    ОкончаниеКомментария =
$         любая цепочка литер, не содержащая КонецСтроки.
$    ТелоКомментария =
$         любая цепочка литер, не содержащая "*/".
```

Существует два способа вставлять комментарии между лексемами. Первый способ – поставить литеры //. При этом весь остаток строки игнорируется. Второй способ – заключить комментарий в пару "комментарных скобок" /* и */. Например:

```
// Это - комментарий.
          // И это - комментарий.
     /* И это - тоже! */
```

2.3.2 Лексемы

```
$    Лексема =
$         Скобка | КлючевоеСлово |
$         ИзображениеЦепочкиЛитер |
$         ИзображениеСлова | ИзображениеЧисла |
$         Переменная.

$    Скобка = "(" | ")" | "{" | "\{" | "}" | "<" | ">".
```

Лексемы представляют собой скобки, ключевые слова, изображения цепочек символов-литер, изображения символов-слов, изображения символов-чисел, и переменные.

2.3.3 Ключевые слова

```
$    КлючевоеСлово =
$         "$box" | "$channel | "$const" | "$error" | "$fail" |
$         "$func" | "$func?" | "$iter" | "$l" | "$r" |
$         "$string" | "$table" | "$trace | | "$traceall" |
$         "$trap" | "$use" | "$vector" | "$with" |
$         "#" | "&" | "," | ":" | "::" | ";" | "=" |
$         "\?" | "\!".
```

2.3.4 Символы-литеры

```
$    ИзображениеЦепочкиЛитер =
$         "'" { ИзображениеЛитеры } "'".

$    ИзображениеЛитеры =
$         ИзображениеОбычнойЛитеры | ИзображениеОсобойЛитеры
$         ШестнадцатиричноеИзображениеЛитеры.
$    ИзображениеОбычнойЛитеры =
$         любая литера, кроме апострофа ('), кавычки ("),
$         обратной косой (\) и конца строки.
$    ИзображениеОсобойЛитеры =
```

```
$          "\n" | "\t" | "\b" | "\r" | "\f" |
$          "\\" | "\'" | '\"' .
$     ШестнадцатиричноеИзображениеЛитеры =
$          "\x" ШестнЦифра ШестнЦифра.

$     Цифра =
$          "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" |
$          "8" | "9".

$     ШестнЦифра =
$           Цифра |
$           "A" | "B" | "C" | "D" | "E" | "F" |
$           "a" | "b" | "c" | "d" | "e" | "f".
```

Символ-литера соответствует одной литере и записывается в виде изображения этой литеры, заключенного в апострофы. Например:

```
'A' 'a' '7' '$'
```

Как правило, изображением литеры является сама эта литера, за исключением следующих литер, для которых предусмотрены особые обозначения:

Новая строка (перевод строки)	HL (LF)	`'\n'`
Горизонтальная табуляция	HT	`'\t'`
Возврат на шаг	BS	`'\b'`
Возврат каретки	CR	`'\r'`
Перевод формата	FF	`'\f'`
Обратная косая		`'\\'`
Апостроф	'	`'\''`
Двойная кавычка	"	`'\"'`

Изображения литер, входящие в изображения цепочек символов-литер и в изображения символов-слов, могут записываться в виде

```
\xZZ
```

где `ZZ` - две шестнадцатиричные цифры, задающие однобайтовый код литеры. При этом символы-слова, содержащие такие изображения литер, должны заключаться в двойные кавычки.

Например, `\x2A` и `\x2a` эквивалентны `*`.

Последовательность из нескольких символов-литер может быть записана в виде одной последовательности изображений литер, заключенной в апострофы. Например:

```
'ABC'
'123'
'\"I don\'t like swimming!\" - said a little girl.'
```

Таким образом, последовательность из трех символов-литер `'A'`, `'B'` и `'C'` может быть записана любым из перечисленных ниже способов:

```
'A' 'B' 'C'
'A''B''C'
'ABC'
```

2.3.5 Символы-слова

```
$    ИзображениеСлова =
$         Идентификатор |
$         '"' { ИзображениеЛитеры } '"'.

$    Идентификатор = НачалоИдентификатора ХвостИдентификатора.
$    НачалоИдентификатора = ПрописнаяБуква | "_".
$    ХвостИдентификатора = { Буква | Цифра | "_" }.

$    Буква = ПрописнаяБуква | СтрочнаяБуква.

$    ПрописнаяБуква =
$         "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" |
$         "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" |
$         "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z".

$    СтрочнаяБуква =
$         "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" |
$         "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" |
$         "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z".
```

Символ-слово соответствует цепочке литер и записывается в виде цепочки изображений этих литер, заключенной в двойные кавычки. Изображения литер такие же, как и при записи символов-литер. Например:

```
"ABC"
"123"
"\"I don\'t like swimming!\" - said a little girl."
```

Следует обратить внимание, что `"ABC"` является изображением одного символа-слова, в то время как `'ABC'` – это изображение цепочки из

трех символов-литер. Кроме того, считается, что все символы-слова, состоящие из одной литеры, не совпадают с соответствующими символами-литерами. Например, символ-литера `'A'` и символ-слово `"A"` – разные символы.

В некоторых случаях символы-слова разрешается записывать без окружающих двойных кавычек. Для этого должны быть выполнены следующие условия.

Во-первых, этот символ-слово может содержать только следующие литеры: прописные буквы, строчные буквы, цифры, подчеркивание.

Во-вторых, этот символ-слово должен начинаться с прописной буквы или подчеркивания.

Например, ниже приведено два изображения одного и того же символа-слова:

```
I_do_not_like_swimming
"I_do_not_like_swimming"
```

2.3.6 Символы-числа

```
$    ИзображениеЧисла =
$          [ "+" | "-" ] Цифра { Цифра } |
$          "0x" ШестнЦифра { ШестнЦифра }.
```

Символы-числа соответствуют целым числам со знаком. Они могут быть неограниченного размера. Например:

```
123   +121   -123   -12345678901234567890123456789O
```

Неотрицательные целые числа можно записывать в виде

```
0xZZZ...ZZ
```

где `ZZZ...ZZ` – непустая последовательность шестнадцатиричных цифр.

Например, `0xFF` и `0xff` эквивалентны `255`.

2.3.7 Переменные

```
$    Переменная =
$         s-переменная | t-переменная | e-переменная |
$         v-переменная.
$    s-переменная = "s" [ "." ] ИндексПеременной.
$    t-переменная = "t" [ "." ] ИндексПеременной.
$    v-переменная = "v" [ "." ] ИндексПеременной.
$    e-переменная = "e" [ "." ] ИндексПеременной.

$    УказательТипаПеременной = "s" | "t" | "v" | "e".
$    ИндексПеременной = ХвостИдентификатора.
```

Переменная состоит из указателя типа переменной, необязательной точки и индекса переменой. Например:

```
tHead  eTail e.1 e1  tX s t e
```

При этом, `eI` и `e.I` – различные изображения одной и той же переменной.

Если две переменные стоят рядом, они должны быть разделены межой. Например, `sAeB` является одной переменной, в то время как `sA eB` – две переменные.

Если индекс переменной отсутствует, это означает, что подразумевается, что эта переменная имеет уникальный индекс и отличается от всех прочих переменных в программе. Таким образом, например, если в программе не используются переменные `e1000` и `e2000`, то запись `e e` может быть заменена на `e1000 e2000`.

Переменные делятся на четыре класса: s-переменные, t-переменные, v-переменные и e-переменные, в соответствии с их указателями типа.

2.3.8 Нормализация потока лексем

В процессе ввода программы сканер разбивает входной поток литер на лексемы. Многие лексемы имеют один и тот же смысл. Так, например, все три лексемы

```
125  000125  +125
```

обозначают одно и то же число 125.

Именно поэтому при описании лексической структуры программы мы употребляли такие термины как “изображение числа” и “изображение слова” вместо “число” и “слово”.

Кроме того, такая лексема как “изображение цепочки литер” является изображением не одного синтаксического объекта, а конечной последовательности литер.

В дальнейшем, при описании синтаксиса, мы будем считать, что сканер произвел “нормализацию” лексем, т.е. преобразовал лексемы к стандартному виду таким образом, чтобы одним и тем же синтаксическим объектам всегда соответствовали одни и те же лексемы. При этом изображение цепочки из N литер разбивается на N отдельных лексем, каждая из которых изображает ровно одну литеру. Таким образом, появляется возможность при описании синтаксиса говорить не об “изображениях объектов”, а об “объектах”.

Соответствие между исходными лексемами и нормализованными лексемами следующее:

`ИзображениеЦепочкиЛитер`	$\Rightarrow$	$\texttt{Литера}_1\ \texttt{Литера}_2 \ldots \texttt{Литера}_n$
`ИзображениеСлова`	$\Rightarrow$	`Слово`
`ИзображениеЧисла`	$\Rightarrow$	`Число`

Не следует путать литеры, из которых составлены исходные лексемы с символами-литерами, которые получаются на выходе сканера.

2.4 Объекты и значения

Под *объектами* обычно понимают некоторые сущности, которые существуют во времени, могут изменяться, но при этом остаются сами собой.

Под *значениями* обычно понимают некоторые сущности, которые не могут изменяться и, в этом смысле, находятся вне времени.

Конечно, значение можно считать частным, вырожденным случаем объекта (а именно, застывшим объектом, не способным к развитию), но мы все же обычно будем называть "объектами" только такие объекты, которые не являются значениями.

Иметь дело с объектами труднее, чем со значениями, ибо они могут изменяться. Поэтому объекты часто снабжают *именами*. Основным свойством имени является то, что оно однозначно связано с объектом (однозначно идентифицирует этот объект). В отличие от самих объектов имена являются типичными значениями, поскольку они не меняются из-за того, что изменяется сам объект.

Программы на Рефале Плюс имеют дело как с объектами, так и со значениями.

Все значения, с которыми работают Рефал-программы, являются *объектными выражениями*, которые представляют собой конечные последовательности *символов* и *скобок*, правильно построенные относительно скобок. Скобки служат для придания объектным выражениям древовидной структуры, в то время как символы представляют собой элементарные данные (литеры, числа, слова и ссылки на объекты).

Объекты, с которыми работают Рефал-программы, могут содержать внутри себя объектные выражения (в частности – ссылки на объекты). Содержимое объектов может изменяться в процессе работы Рефал-программы. Доступ к объектам осуществляется через их имена, в качестве которых служат символы-ссылки.

Объекты создаются как во время компиляции Рефал-программы, так и во время ее исполнения. С теоретической точки зрения, любой объект, будучи создан, существует бесконечно. На практике, однако, Рефал-программы исполняются на компьютерах, имеющих конечный объем памяти, поэтому все реализации Рефала Плюс должны предусматривать *сборку мусора*. Сборка мусора обеспечивает автоматическое уничтожение тех объектов, которые недоступны программе и которые, в силу этого, заведомо не могут повлиять на дальнейшую работу программы.

2.5 Объектные выражения

2.5.1 Синтаксис объектных выражений

```
$    ОбъектноеВыражение = { ОбъектныйТерм }.
$    ОбъектныйТерм = Символ | "(" ОбъектноеВыражение ")".
```

В дальнейшем мы будем обозначать объектные выражения через $\mathcal{E}$, объектные термы – через $\mathcal{T}$, а символы – через $\mathcal{S}$.

2.5.2 Статические и динамические символы

```
$    Символ = СтатическийСимвол | ДинамическийСимвол.
$    СтатическийСимвол = Литера | Слово | Число.
$    ДинамическийСимвол = ССылкаНаФункцию | ССылкаНаТаблицу |
$         ССылкаНаЯщик | ССылкаНаВектор |
$         ССылкаНаСтроку | ССылкаНаКанал.
```

Символы делятся, на две категории: статические и динамические.

К статическим символам относятся символы-литеры, символы-слова и символы-числа.

Статические символы существуют “объективно”: статический символ может быть выведен в канал ввода-вывода, а затем введен обратно. При этом мы получим опять тот же символ. В этом смысле статические символы существуют еще до начала исполнения программы и продолжают существовать после завершения исполнения программы.

Динамические символы являются ссылками на объекты. А именно, каждый динамический символ представляет собой указатель на то место в памяти, где находится объект во время исполнения программы. При этом объект может быть определением функции, ящиком, вектором, строкой, таблицей или каналом.

Динамические символы, в противоположность статическим, “субъективны”. Они создаются в момент загрузки программы и в процессе ее исполнения. Хотя их можно вывести в канал ввода-вывода, их нельзя ввести обратно. Когда выполнение программы завершается, все динамические символы, созданные в процессе ее выполнения теряют всякий смысл.

К динамическим символам относятся ссылки на функции, ссылки на ящики, ссылки на векторы, ссылки на строки, ссылки на таблицы и ссылки на каналы.

2.5.3 Символические имена

Различным сущностям в программах на Рефале Плюс могут присваиваться *символические имена*, которые, с синтаксической точки зрения, являются идентификаторами.

А именно, в программах могут использоваться:

- Имена констант (раздел 2.12.1).
- Имена функций (раздел 2.12.2).
- Имена ссылок на объекты (раздел 2.12.3).
- Имена модулей (раздел 2.14).

Подробнее об этом будет сказано в соответствующих разделах.

2.5.4 Символические имена выражений

```
$    ИменованноеВыражение = "&" ИмяВыражения.
$    ИмяВыражения = Идентификатор.
```

Объектным выражениям в программе на Рефале Плюс могут присваиваться символические имена. Каждое из этих имен обозначает либо статическое объектное выражение (раздел 2.12.1), либо ссылку на объект (раздел 2.12.2) либо ссылку на функцию (раздел 2.12.3).

Если имя $\mathcal{N}$ обозначает объектное выражение $\mathcal{E}$, то конструкция вида `&`$\mathcal{N}$ означает, что в этом месте должно быть подставлено объектное выражение $\mathcal{E}$.

Поскольку все объекты, как и ссылки на них, создаются при компиляции программы или в процессе ее загрузки и исполнения, символы-ссылки не могут появляться в тексте программы непосредственно. Тем не менее, если объект объявлен в программе, то ссылке на него присваивается символическое имя, и, таким образом, эта ссылка может быть подставлена в нужные места программы.

2.5.5 Устранение символических имен выражений

Если $\mathcal{N}$ является символическим именем объектного выражения $\mathcal{E}$, то все использования этого имени в программе можно устранить, заменив каждую конструкцию `&`$\mathcal{N}$ на $\mathcal{E}$.

В дальнейшем, при описании контекстных ограничений и семантики языка, мы будем предполагать, что это преобразование уже выполнено, и программа не содержит использований символический имен. При этом, с другой стороны, в тексте преобразованной программы могут появиться динамические символы.

2.6 Значения переменных и среды

Вычисление любой конструкции Рефала Плюс требует знания значений переменных, входящих в эту конструкцию. Эта информация о зна-

чениях переменных естественным образом представляется в виде функции с конечной областью определения, ставящей в соответствие каждой переменной из ее области определения значение этой переменной.

А именно, пусть ρ – среда с областью определения $\{V_1, \ldots, V_n\}$. Тогда $\rho(V_j) = \mathcal{E}_j$ является значением переменной V_j. Такую среду мы будем обозначать через $\{V_1 = \mathcal{E}_1, \ldots, V_n = \mathcal{E}_n\}$. В частности, пустая среда будет обозначаться через $\{\}$.

Область определения среды ρ мы будем обозначать через $\mathrm{dom}[\rho]$. Таким образом, $\mathrm{dom}[\{V_1 = \mathcal{E}_1, \ldots, V_n = \mathcal{E}_n\}] = \{V_1, \ldots, V_n\}$.

При этом, без дальнейших оговорок, мы всегда будем предполагать, что переменные обязательно имеют значения, соответствующие их типам. Т.е. значениями s-переменных могут быть только символы, значениями t-переменных - только объектные термы, значениями e-переменных – произвольные объектные выражения, а значениями v-переменных - непустые объектные выражения.

В дальнейшем $\rho + \rho'$ будет означать пополнение среды ρ значениями переменных из среды ρ', которое строится следующим образом.

$\mathrm{dom}[\rho + \rho']$ содержит все переменные из $\mathrm{dom}[\rho']$, а также все переменные из $\mathrm{dom}[\rho]$, индексы которых отличаются от индексов переменных из $\mathrm{dom}[\rho']$.

Для любой переменной V из $\mathrm{dom}[\rho + \rho']$, если $\rho'(V)$ определено, то $(\rho + \rho')(V) = \rho'(V)$, а если $\rho'(V)$ не определено, то $(\rho + \rho')(V) = \rho(V)$.

Например

```
{sX = 1, sY = 2} + {sY = 200, sZ = 300}
     = {sX = 1, sY = 200, sZ = 300}

{sX = 1, sY = 2} + {eY = 200, sZ = 300}
     = {sX = 1, eY = 200, sZ = 300}
```

2.7 Результатные выражения

2.7.1 Синтаксис

```
$    РезультатноеВыражение =
$         { РезультатныйТерм | ИменованноеВыражение }.
$    РезультатныйТерм =
$         СтатическийСимвол | Переменная |
$         "(" РезультатноеВыражение ")" |
$         ВызовФункции.
$    ВызовФункции =
$         "<" ИмяФункции АргументВызова ">".
$    ИмяФункции = Идентификатор.
$    АргументВызова =
```

```
$          РезультатноеВыражение.
```

В дальнейшем мы будем обозначать результатные выражения через Re, результатные термы – через Rt, переменные – через V, е-переменные - через Ve, а имена функций – через $\mathcal{F}$.

2.7.2 Вычисление результатных выражений

Результатные выражения служат для порождения объектных выражений.

Результатное выражение Re может быть вычислено в любой среде ρ, при условии, что ρ дает значения для всех переменных, входящих в Re.

Если вычисление результатного выражения Re заканчивается, то результатом этого вычисления является либо объектное выражение $\mathcal{E}$, либо неуспех `$fail(0)`, либо ошибка `$error(`$\mathcal{E}$`)`, где $\mathcal{E}$ – сообщение об ошибке.

Вычисление вызовов функций может приводить к изменению глобального состояния программы (например, в случае выполнения операций ввода-вывода или каких-то действий над объектами), поэтому, если результатное выражение содержит вызовы функций, его вычисление может также привести к изменению глобального состояния.

Запись $\rho, \sigma \vdash Re \Rightarrow X, \sigma'$ означает, что результатом вычисления результатного выражения Re в среде ρ является X. При этом, если вычисление началось в глобальном состоянии σ, оно закончится в глобальном состоянии σ'.

Результатное выражение Re вычисляется слева направо. При этом вместо переменных подставляются их значения и выполняются вызовы функций.

Вычисление вызовов функций выполняется следующим образом. Если вызов имеет вид <$\mathcal{F}$ Re>, то вычисляется результатное выражение Re. Если в результате получится объектное выражение $\mathcal{E}$, то функция $\mathcal{F}$ применяется к $\mathcal{E}$.

Запись $\sigma \vdash$ <$\mathcal{F}$ $\mathcal{E}$> $\Rightarrow X, \sigma'$ означает, что результатом применения функции $\mathcal{F}$ к объектному выражению $\mathcal{E}$ является X, причем, если вычисление началось в глобальном состоянии σ, оно закончится в глобальном состоянии σ'.

$$\rho, \sigma \vdash \Rightarrow, \sigma$$

$$\frac{\rho, \sigma \vdash Re \Rightarrow \mathcal{E}', \sigma' \qquad \rho, \sigma' \vdash Rt \Rightarrow \mathcal{E}'', \sigma''}{\rho, \sigma \vdash Re\ Rt \Rightarrow \mathcal{E}'\ \mathcal{E}'', \sigma''}$$

$$\frac{\rho, \sigma \vdash Re \Rightarrow \mathcal{E}', \sigma' \qquad \rho, \sigma' \vdash Rt \Rightarrow \texttt{\$fail(0)}, \sigma''}{\rho, \sigma \vdash Re\ Rt \Rightarrow \texttt{\$fail(0)}, \sigma''}$$

$$\frac{\begin{array}{lll}\rho,\sigma & \vdash & Re \Rightarrow \mathcal{E}',\sigma' \\ \rho,\sigma' & \vdash & Rt \Rightarrow \texttt{\$error}(\mathcal{E}''),\sigma''\end{array}}{\rho,\sigma \quad \vdash \quad Re\ Rt \Rightarrow \texttt{\$error}(\mathcal{E}''),\sigma''}$$

$$\frac{\rho,\sigma \quad \vdash \quad Re \Rightarrow \texttt{\$fail(0)},\sigma'}{\rho,\sigma \quad \vdash \quad Re\ Rt \Rightarrow \texttt{\$fail(0)},\sigma'}$$

$$\frac{\rho,\sigma \quad \vdash \quad Re \Rightarrow \texttt{\$error}(\mathcal{E}'),\sigma'}{\rho,\sigma \quad \vdash \quad Re\ Rt \Rightarrow \texttt{\$error}(\mathcal{E}'),\sigma'}$$

$$\begin{array}{lll}\rho,\sigma & \vdash & \mathcal{S} \Rightarrow \mathcal{S},\sigma \\ \rho,\sigma & \vdash & V \Rightarrow \mathcal{E},\sigma \\ & & \quad \texttt{где}\ \mathcal{E} = \rho(V).\end{array}$$

$$\frac{\rho,\sigma \quad \vdash \quad Re \Rightarrow \mathcal{E},\sigma'}{\rho,\sigma \quad \vdash \quad (Re) \Rightarrow (\mathcal{E}),\sigma'}$$

$$\frac{\rho,\sigma \quad \vdash \quad Re \Rightarrow \texttt{\$fail(0)},\sigma'}{\rho,\sigma \quad \vdash \quad (Re) \Rightarrow \texttt{\$fail(0)},\sigma'}$$

$$\frac{\rho,\sigma \quad \vdash \quad Re \Rightarrow \texttt{\$error}(\mathcal{E}),\sigma'}{\rho,\sigma \quad \vdash \quad (Re) \Rightarrow \texttt{\$error}(\mathcal{E}),\sigma'}$$

$$\frac{\begin{array}{lll}\rho,\sigma & \vdash & Re \Rightarrow \mathcal{E},\sigma' \\ \sigma' & \vdash & \texttt{<}\mathcal{F}\ \mathcal{E}\texttt{>} \Rightarrow X,\sigma''\end{array}}{\rho,\sigma \quad \vdash \quad \texttt{<}\mathcal{F}\ Re\texttt{>} \Rightarrow X,\sigma''}$$

$$\frac{\rho,\sigma \quad \vdash \quad Re \Rightarrow \texttt{\$fail(0)},\sigma'}{\rho,\sigma \quad \vdash \quad \texttt{<}\mathcal{F}\ Re\texttt{>} \Rightarrow \texttt{\$fail(0)},\sigma'}$$

$$\frac{\rho,\sigma \quad \vdash \quad Re \Rightarrow \texttt{\$error}(\mathcal{E}),\sigma'}{\rho,\sigma \quad \vdash \quad \texttt{<}\mathcal{F}\ Re\texttt{>} \Rightarrow \texttt{\$error}(\mathcal{E}),\sigma'}$$

2.7.3 Примеры

Ниже приведены примеры результатных выражений:

```
(A B) C D
t.Head e.Tail
While t.Condition Do t.Statement
<Mult sN <Factorial <Sub sN 1>>
```

Следующие результатные выражения являются результатными термами:

```
(A B)
t.Head
<Mult sN <Factorial <Sub sN 1>>
```

Пусть $\rho =$ `{sM = 2, sN = 3, eA = A B C, tB = (D E F)}`, и пусть `Add` – имя функции, выполняющей сложение целых чисел, а `Mult` – имя функции, выполняющей умножение целых чисел, т.е. в частности имеет место

$$\sigma \vdash \texttt{<Add 3 100>} \Rightarrow 103, \sigma$$
$$\sigma \vdash \texttt{<Mult 2 103>} \Rightarrow 206, \sigma$$

для любого глобального состояния σ, ибо функции `Add` и `Mult` не изменяют глобальное состояние.

Тогда имеем

$$\rho, \sigma \vdash \texttt{eA (eA tB) tB} \Rightarrow \texttt{A B C (A B C (D E F)) (D E F)}, \sigma$$
$$\rho, \sigma \vdash \texttt{<Mult sM <Add sN 100>>} \Rightarrow 206, \sigma$$

для любого глобального состояния σ.

2.8 Образцы

2.8.1 Синтаксис

```
$    Образец = УказательНаправления ОбразцовоеВыражение.
$    УказательНаправления = [ "$l" | "$r" ].

$    ОбразцовоеВыражение =
$         { ОбразцовыйТерм | ИменованноеВыражение }.
$    ОбразцовыйТерм =
$         СтатическийСимвол | Переменная |
$         "(" ОбразцовоеВыражение ")".
```

Образец представляет собой образцовое выражение, перед которым может стоять указатель направления `"$l"` или `"$r"`. Указатель направления `"$l"` означает что сопоставление с образцом должно выполняться слева направо. Указатель `"$r"` – что справа налево. Если указатель направления отсутствует, подразумевается указатель `"$l"`.

В дальнейшем мы будем обозначать образцы через P, образцовые выражения – через Pe, образцовые термы – через Pt, а указатели направления сопоставления – через D.

2.8.2 Сопоставление с образцом

Мы говорим, что среда ρ' является результатом сопоставления объектного выражения $\mathcal{E}$ с образцом P в начальной среде ρ, если выполнены следующие условия.

- Среда ρ' является расширением среды ρ, т.е. область определения среды ρ' является расширением области определения среды ρ, и для любой переменной V из области определения среды ρ имеет место $\rho'(V) = \rho(V)$.

- Если все переменные, входящие в P, заменить на их значения в соответствии с ρ', а указатель направления отбросить, получится объектное выражение $\mathcal{E}$.

В этом случае среду ρ' мы будем также называть вариантом сопоставления $\mathcal{E}$ с P в среде ρ, а множество таких вариантов сопоставления будем обозначать через $\mathbb{M}(\rho, \mathcal{E}, P)$.

На множестве $\mathbb{M}(\rho, \mathcal{E}, P)$ вводится отношение порядка следующим образом.

Пусть $\mathbb{M}(\rho, \mathcal{E}, P)$ содержит два различных варианта сопоставления ρ' и ρ''. Рассматриваются все вхождения переменных в P.

Если P имеет указатель направления `$l`, то отыскивается самое первое слева вхождение, которое принимает для двух вариантов сопоставления ρ' и ρ'' различные значения.

Если P имеет указатель направления `$r`, то отыскивается самое первое справа вхождение, которое принимает для двух вариантов сопоставления ρ' и ρ'' различные значения.

Пусть найденное вхождение является вхождением переменной V. Тогда по определению объектные выражения $\rho'(V)$ и $\rho''(V)$ не равны. Более того, можно показать, что заведомо либо $\rho'(V)$ является частью $\rho''(V)$, либо $\rho''(V)$ является частью $\rho'(V)$.

Тогда если $\rho'(V)$ является частью $\rho''(V)$, считается, что ρ' предшествует ρ''. В противном случае считается, что ρ'' предшествует ρ'.

Конечную последовательность сред ρ_1, ρ_2, $\ldots$, ρ_n будем записывать в виде $[\rho_1, \rho_2, \ldots, \rho_n]$, пустую последовательность – в виде $[\,]$.

Запись $[\rho_0] \frown [\rho_1, \ldots, \rho_n]$ будет обозначать $[\rho_0, \rho_1, \ldots, \rho_n]$.

Запись $\rho \vdash \mathcal{E} : P \Rightarrow [\rho_1, \ldots \rho_n]$ будет означать, что $\mathbb{M}(\rho, \mathcal{E}, P) = \{\rho_1, \ldots, \rho_n\}$ и при этом ρ_i предшествует ρ_j для всех $i < j$.

2.8.3 Примеры

Ниже приведены примеры образцов:

```
t.Head e.Tail
eX (eY)
eA '+' eB
$l eA '+' eB
$r eA '+' eB
```

Далее приведены примеры сопоставления с образцом:

```
{} ⊢ A () C D E : $l sX tY tZ e1 ⇒ [
          {sX = A, tY = (), tZ = C, e1 = D E} ]

{} ⊢ 1 2 3 : $l eA eB ⇒ [
          {eA = ,         eB = 1 2 3},
          {eA = 1,        eB = 2 3},
          {eA = 1 2,      eB = 3},
          {eA = 1 2 3,    eB = }  ]

{} ⊢ 1 2 3 : $r eA eB ⇒ [
          {eA = 1 2 3,    eB =       },
          {eA = 1 2,      eB = 3     },
          {eA = 1,        eB = 2 3   },
          {eA = ,         eB = 1 2 3 }  ]

{eA = 1 2} ⊢ $l 1 2 3 4 5 : eA eB ⇒ [
                  {eA = 1 2, eB = 3 4 5} ]
```

2.9 Жесткие выражения

2.9.1 Синтаксис

```
$    ЖесткоеВыражение =
$         { ЖесткийКрай } |
$         { ЖесткийКрай } e-переменная { ЖесткийКрай } |
$         { ЖесткийКрай } v-переменная { ЖесткийКрай }.
$    ЖесткийКрай = { ЖесткийТерм | ИменованноеВыражение }.
$    ЖесткийТерм =
$         СтатическийСимвол | s-переменная | t-переменная |
$         "(" ЖесткоеВыражение ")".
```

Таким образом жесткое выражение на каждом уровне скобок содержит не более чем одну e-переменную или v-переменную.

Индексы всех переменных, входящих в одно и то же жесткое выражение, должны быть попарно различны.

В дальнейшем мы будем обозначать жесткие выражения через He, а жесткие термы – через Ht.

2.9.2 Сопоставление с жестким выражением

Любое жесткое выражение He представляет собой частный случай образцового выражения. Его особенностью является то, что для любого объектного выражения $\mathcal{E}$ либо сопоставление $\mathcal{E}$ с He невозможно, либо

может быть сделано только одним способом. Таким образом, имеет место либо $\{\} \vdash \mathcal{E} \ :: \ He \Rightarrow [\,]$, либо $\{\} \vdash \mathcal{E} \ :: \ He \Rightarrow [\rho]$.

Запись $\rho \vdash \mathcal{E} \ :: \ He \Rightarrow \rho'$ будет означать, что $\{\} \vdash \mathcal{E} \ :: \ He \Rightarrow \rho''$ и $\rho' = \rho + \rho''$, т.е. что ρ' получается из ρ следующим образом. Сначала $\mathcal{E}$ сопоставляется с He в пустой среде. Т.е. прежние значения переменных игнорируются. В результате получается среда ρ'', которая содержит значения для переменных из He. После этого исходная среда ρ корректируется в соответствии с новыми значениями переменных из He, в результате чего получается новая среда ρ'.

2.9.3 Примеры

Ниже приведены примеры жестких выражений:

```
t.Head e.Tail
sX (eY) eZ (A eA)
```

Далее приведены примеры сопоставления с жесткими выражениями:

```
{sX = XXX, eA = A B C}  ⊢  X Y Z :: sY eA ⇒
                                {sX = XXX, eA = Y Z, sY = X}

{sX = XXX, eA = A B C}  ⊢  X Y Z :: eA sY ⇒
                                {sX = XXX, eA = X Y, sY = Z}
```

2.10 Тропы

2.10.1 Синтаксис

```
$    Тропа =
$         Условие | Присваивание | Поиск | Перестройка |
$         Хвост | Источник.

$    Условие =
$         Источник Хвост.
$    Присваивание =
$         Источник "::" ЖесткоеВыражение [ Хвост ].
$    Поиск =
$         Источник "$iter" Источник
$              [ "::" ЖесткоеВыражение ] [ Хвост ].
$    Перестройка =
$         Источник ":" Образец [ Хвост ].

$    Хвост =
$         ОгражденнаяТропа | ОтрицаниеУсловия |
$         Забор | Отсечение |
```

```
$          Тупик | ПраваяЧасть | Авария |
$          ПерехватАварий.

$     ОгражденнаяТропа =
$          "," Тропа.
$     ОтрицаниеУсловия =
$          "#" Источник [ Хвост ].
$     Забор =
$          "\?" Тропа.
$     Отсечение =
$          "\!" Тропа.
$     Тупик =
$          "$fail".
$     ПраваяЧасть =
$          "=" Тропа.
$     Авария =
$          "$error" Тропа.
$     ПерехватАварий =
$          "$trap" Тропа "$with" ОбразцовоеРаспутье.

$     Источник =
$          Распутье | Выбор | РезультатноеВыражение.

$     Распутье =
$          "\{" ПослТроп "}" |
$           "{" ПослТроп "}".

$     Выбор =
$          Источник ":" ОбразцовоеРаспутье.

$     ОбразцовоеРаспутье =
$          "\{" ПослПредложений "}" |
$          " {" ПослПредложений "}" |

$     ПослТроп = { Тропа ";" }.

$     ПослПредложений =
$          { Предложение ";" }.

$     Предложение = Образец [ Хвост ].
```

В дальнейшем мы будем обозначать тропы через Q, хвосты – через R, источники – через S, образцовые распутья – через $Palt$, а предложения – через Snt.

Синтаксис троп выглядит несколько запутанным. Это вызвано стремлением избавить пользователя от необходимости писать лишние разделители в тех случаях, когда в этом нет реальной необходимости.

Для этого особо выделяются два частных случая троп: "хвосты" и "источники", обладающие полезными синтаксическими свойствами. Хвосты начинаются с легко распознаваемых ключевых слов и не могут "склеиться" с предшествующими результатными выражениями и образцами, а источники не содержат запятую на нулевом уровне фигурных скобок, благодаря чему они однозначно отделяются от последующих конструкций.

2.10.2 Вычисление троп

Вычисление тропы Q производится при условии, что задана среда ρ и целое неотрицательное число m. Среда ρ содержит значения переменных, которые могут потребоваться в процессе вычисления тропы. Число m, именуемое *уровнем* тропы, характеризует степень вложенности тропы в заборы, т.е. количество заборов `"\?"`, окружающих Q, еще не закрытых отсечениями `"\!"`.

Если вычисление тропы заканчивается, то результатом вычисления является либо объектное выражение $\mathcal{E}$, либо неуспех `$fail(`k`)`, где целое неотрицательное k характеризует "степень серьезности" неуспеха, либо ошибка `$error(`$\mathcal{E}$`)`, где $\mathcal{E}$ – сообщение об ошибке.

Если тропа находится на уровне m, а результатом ее вычисления является неуспех `$fail(`k`)`, "серьезность неуспеха" всегда удовлетворяет условию $0 \le k \le m+1$. В частности, если тропа находится на уровне 0, всегда либо $k = 0$, либо $k = 1$.

В дальнейшем запись $\rho, m, \sigma \vdash Q \Rightarrow X, \sigma'$ означает, что если тропа Q находится на уровне m, то результатом ее вычисления в среде ρ является X, причем если вычисление тропы Q началось в глобальном состоянии σ, то оно завершится в глобальном состоянии σ'.

Хвосты и источники являются частными случаями троп, поэтому это же обозначение используется и применительно к вычислению хвостов и троп.

Во многих случаях смысл тропы Q может быть объяснен через смысл другой тропы Q'. А именно, чтобы вычислить тропу Q, следует вычислить тропу Q', и то, что получится, следует считать результатом вычисления тропы Q. Формально это можно выразить с помощью следующего правила вывода:

$$\frac{\rho, m, \sigma \vdash Q' \Rightarrow X, \sigma'}{\rho, m, \sigma \vdash Q \Rightarrow X, \sigma'}$$

Поскольку такие правила встречаются довольно часто, мы будем в

дальнейшем записывать их в сокращенном виде следующим образом:

$$Q \Rightarrow\Rightarrow Q'$$

2.10.3 Условия

Тропа $S\ R$ означает, что следует вычислить источник S, а затем, в случае успеха, вычислять хвост R.

$$\frac{\begin{array}{lll}\rho, 0, \sigma & \vdash & S \Rightarrow, \sigma' \\ \rho, m, \sigma' & \vdash & R \Rightarrow X, \sigma''\end{array}}{\rho, m, \sigma \quad \vdash \quad S\ R \Rightarrow X, \sigma''}$$

$$\frac{\rho, 0, \sigma \quad \vdash \quad S \Rightarrow \texttt{\$fail(}k\texttt{)}, \sigma'}{\rho, m, \sigma \quad \vdash \quad S\ R \Rightarrow \texttt{\$fail(0)}, \sigma'}$$

$$\frac{\rho, 0, \sigma \quad \vdash \quad S \Rightarrow \texttt{\$error(}\mathcal{E}\texttt{)}, \sigma'}{\rho, m, \sigma \quad \vdash \quad S\ R \Rightarrow \texttt{\$error(}\mathcal{E}\texttt{)}, \sigma'}$$

2.10.4 Присваивания

Тропа S `::` $He\ R$ позволяет вычислить источник S и связать полученные результаты с переменными в He. В результате получается новая среда, в которой и вычисляется хвост R.

Считается, что источник S находится на нулевом уровне.

Если хвост R является огражденной пустой тропой (всегда вырабатывающей пустое выражение), он может быть опущен.

$$S\ \texttt{::}\ He \Rightarrow\Rightarrow S\ \texttt{::}\ He,$$

$$\frac{\begin{array}{lll}\rho, 0, \sigma & \vdash & S \Rightarrow \mathcal{E}, \sigma' \\ \rho & \vdash & \mathcal{E} :: He \Rightarrow \rho' \\ \rho', m, \sigma' & \vdash & R \Rightarrow X, \sigma''\end{array}}{\rho, m, \sigma \quad \vdash \quad S :: He\ R \Rightarrow X, \sigma''}$$

$$\frac{\rho, 0, \sigma \quad \vdash \quad S \Rightarrow \texttt{\$fail(}k\texttt{)}, \sigma'}{\rho, m, \sigma \quad \vdash \quad S :: He\ R \Rightarrow \texttt{\$fail(0)}, \sigma'}$$

$$\frac{\rho, 0, \sigma \quad \vdash \quad S \Rightarrow \texttt{\$error(}\mathcal{E}\texttt{)}, \sigma'}{\rho, m, \sigma \quad \vdash \quad S :: He\ R \Rightarrow \texttt{\$error(}\mathcal{E}\texttt{)}, \sigma'}$$

Например, результатом выполнения тропы

```
100 :: sN, <"+" sN 1> :: sN = sN
```

является число 101.

2.10.5 Поиски

Тропа S'' `$iter` S' `::` He означает, что для переменных из He следует подобрать такие значения, что попытка вычислить R успешно завершится. При этом полученный результат и будет результатом всей конструкции.

Если жесткое выражение He – пустое, оно может быть опущено вместе с ключевым словом "`::`". Если хвост R является огражденной пустой тропой (всегда вырабатывающей пустое выражение), он может быть опущен.

S'' `$iter` S' $\Rightarrow\Rightarrow$ S'' `$iter` S' `::` `,`
S'' `$iter` $S'R$ $\Rightarrow\Rightarrow$ S'' `$iter` S' `::` R
S'' `$iter` S' `::` He $\Rightarrow\Rightarrow$ S'' `$iter` S' `::` He `,`

Источник S'' служит для вычисления начальных значений переменных из He, а источник S' служит для вычисления новых значений переменных из He на основе их предыдущих значений.

Считается, что источники S'' и S' находятся на нулевом уровне.

Подбор значений переменных делается следующим образом. Сначала для переменных вычисляются начальные значения. Для этого вычисляется значение источника S'' и полученное объектное выражение $\mathcal{E}$ сопоставляется с образцом He. Затем делается попытка вычислить R в новой среде. Если эта попытка заканчивается неуспехом `$fail(0)`, то вычисляется S' и полученное объектное выражение вновь сопоставляется с He, после чего опять делается попытка вычислить R и.т.д.

S'' `$iter` S' `::` $He\,R \Rightarrow\Rightarrow$
S'' `::` He, `\{` R`;` S' `$iter` S' `::` $He\,R$`; }`

Например, если значения переменных `eA` и `eB` еще не определены, перестройка

```
eX : $1 eA eB,
 <Writeln eA>, <Writeln eB> $fail
```

эквивалентна поиску

```
()(eX)
  $iter \{ eB : t1 e2 = (eA t1)(e2); }
    :: (eA)(eB),
  <Writeln eA>, <Writeln eB> $fail
```

2.10.6 Перестройки

Тропа S `:` P R имеет следующий смысл. Следует вычислить источник S и сопоставить полученное объектное выражение $\mathcal{E}$ с образцом

P. В результате получится, вообще говоря, несколько вариантов сопоставления. Затем следует попытаться найти самый первый из вариантов сопоставления, для которого вычисление хвоста R завершается успешно.

Если хвост R является огражденной пустой тропой (всегда вырабатывающей пустое выражение), он может быть опущен.

Для описания семантики перестройки нам понадобится следующее обозначение. Запись $\lambda, m, \sigma \vdash\vdash Q \Rightarrow X, \sigma'$ означает, что если тропа Q находится на уровне m, то результатом ее вычисления для списка сред λ является X.

$$\frac{\rho, m, \sigma \vdash Q \Rightarrow \mathcal{E}, \sigma'}{[\rho] \frown \lambda, m, \sigma \vdash\vdash Q \Rightarrow \mathcal{E}, \sigma'}$$

$$\frac{\begin{array}{l}\rho, m, \sigma \vdash Q \Rightarrow \texttt{\$fail(0)}, \sigma' \\ \lambda, m, \sigma' \vdash\vdash Q \Rightarrow X, \sigma''\end{array}}{[\rho] \frown \lambda, m, \sigma \vdash\vdash Q \Rightarrow X, \sigma''}$$

$$\frac{\rho, m, \sigma \vdash Q \Rightarrow \texttt{\$fail}(k+1), \sigma'}{[\rho] \frown \lambda, m, \sigma \vdash\vdash Q \Rightarrow \texttt{\$fail}(k+1), \sigma'}$$

$$\frac{\rho, m, \sigma \vdash Q \Rightarrow \texttt{\$error}(\mathcal{E}), \sigma'}{[\rho] \frown \lambda, m, \sigma \vdash\vdash Q \Rightarrow \texttt{\$error}(\mathcal{E}), \sigma'}$$

$$[\,], m, \sigma \vdash\vdash Q \Rightarrow \texttt{\$fail(0)}, \sigma$$

Теперь мы можем описать семантику перестройки.

$$S \ : \ P \Rightarrow\Rightarrow S \ : \ P,$$

$$\frac{\begin{array}{lll}\rho, 0, \sigma & \vdash & S \Rightarrow \mathcal{E}, \sigma' \\ \rho & \vdash & \mathcal{E} : P \Rightarrow \lambda \\ \lambda, m, \sigma' & \vdash\vdash & R \Rightarrow X, \sigma''\end{array}}{\rho, m, \sigma \quad \vdash \quad S \ : \ P\ R \Rightarrow X, \sigma''}$$

$$\frac{\rho, 0, \sigma \quad \vdash \quad S \Rightarrow \texttt{\$fail}(k), \sigma'}{\rho, m, \sigma \quad \vdash \quad S \ : \ P\ R \Rightarrow \texttt{\$fail(0)}, \sigma'}$$

$$\frac{\rho, 0, \sigma \quad \vdash \quad S \Rightarrow \texttt{\$error}(\mathcal{E}), \sigma'}{\rho, m, \sigma \quad \vdash \quad S \ : \ P\ R \Rightarrow \texttt{\$error}(\mathcal{E}), \sigma'}$$

Например, вычисление приведенной ниже тропы заканчивается неуспехом. При этом печатается цепочка литер `'CBA'`:

```
'ABC' : $r e sX e, <Print sX> $fail
```

2.10.7 Огражденные тропы

Вычисление хвоста

$$, \; Q$$

всегда дает тот же результат, что и вычисление тропы Q.

$$, \; Q \Rightarrow\Rightarrow Q$$

2.10.8 Отрицания условий

Хвост `#` $S \; R$ означает, что следует вычислить S, а затем взять "логическое отрицание" полученного результата.

Если хвост R является огражденной пустой тропой (всегда вырабатывающей пустое выражение), он может быть опущен.

$$\texttt{\#} \; S \Rightarrow\Rightarrow \; \texttt{\#} \; S \; ,$$

$$\frac{\rho, 0, \sigma \;\vdash\; S \Rightarrow \;\; , \;\; \sigma'}{\rho, m, \sigma \;\vdash\; \texttt{\#} \; S \; R \Rightarrow \texttt{\$fail(0)}, \sigma'}$$

$$\frac{\begin{array}{l}\rho, 0, \sigma \;\vdash\; S \Rightarrow \texttt{\$fail}(k), \sigma' \\ \rho, m, \sigma' \;\vdash\; R \Rightarrow X, \sigma''\end{array}}{\rho, m, \sigma \;\vdash\; \texttt{\#} \; S \; R \Rightarrow X, \sigma''}$$

$$\frac{\rho, 0, \sigma \;\vdash\; S \Rightarrow \; \texttt{\$error}(\mathcal{E}), \sigma'}{\rho, m, \sigma \;\vdash\; \texttt{\#} \; S \; R \Rightarrow \texttt{\$error}(\mathcal{E}), \sigma'}$$

2.10.9 Заборы

Хвост `\?` Q означает, что если при вычислении тропы Q возникнет неуспех с ненулевой "серьезностью", следует уменьшить "серьезность" на единицу. При этом считается, что уровень тропы Q на единицу больше, чем уровень всей конструкции.

$$\frac{\rho, m+1, \sigma \;\vdash\; Q \Rightarrow \mathcal{E}, \sigma'}{\rho, m, \sigma \;\vdash\; \texttt{\textbackslash?} \; Q \Rightarrow \mathcal{E}, \sigma'}$$

$$\frac{\rho, m+1, \sigma \;\vdash\; Q \Rightarrow \texttt{\$fail(0)}, \sigma'}{\rho, m, \sigma \;\vdash\; \texttt{\textbackslash?} \; Q \Rightarrow \texttt{\$fail(0)}, \sigma'}$$

$$\frac{\rho, m+1, \sigma \;\vdash\; Q \Rightarrow \texttt{\$fail}(k+1), \sigma'}{\rho, m, \sigma \;\vdash\; \texttt{\textbackslash?} \; Q \Rightarrow \texttt{\$fail}(k), \sigma'}$$

$$\frac{\rho, m+1, \sigma \;\vdash\; Q \Rightarrow \texttt{\$error}(\mathcal{E}), \sigma'}{\rho, m, \sigma \;\vdash\; \texttt{\textbackslash?} \; Q \Rightarrow \texttt{\$error}(\mathcal{E}), \sigma'}$$

2.10.10 Отсечения

Хвост `\!` Q означает, что если при вычислении тропы Q возникнет неуспех, нужно увеличить "серьезность" неуспеха на единицу. При этом считается, что уровень тропы Q на единицу меньше, чем уровень всей конструкции.

$$\frac{\rho, m, \sigma \quad \vdash \quad Q \Rightarrow \mathcal{E}, \sigma'}{\rho, m+1, \sigma \quad \vdash \quad \texttt{\textbackslash!}\ Q \Rightarrow \mathcal{E}, \sigma'}$$

$$\frac{\rho, m, \sigma \quad \vdash \quad Q \Rightarrow \texttt{\$fail}(k), \sigma'}{\rho, m+1, \sigma \quad \vdash \quad \texttt{\textbackslash!}\ Q \Rightarrow \texttt{\$fail}(k+1), \sigma'}$$

$$\frac{\rho, m, \sigma \quad \vdash \quad Q \Rightarrow \texttt{\$error}(\mathcal{E}), \sigma'}{\rho, m+1, \sigma \quad \vdash \quad \texttt{\textbackslash!}\ Q \Rightarrow \texttt{\$error}(\mathcal{E}), \sigma'}$$

Например, в результате вычисления тропы, приведенной ниже, будет напечатана цепочка литер `'ABD'` и выдан результат `'2'`.

```
{
  \? {
    <Print 'A'> $fail;
    <Print 'B'> \! $fail;
    <Print 'C'> = '1';
    };
  <Print 'D'> = '2';
  }
```

2.10.11 Тупики

Вычисление хвоста `$fail` всегда дает `$fail(0)`, т.е. неуспех серьезности 0.

$$\rho, m, \sigma \ \vdash \ \texttt{\$fail} \Rightarrow \texttt{\$fail(0)}, \sigma$$

2.10.12 Правые части

Хвост `=` Q означает, что перед вычислением тропы Q следует начать "новую жизнь", т.е. считать, что Q находится на нулевом уровне. Если в результате вычисления Q получится неуспех `$fail(`k`)`, то результатом хвоста `=` Q является `$fail(`$m+1$`)`, где m – уровень, на котором находится хвост `=` Q. Таким образом, вырабатывается неуспех, силы которого достаточно, чтобы пробить все заборы, которые еще не отменены с помощью отсечений.

$$\frac{\rho, 0, \sigma \quad \vdash \quad Q \Rightarrow \mathcal{E}, \sigma'}{\rho, m, \sigma \quad \vdash \quad \texttt{=}\ Q \Rightarrow \mathcal{E}, \sigma'}$$

$$\frac{\rho, 0, \sigma \quad \vdash \quad Q \Rightarrow \texttt{\$fail}(k), \sigma'}{\rho, m, \sigma \quad \vdash \quad \texttt{=}\ Q \Rightarrow \texttt{\$fail}(m+1), \sigma'}$$

$$\frac{\rho, 0, \sigma \quad \vdash \quad Q \Rightarrow \texttt{\$error}(\mathcal{E}), \sigma'}{\rho, m, \sigma \quad \vdash \quad \texttt{=}\ Q \Rightarrow \texttt{\$error}(\mathcal{E}), \sigma'}$$

2.10.13 Аварии

Хвост `$error` Q позволяет породить ошибку `$error`$(\mathcal{E})$, где объектное выражение $\mathcal{E}$ является результатом вычисления тропы Q.

$$\frac{\rho, 0, \sigma \quad \vdash \quad Q \Rightarrow \mathcal{E}, \sigma'}{\rho, m, \sigma \quad \vdash \quad \texttt{\$error}\ Q \Rightarrow \texttt{\$error}(\mathcal{E}), \sigma'}$$

$$\frac{\begin{array}{l}\rho, 0, \sigma \quad \vdash \quad Q \Rightarrow \texttt{\$fail}(k), \sigma' \\ \mathcal{F} \text{ - имя функции, в которой находится конструкция.}\end{array}}{\rho, m, \sigma \quad \vdash \quad \texttt{\$error}\ Q \Rightarrow \texttt{\$error}(\mathcal{F}\texttt{"Unexpected fail"}), \sigma'}$$

$$\frac{\rho, 0, \sigma \quad \vdash \quad Q \Rightarrow \texttt{\$error}(\mathcal{E}), \sigma'}{\rho, m, \sigma \quad \vdash \quad \texttt{\$error}\ Q \Rightarrow \texttt{\$error}(\mathcal{E}), \sigma'}$$

2.10.14 Перехваты аварий

Хвост `$trap` Q `$with` $Palt$ означает, что следует попытаться вычислить Q. Если результатом этого вычисления является `$error`$(\mathcal{E})$, то дальше вычисляется выбор

$$\mathcal{E}\ :\ Palt$$

и то, что получится, считается результатом всей конструкции.

Считается, что тропа Q находится на нулевом уровне.

$$\frac{\rho, 0, \sigma \quad \vdash \quad Q \Rightarrow \mathcal{E}, \sigma'}{\rho, m, \sigma \quad \vdash \quad \texttt{\$trap}\ Q\ \texttt{\$with}\ Palt \Rightarrow \mathcal{E}, \sigma'}$$

$$\frac{\begin{array}{l}\rho, 0, \sigma \quad \vdash \quad Q \Rightarrow \texttt{\$fail}(k), \sigma' \\ \rho, m, \sigma' \quad \vdash \quad \mathcal{F}\ \texttt{"Unexpected fail"}\ :\ Palt \Rightarrow X, \sigma'' \\ \text{где } \mathcal{F} \text{ - имя функции, в которой находится конструкция.}\end{array}}{\rho, m, \sigma \quad \vdash \quad \texttt{\$trap}\ Q\ \texttt{\$with}\ Palt \Rightarrow X, \sigma''}$$

$$\frac{\begin{array}{l}\rho, 0, \sigma \quad \vdash \quad Q \Rightarrow\ \texttt{\$error}(\mathcal{E}), \sigma' \\ \rho, m, \sigma' \quad \vdash \quad \mathcal{E}\ :\ Palt \Rightarrow X, \sigma''\end{array}}{\rho, m, \sigma \quad \vdash \quad \texttt{\$trap}\ Q\ \texttt{\$with}\ Palt \Rightarrow X, \sigma''}$$

2.10.15 Распутья

Источник \{Q_1; Q_2; ... Q_n;} означает, что следует вычислять тропы Q_1, Q_2, ..., Q_n слева направо, пока не удастся найти тропу, вычисление которой завершится успешно.

А именно, рассмотрим результат вычисления очередной тропы Q_j.

Если результат – объектное выражение $\mathcal{E}$, то $\mathcal{E}$ считается результатом всего распутья. Если результат – `$error(`$\mathcal{E}$`)`, то `$error(`$\mathcal{E}$`)` является результатом всего распутья. Если результат – `$fail(`$k+1$`)`, то результатом вычисления всего распутья является `$fail(`$k+1$`)`. И, наконец, если результат – `$fail(0)`, то этот неуспех "перехватывается", т.е. делается попытка вычислить следующую тропу. Если же следующей тропы не существует (т.е. $j = n$), то результатом вычисления всей тропы является неуспех `$fail(0)`.

Распутье вида {Q_1; Q_2; ... Q_n;} эквивалентно распутью

\{Q_1; Q_2; ...Q_n;`$error(`$\mathcal{F}$ `"Unexpected fail");}`

где $\mathcal{F}$ – имя функции, в которой находится конструкция.

{Q_1; Q_2; ... Q_n;} $\Rightarrow\Rightarrow$
\{Q_1; Q_2; ... Q_n; `$error(`$\mathcal{F}$ `"Unexpected fail");}`
где $\mathcal{F}$– имя функции, в которой находится конструкция.

$\rho, m, \sigma \vdash$ \{} $\Rightarrow$ `$fail(0)`$, \sigma$

$$\frac{\rho, m, \sigma \vdash Q_1 \Rightarrow \mathcal{E}, \sigma'}{\rho, m, \sigma \vdash \backslash\{Q_1;\ Q_2;\ \ldots\ Q_n;\} \Rightarrow \mathcal{E}, \sigma'}$$

$$\frac{\begin{array}{l}\rho, m, \sigma \vdash Q_1 \Rightarrow \texttt{\$fail(0)}, \sigma' \\ \rho, m, \sigma' \vdash \backslash\{Q_2;\ \ldots\ Q_n;\} \Rightarrow X, \sigma''\end{array}}{\rho, m, \sigma \vdash \backslash\{Q_1;\ Q_2;\ \ldots\ Q_n;\} \Rightarrow X, \sigma''}$$

$$\frac{\rho, m, \sigma \vdash Q_1 \Rightarrow \texttt{\$fail}(k+1), \sigma'}{\rho, m, \sigma \vdash \backslash\{Q_1;\ Q_2;\ \ldots\ Q_n;\} \Rightarrow \texttt{\$fail}(k+1), \sigma'}$$

$$\frac{\rho, m, \sigma \vdash Q_1 \Rightarrow \texttt{\$error}(\mathcal{E}), \sigma'}{\rho, m, \sigma \vdash \backslash\{Q_1;\ Q_2;\ \ldots\ Q_n;\} \Rightarrow \texttt{\$error}(\mathcal{E}), \sigma'}$$

2.10.16 Выборы

Вычисление источника S : \{Snt_1; ... Snt_n;} всегда дает тот же результат, что и вычисление тропы S : Ve, \{Ve : Snt_1; ... Ve : Snt_n;}, при условии, что Ve – некоторая e-переменная, которая больше нигде не используется в программе.

Источник S `:` `{`Snt_1`;` $\ldots$ Snt_n`;}` эквивалентен источнику S `:` `\{`Snt_1`;` $\ldots$ Snt_n`;` `e $error(`$\mathcal{F}$ `"Unexpected fail");}`, где $\mathcal{F}$ – имя функции, в которой находится данная конструкция.

S `:` `{`Snt_1`;` $\ldots$ Snt_n`;}` $\Rightarrow\Rightarrow$
S `:` `\{`Snt_1`;` $\ldots$ Snt_n`;` e `$error(Fname "Unexpected fail");}`
где $\mathcal{F}$ – имя функции, в которой находится конструкция.

$$\frac{\rho, 0, \sigma \quad \vdash \quad S \Rightarrow \mathcal{E}, \sigma' \qquad \rho, m, \sigma' \quad \vdash \quad \backslash\{\mathcal{E} \ : \ Snt_1; \ \ldots \ \mathcal{E} \ : \ Snt_n;\} \ \Rightarrow X, \sigma''}{\rho, m, \sigma \quad \vdash \quad S \ : \ \backslash\{Snt_1; \ \ldots \ Snt_n;\} \ \Rightarrow X, \sigma''}$$

$$\frac{\rho, 0, \sigma \quad \vdash \quad S \Rightarrow \ \texttt{\$fail}(k), \sigma'}{\rho, m, \sigma \quad \vdash \quad S \ : \ \backslash\{Snt_1; \ \ldots \ Snt_n;\} \ \Rightarrow \ \texttt{\$fail(0)}, \sigma'}$$

$$\frac{\rho, 0, \sigma \quad \vdash \quad S \Rightarrow \ \texttt{\$error}(\mathcal{E}), \sigma'}{\rho, m, \sigma \quad \vdash \quad S \ : \ \backslash\{Snt_1; \ \ldots \ Snt_n;\} \ \Rightarrow \ \texttt{\$error}(\mathcal{E}), \sigma'}$$

2.10.17 Результатные выражения как источники

Источник вида Re означает, что следует попытаться вычислить Re, и то, что получится, считать результатом вычисления источника.

$$\frac{\rho, \sigma \quad \vdash \quad Re \Rightarrow X, \sigma'}{\rho, m, \sigma \quad \vdash \quad Re \Rightarrow X, \sigma'}$$

2.11 Определения функций

```
$    ОпределениеФункции =
$         ИмяФункции ТелоФункции ";".
$    ТелоФункции =
$         ОбразцовоеРаспутье | Предложение.
```

Определение функции связывает имя функции с ее телом, т.е. конструкцией, описывающей способ ее вычисления.

Если определение функции имеет вид $\mathcal{F}$ Snt`;` , оно эквивалентно определению $\mathcal{F}$ `\{` Snt`; };` .

Пусть определение функции $\mathcal{F}$ имеет вид

$\mathcal{F}$ $Palt$;

Тогда вычисление вызова этой функции, имеющего вид <$\mathcal{F}$ Re> сводится к вычислению источника $\mathcal{E}$ `:` $Palt$, где $\mathcal{E}$ – результат вычисления результатного выражения Re. Если при этом получается объектное выражение $\mathcal{E}'$ или ошибка `$error(`$\mathcal{E}'$`)`, они и считаются результатом вычисления вызова. Если же получается неуспех `$fail(`k`)`, то дальнейшие

действия зависят от того, была ли функция $\mathcal{F}$ объявлена как откатная или безоткатная. Для откатной функции в качестве результата выдается `$fail(0)`, а для безоткатной – `$error(`$\mathcal{F}$ `"Unexpected fail")`.

$$\frac{\{\},0,\sigma \ \vdash \ \mathcal{E} \ : \ Palt \Rightarrow \mathcal{E}',\sigma'}{\sigma \qquad \vdash \ \texttt{<}\mathcal{F}\ \mathcal{E}\texttt{>} \ \Rightarrow \mathcal{E}',\sigma'}$$

$$\frac{\{\},0,\sigma \vdash \mathcal{E} \ : \ Palt \Rightarrow \ \texttt{\$fail(}k\texttt{)},\sigma' \quad \text{Функция } \mathcal{F} \text{ – откатная, т.е. объявлена как } \texttt{\$func? }\mathcal{F}\ F_{in}\ \texttt{=}\ F_{out}\texttt{;.}}{\sigma \vdash \texttt{<}\mathcal{F}\ \mathcal{E}\texttt{>} \ \Rightarrow \ \texttt{\$fail(0)},\sigma'}$$

$$\frac{\{\},0,\sigma \vdash \mathcal{E} \ : \ Palt \Rightarrow \ \texttt{\$fail(}k\texttt{)},\sigma' \quad \text{Функция } \mathcal{F} \text{ – безоткатная, т.е. объявлена как } \texttt{\$func }\mathcal{F}\ F_{in}\ \texttt{=}\ F_{out}\texttt{;.}}{\sigma \vdash \texttt{<}\mathcal{F}\ \mathcal{E}\texttt{>} \ \Rightarrow \texttt{\$error(}\mathcal{F}\ \texttt{"Unexpected fail")},\sigma'}$$

$$\frac{\{\},0,\sigma \vdash \mathcal{E} \ : \ Palt \Rightarrow \ \texttt{\$error(}\mathcal{E}'\texttt{)},\sigma'}{\sigma \ \vdash \ \texttt{<}\mathcal{F}\ \mathcal{E}\texttt{>} \ \Rightarrow \ \texttt{\$error(}\mathcal{E}'\texttt{)},\sigma'}$$

В вышеприведенных правилах вывода предполагается, что функция $\mathcal{F}$ имеет определение $\mathcal{F}\ Palt$.

2.12 Объявления

2.12.1 Объявления констант

```
$    ОбъявлениеКонстант =
$         "$const" [ ОбъКонст { "," ОбъКонст } ] ";".
$    ОбъКонст =
$         ИмяКонстанты "=" КонстантноеВыражение.
$    ИмяКонстанты = Идентификатор.
$    КонстантноеВыражение =
$         { КонстантныйТерм | ИменованноеВыражение }.
$    КонстантныйТерм =
$         СтатическийСимвол | "(" КонстантноеВыражение ")".
```

Объявления констант позволяют вводить для объектных выражений символические имена, с тем чтобы в дальнейшем использовать их вместо самих выражений. В качестве имен используются слова. Если некоторому объектному выражению $\mathcal{E}$ присвоено имя $\mathcal{N}$, то в дальнейшем конструкция `&`$\mathcal{N}$ является изображением этого выражения. Например, следующее объявление вводит три имени объектных выражений:

```
$const Lf = 10, Cr = 13, ABC = A B C;
```

после чего `&Lf` обозначает 10, `&Cr` обозначает 13, а `&ABC` обозначает `A B C`.

Объявление константы само может ссылаться на предшествующие объявления констант, ящиков, таблиц, каналов и функций. Например, после объявления

```
$const CrLf_ABC = &Cr &Lf &ABC;
```

`&CrLf_ABC` начинает обозначать выражение `13 10 A B C`.

2.12.2 Объявления ящиков, векторов, строк, таблиц и каналов

```
$    ОбъявлениеЯщиков   = "$box"     { ИмяСсылки } ";".
$    ОбъявлениеВекторов = "$vector"  { ИмяСсылки } ";".
$    ОбъявлениеСтрок    = "$string"  { ИмяСсылки } ";".
$    ОбъявлениеТаблиц   = "$table"   { ИмяСсылки } ";".
$    ОбъявлениеКаналов  = "$channel" { ИмяСсылки } ";".
$    ИмяСсылки = Идентификатор.
```

Объявления ящика, вектора, строки, таблицы или канала означают, что в момент загрузки программы следует породить объект соответствующего типа а символ-ссылку на этот объект обозначить с помощью символического имени, указанного в объявлении объекта. Таким образом, в дальнейшем это имя может использоваться для получения ссылки на объект.

Например, после объявления `$box X;` конструкция `&X` обозначает символ-ссылку на ящик. Ниже приведены примеры объявлений:

```
$box B1 B2 B3;
$vector V1 V2;
$table T1 T2;
$channel Input Output;
```

2.12.3 Объявления функций

```
$    ОбъявлениеФункции =
$         "$func"   ИмяФункции
$              ВходнойФормат "=" ВыходнойФормат ";" |
$         "$func?"  ИмяФункции
$              ВходнойФормат "=" ВыходнойФормат ";".
$    ВходнойФормат = ФорматноеВыражение.
$    ВыходнойФормат = ФорматноеВыражение.

$    ФорматноеВыражение = ЖесткоеВыражение.
```

Объявление функции вводит символическое имя для функции и должно предшествовать как определению функции, так и всем использованиям ее имени.

Определение функции описывает ограничения, которые накладываются на внешний вид всех вызовов этой функции, на вид входных образцов в ее определении и на вид результатных выражений, генерирующих результат работы функции. Эти ограничения будут описаны подробно в дальнейшем.

Входной и выходной форматы должны быть *жесткими*, т.е. на каждом уровне скобок может находиться максимум одна e-переменная или v-переменная.

Индексы переменных в форматах могут опускаться, не оказывают никакого влияния на смысл программы, и обычно используются в качестве комментариев. Следует отметить, что хотя контекстно-свободный синтаксис форматных выражений совпадает с синтаксисом жестких выражений, они все же считаются разными конструкциями языка вследствие различной трактовки индексов переменных.

Если в объявлении функции использовано ключевое слово `$func`, это означает, что функция – безоткатная, т.е. ее вычисление может завершиться либо успешной выдачей результата, либо выдачей ошибки.

Если в объявлении функции использовано ключевое слово `$func?`, это означает, что функция – откатная, т.е. ее вычисление может завершиться либо успешной выдачей результата, либо выдачей неуспеха, либо выдачей ошибки.

Например:

```
$func  Interpreter (e.Program) (e.Input) = e.Result;
$func? Attempt t.Arg = s.Result1 t.Result2 (e.Result3);
```

2.12.4 Директивы отладки

```
$    ДирективаОтладки =
$         "$trace" { ИмяФункции } ";" |
$         "$traceall" ";".
```

Директива отладки `"$trace"` означает, что в процессе выполнения программы для функций, имена которых перечислены в директиве, следует печатать отладочную информацию: имя функции и ее аргументы в момент обращения к ней, а также ее результаты в момент выхода из нее. Директива `"$traceall"` означает, что следует печатать отладочную информацию для всех последующих функций.

2.13 Контекстные ограничения

2.13.1 Устранение избыточных конструкций

В этом разделе мы описываем различные контекстные синтаксические ограничения, которым должна удовлетворять любая программа на Рефале Плюс.

Для того, чтобы не загромождать изложение, мы будем предполагать, что предварительно была выполнена нормализация программы, в результате которой все конструкции, являющиеся сокращениями для других конструкций заменены на эквивалентные им полные конструкции следующим образом.

Хвосты и жесткие образцы, опущенные в присваиваниях, поисках и перестройках, заменены на пустые образцы и пустые огражденные тропы следующим образом:

$$
\begin{array}{lcl}
S \texttt{ :: } He & \Rightarrow & S \texttt{ :: } He \texttt{ ,} \\
S' \texttt{ \$iter } S'' & \Rightarrow & S' \texttt{ \$iter } S'' \texttt{ :: ,} \\
S' \texttt{ \$iter } S'' \texttt{ :: } He & \Rightarrow & S' \texttt{ \$iter } S'' \texttt{ :: } He \texttt{ ,} \\
S' \texttt{ \$iter } S'' \, R & \Rightarrow & S' \texttt{ \$iter } S'' \texttt{ ::} \;\; R \\
S \texttt{ : } P & \Rightarrow & S \texttt{ : } P \texttt{ ,} \\
\texttt{\# } S & \Rightarrow & \texttt{\# } S \texttt{ ,}
\end{array}
$$

Хвосты, опущенные в предложениях, заменены на пустые огражденные тропы следующим образом:

$$
P \quad \Rightarrow \quad P \texttt{ ,}
$$

“Непрозрачные” фигурные скобки `"{"` заменены на прозрачные фигурные скобки `"\{"` в распутьях и выборах следующим образом:

$\texttt{\{}Snt_1\texttt{; } \ldots\ Snt_n\texttt{;\}} \Rightarrow$
    `\{`Snt_1`; ` $\ldots$ Snt_n`;`
        `$error(`$\mathcal{F}$ `"Unexpected fail");}`

S `: {`Snt_1`; ` $\ldots$ Snt_n`;} ⇒`
    S `: \{`Snt_1`; ` $\ldots$ Snt_n`;`
        `e $error(`$\mathcal{F}$ `"Unexpected fail");}`

где $\mathcal{F}$ – имя функции, в которой находится конструкция.

“Непрозрачные” фигурные скобки `"{"` заменены на прозрачные фигурные скобки `"\{"` в определениях функций следующим образом:

$\mathcal{F}$ `{`Snt_1`; ` $\ldots$ Snt_n`;} ⇒`
    $\mathcal{F}$ `\{`Snt_1`; ` $\ldots$ Snt_n`;`
        F_{in} `$error(`$\mathcal{F}$ `"Unexpected fail");}`

где F_{in} – входной формат функции из объявления функции $\mathcal{F}$ (в котором опущены индексы переменных).

Тела функций, являющиеся предложениями, заменены на соответствующие образцовые распутья:

$\mathcal{F}\ Snt\texttt{;}\ \Rightarrow\ \mathcal{F}\ \texttt{\textbackslash\{}\ Snt\texttt{;}\ \texttt{\}};$

2.13.2 Ограничения налагаемые объявлениями функций

Объявления функций имеют вид

`$func` $\mathcal{F}\ F_{in}$ `=` F_{out}`;`
`$func?` $\mathcal{F}\ F_{in}$ `=` F_{out}`;`

где F_{in} – входной формат функции, т.е. форматное выражение, задающее структуру аргумента функции, а F_{out} – выходной формат функции, т.е. форматное выражение, задающее структуру результата функции. Как уже говорилось выше, индексы переменных в форматах не играют никакой роли, и поэтому мы будем в дальнейшем, без ограничения общности, предполагать, что они отсутствуют.

Определения функций должны удовлетворять ограничениям, налагаемым входными и выходными форматами функций. Чтобы описать эти ограничения, мы введем на множестве форматов отношение частичного порядка.

Пусть F' и F'' два формата. Запись $F'' \supseteq F'$ будет означать, что формат F' является (частным) случаем формата F''. Отношение $\supseteq$ определяется следующими правилами.

- $F \supseteq F$.
- Если $F'_1 \supseteq F_1$ и $F'_2 \supseteq F_2$, то $F'_1\ F'_2 \supseteq F_1\ F_2$.
- Если $F_1 \supseteq F_2$, то `(`F_1`)` $\supseteq$ `(`F_2`)`.
- `e` $\supseteq F$.
- Если F не имеет вида `e e ... e`, то `v` $\supseteq F$.
- `t` $\supseteq \mathcal{S}$, для любого символа $\mathcal{S}$.
- `t` $\supseteq$ `s`.
- `t` $\supseteq$ `(`F`)`.
- `s` $\supseteq \mathcal{S}$, для любого символа $\mathcal{S}$.

Если в программе имеется результатное выражение Re, то его форматом считается форматное выражение F, которое получается из Re следующим образом.

- У всех переменных, входящих в Re, отбрасываются индексы.

- Вызовы всех функций, появляющиеся в Re, заменяются на выходные форматы этих функций. А именно, если в Re имеется вызов функции вида `<`$\mathcal{F}$ Re'`>`, а объявление функции $\mathcal{F}$ имеет вид `$func` $\mathcal{F}$ F_{in} `=` F_{out}`;` либо `$func?` $\mathcal{F}$ F_{in} `=` F_{out}`;`, то `<`$\mathcal{F}$ Re'`>` заменяется на F_{out}.

Если в программе имеется образец P, то его форматом считается форматное выражение F, которое получается из P следующим образом.

- У всех переменных, входящих в P, отбрасываются индексы.

- Если P имеет указатель направления сопоставления, то этот указатель отбрасывается.

Если в программе имеется жесткое выражение He, то его форматом считается форматное выражение F, которое получается из He в результате отбрасывания индексов у всех переменных, входящих в He.

В дальнейшем $\mathbb{F}[Re]$ будет обозначать формат результатного выражения Re, $\mathbb{F}[P]$ – формат образца P, а $\mathbb{F}[He]$ – формат жесткого выражения He.

Следует подчеркнуть, что формат результатного выражения Re зависит не только от внешнего вида выражения Re, но и от выходных форматов тех функций, вызовы которых содержатся в Re. Тем не менее, для каждой конкретной программы $\mathbb{F}[Re]$ определен однозначно.

Теперь мы можем описать ограничения, которым должны удовлетворять определения функций. Эти ограничения касаются вида вызовов функций, вида входных образцов функций, и вида результатов, вырабатываемых тропами.

Пусть объявление функции $\mathcal{F}$ имеет входной формат F_{in}, выходной формат F_{out}, а в ее определении

$$\mathcal{F}\ Palt$$

образцовое распутье $Palt$ имеет вид `\{`P_1 R_1`;` $\ldots$ P_n R_n`;}`.

Тогда должны быть выполнены следующие условия.

- Входные образцы функции $P_1, \ldots, P_n$ должны удовлетворять следующему ограничению: $F_{in} \supseteq \mathbb{F}[P_j]$.

- Все вызовы функции $\mathcal{F}$ во всех определениях функций должны удовлетворять следующему ограничению.

  Пусть вызов этой функции имеет вид `<`$\mathcal{F}$ Re`>`. Тогда должно выполнятся условие $F_{in} \supseteq \mathbb{F}[Re]$.

Чтобы описать ограничения, налагаемые на результаты, вырабатываемые тропами, введем следующие обозначения.

Тот факт, что результаты, вырабатываемые тропой Q, удовлетворяют формату F, мы будем записывать в виде $F \vdash Q$. Поскольку хвосты, источники и результатные выражения являются частными случаями троп, то же самое обозначение используется и для них.

Точно так же, тот факт, что результаты, вырабатываемые образцовым распутьем $Palt$, удовлетворяют формату F, мы будем записывать в виде $F \vdash Palt$.

Теперь мы можем следующим образом описать ограничения, накладываемые на определения функций выходными форматами функций.

Если функция $\mathcal{F}$ имеет определение $\mathcal{F}\ Palt$ и выходной формат F_{out}, то должно быть выполнено $F_{out} \vdash Palt$.

Отношения $F \vdash Q$ и $F \vdash Palt$ определяются с помощью следующих правил.

$$\frac{\mathbb{F}[\,] \vdash S \qquad F \vdash R}{F \vdash S\ R} \qquad \frac{\mathbb{F}[He] \vdash S \qquad F \vdash R}{F \vdash S\ \texttt{::}\ He\ R}$$

$$\frac{\mathbb{F}[He] \vdash S'' \qquad \mathbb{F}[He] \vdash S' \qquad F \vdash R}{F \vdash S''\ \texttt{\$iter}\ S'\ \texttt{::}\ He\ R}$$

$$\frac{F \vdash R}{F \vdash S\ \texttt{:}\ P\ R} \qquad \frac{F \vdash Q}{F \vdash \texttt{,}\ Q} \qquad \frac{\mathbb{F}[\,] \vdash S \qquad F \vdash R}{F \vdash \texttt{\#}\ S\ R}$$

$$\frac{F \vdash Q}{F \vdash \texttt{\textbackslash?}\ Q} \qquad \frac{F \vdash Q}{F \vdash \texttt{\textbackslash!}\ Q} \qquad F \vdash \texttt{\$fail}$$

$$\frac{F \vdash Q}{F \vdash \texttt{=}\ Q} \qquad F \vdash \texttt{\$error}\ Q$$

$$\frac{F \vdash Q \qquad F \vdash Palt}{F \vdash \texttt{\$trap}\ Q\ \texttt{\$with}\ Palt}$$

$$\frac{F \vdash Q_j \quad \text{для всех } j = 1, \ldots, n}{F \vdash \texttt{\textbackslash\{}Q_1\texttt{;}\ \ldots\ Q_n\texttt{;\}}}$$

$$\frac{F \vdash Palt}{F \vdash S\ \texttt{:}\ Palt} \qquad \frac{F \supseteq \mathbb{F}[Re]}{F \vdash Re}$$

$$\frac{F \vdash R_j \quad \text{для всех } j = 1, \ldots, n}{F \vdash \texttt{\textbackslash\{}P_1\ R_1\texttt{;}\ \ldots\ P_n\ R_n\texttt{;\}}}$$

2.13.3 Ограничения на использование ссылок на функции

Если в образцовом или результатном выражении появляется ссылка на функцию $\mathcal{F}$ вида `&`$\mathcal{F}$, то эта функция должна быть объявлена как `$func` $\mathcal{F}$ `e = e` или `$func?` $\mathcal{F}$ `e = e`.

2.13.4 Ограничения на использование переменных

Переменные, появляющиеся в определении функции, должны удовлетворять определенным ограничениям.

- Если переменная используется в результатном выражении, она должна быть к этому моменту определена. Определением переменной является ее появление в образце, если к этому моменту она еще не была определена.

- Если в каком-то месте являются определенными несколько различных переменных, их индексы должны быть попарно различны.

Сформулируем эти требования более точно. Для этого введем следующие обозначения.

$\mathbb{V}[X]$ обозначает множество переменных, входящих в конструкцию X.

`{}` обозначает пустое множество.

$\varphi' \cup \varphi''$ обозначает объединение множеств φ' и φ''.

$\varphi' \subseteq \varphi''$ означает, что все элементы множества φ' входят в φ''

$\varphi' \nabla \varphi''$ обозначает пополнение множества переменных φ' переменными из φ''. А именно, $\varphi' \nabla \varphi''$ содержит все переменные из φ'', а также все переменные из φ', индексы которых отличаются от индексов всех переменных из φ''. Например, `{sX, sY}` ∇ `{eY, eZ} = {sX, eY, eZ}`.

Запись $\varphi \vdash Q$ означает, что все переменные из φ имеют разные индексы, и все переменные, значения которых могут понадобиться для вычисления тропы Q, входят в φ. Поскольку хвосты, источники и результатные выражения являются частными случаями троп, то же самое обозначение используется и для них.

Аналогично, запись $\varphi \vdash Palt$ означает, что все переменные из φ имеют разные индексы, и все переменные, значения которых могут понадобиться для вычисления образцового распутья $Palt$, входят в φ.

Теперь мы можем описать ограничения, налагаемые на использование переменных в определениях функций, следующим образом.

Пусть функция $\mathcal{F}$ имеет определение $\mathcal{F}\ Palt$. Тогда должно выполняться `{}` $\vdash Palt$.

Отношения $\varphi \vdash Q$ и $\varphi \vdash Palt$ определяются с помощью следующих

правил.

$$\frac{\varphi \ \vdash \ S \qquad \varphi \ \vdash \ R}{\varphi \ \vdash \ S\ R} \qquad \frac{\varphi \ \vdash \ S \qquad \varphi \ \nabla \ \mathbb{V}[He] \vdash R}{\varphi \ \vdash \ S \ \texttt{::}\ He\ R}$$

$$\frac{\varphi \ \vdash \ S'' \qquad \varphi \ \nabla \ \mathbb{V}[He] \vdash S' \qquad \varphi \ \nabla \ \mathbb{V}[He] \vdash R}{\varphi \ \vdash \ S''\ \texttt{\$iter}\ S' \ \texttt{::}\ He\ R} \qquad \frac{\varphi \ \vdash \ S \qquad \varphi \ \cup \ \mathbb{V}[p] \vdash R}{\varphi \ \vdash \ S \ \texttt{:}\ P\ R}$$

$$\frac{\varphi \ \vdash \ Q}{\varphi \ \vdash \ \texttt{,}\ Q} \qquad \frac{\varphi \ \vdash \ S \qquad \varphi \ \vdash \ R}{\varphi \ \vdash \ \texttt{\#}\ S\ R} \qquad \frac{\varphi \ \vdash \ Q}{\varphi \ \vdash \ \texttt{\textbackslash?}\ Q} \qquad \frac{\varphi \ \vdash \ Q}{\varphi \ \vdash \ \texttt{\textbackslash!}\ Q}$$

$$\varphi \vdash \texttt{\$fail} \qquad \frac{\varphi \ \vdash \ Q}{\varphi \ \vdash \ \texttt{=}\ Q} \qquad \frac{\varphi \ \vdash \ Q}{\varphi \ \vdash \ \texttt{\$error}\ Q}$$

$$\frac{\varphi \ \vdash \ Q \qquad \varphi \ \vdash \ Palt}{\varphi \ \vdash \ \texttt{\$trap}\ Q\ \texttt{\$with}\ Palt}$$

$$\frac{\varphi \ \vdash \ Q_j \qquad \text{для всех } j = 1, \ldots, n}{\varphi \ \vdash \ \texttt{\textbackslash\{}Q_1\texttt{;}\ \ldots\ Q_n\texttt{;\}}} \qquad \frac{\varphi \ \vdash \ S \qquad \varphi \ \vdash \ Palt}{\varphi \ \vdash \ S \ \texttt{:}\ Palt}$$

$$\frac{\text{Все переменные из } \varphi \text{ имеют разные индексы} \qquad \mathbb{V}[Re] \subseteq \varphi}{\varphi \vdash Re}$$

$$\frac{\varphi \ \cup \ \mathbb{V}[P_j] \ \vdash R_j \qquad \text{для всех } j = 1, \ldots, n}{\varphi \ \vdash \ \texttt{\textbackslash\{}P_1\ R_1\texttt{;}\ \ldots\ P_n\ R_n\texttt{;\}}}$$

2.13.5 Ограничения на использование отсечений

Каждой тропе, входящей в определение функции, может быть поставлено в соответствие целое неотрицательное число k, именуемое ее уровнем. Если двигаться вдоль тропы, то прохождение через `"\?"` увеличивает текущий уровень на 1, а прохождение через `"\!"` уменьшает уровень на 1. Таким образом, каждому отсечению `"\!"` должен однозначно соответствовать некоторый “парный” забор `"\?"`.

Сформулируем это требование более точно. Для этого введем следующие обозначения.

Пусть k – целое неотрицательное число, а Q – тропа. Запись $k \vdash Q$ означает, что тропе Q может быть приписан уровень k. Поскольку хвосты, источники и результатные выражения являются частными случаями троп, то же самое обозначение используется и для них.

Аналогично, если $Palt$ – образцовое распутье, то запись $k \vdash Palt$ означает, что образцовому распутью $Palt$ может быть приписан уровень k.

Теперь мы можем описать ограничения, налагаемые на использование отсечений в определениях функций, следующим образом.

Пусть функция $\mathcal{F}$ имеет определение $\mathcal{F}\ Palt$. Тогда должно выполняться $0 \vdash Palt$.

Отношения $k \vdash Q$ и $k \vdash Palt$ определяются с помощью следующих правил.

$$\frac{0 \vdash S \qquad k \vdash R}{k \vdash S\ R} \qquad \frac{0 \vdash S \qquad k \vdash R}{k \vdash S\ \texttt{::}\ He\ R}$$

$$\frac{0 \vdash S'' \qquad 0 \vdash S' \qquad k \vdash R}{k \vdash S''\ \texttt{\$iter}\ S'\ \texttt{::}\ He\ R} \qquad \frac{0 \vdash S \qquad k \vdash R}{k \vdash S\ \texttt{:}\ P\ R}$$

$$\frac{k \vdash Q}{k \vdash \texttt{,}\ Q} \qquad \frac{0 \vdash S \qquad k \vdash R}{k \vdash \texttt{\#}\ S\ R} \qquad \frac{k+1 \vdash Q}{k \vdash \texttt{\textbackslash?}\ Q}$$

$$\frac{k \vdash Q}{k+1 \vdash \texttt{\textbackslash!}\ Q} \qquad k \vdash \texttt{\$fail} \qquad \frac{0 \vdash Q}{k \vdash \texttt{=}\ Q}$$

$$\frac{0 \vdash Q}{k \vdash \texttt{\$error}\ Q} \qquad \frac{0 \vdash Q \qquad k \vdash Palt}{k \vdash \texttt{\$trap}\ Q\ \texttt{\$with}\ Palt}$$

$$\frac{k \vdash Q_j \quad \text{для всех } j = 1, \ldots, n}{k \vdash \texttt{\textbackslash\{}Q_1\texttt{;}\ \ldots\ Q_n\texttt{;\}}}$$

$$\frac{0 \vdash S \qquad k \vdash Palt}{k \vdash S\ \texttt{:}\ Palt} \qquad k \vdash Re$$

$$\frac{k \vdash R_j \quad \text{для всех } j = 1, \ldots, n}{k \vdash \texttt{\textbackslash\{}P_1\ R_1\texttt{;}\ \ldots\ P_n\ R_n\texttt{;\}}}$$

2.14 Модули

Каждая программа на Рефале Плюс состоит из одного или нескольких *модулей*. Каждый модуль состоит из двух частей: *интерфейса модуля* и *реализации модуля*.

Интерфейс модуля содержит те части модуля, которые доступны из других модулей, в то время как реализация модуля содержит те части модуля, которые недоступны из других модулей.

Каждый модуль `MMMM` занимает два файла. А именно, интерфейс модуля хранится в файле `MMMM.rfi`, а реализация – в файле `MMMM.rf`.

```
$    ИнтерфейсМодуля =
$         { Объявление }.
$    Объявление =
$         ОбъявлениеКонстант | ОбъявлениеЯщиков |
$         ОбъявлениеВекторов | ОбъявлениеСтрок |
$         ОбъявлениеТаблиц | ОбъявлениеКаналов |
$         ОбъявлениеФункции.

$    РеализацияМодуля =
$         { Импорт } { ДирективаРеализации }.
$    ДирективаРеализации =
$         Объявление |
$         ДирективаОтладки |
$         ОпределениеФункции.

$    Импорт = "$use" { ИмяМодуля } ";".
$    ИмяМодуля = Идентификатор.
```

Если внутри реализации модуля `XXXX` требуется получить доступ к именам, объявленным в интерфейсе модуля `YYYY`, следует поместить внутри реализации модуля `XXXX` директиву `$use YYYY` следующим образом:

```
// Интерфейс модуля XXXX.
......
```

Файл `XXXX.rfi`

```
// Реализация модуля XXXX.
$use ... YYYY ... ;
// С этого места доступны имена,
// объявленные в YYYY.rfi
......
```

Файл `XXXX.rf`

```
// Интерфейс модуля YYYY.
......
```

Файл `YYYY.rfi`

```
// Реализация модуля YYYY.
......
```

Файл `YYYY.rf`

Некоторые операционные системы игнорируют различие между строчными и прописными буквами в именах файлов. Например, `StdIO.rfi`, `StdIo.rfi`, `Stdio.rfi` рассматриваются как варианты имени одного и того же файла. Поэтому, не рекомендуется использовать в одной и той же программе имена модулей, отличающиеся только размером букв.

2.15 Исполнение программы

Программа на Рефале Плюс может состоять из нескольких модулей, один из которых обязательно должен экспортировать функцию `Main`. Эта функция называется главной функцией программы. Главная функция всегда должна иметь объявление

```
$func Main = e;
```

Если же из какого-то модуля экспортируется функция с именем `Main` имеющая другое объявление, это считается ошибкой.

Исполнение Рефал-программы заключается в вычислении вызова функции `Main` с пустым аргументом

```
<Main >
```

Модуль, содержащий определение главной функции, может не иметь интерфейсной части, поскольку если компилятор не находит файл, содержащий интерфейс модуля, он считает, что интерфейс модуля состоит из единственного объявления функции

```
$func Main = e;
```

Глава 3

Библиотека функций

3.1 Использование библиотечных функций

Составной частью системы Рефал Плюс является библиотека функций, состоящая из нескольких модулей.

Если в каком-то модуле пользователя используется библиотечная функция, входящая в библиотечный модуль `MMMM`, в начале модуля пользователя следует поместить директиву

```
$use MMMM;
```

В результате этого в модуль пользователя будут импортированы объявления всех функций, входящих в библиотечный модуль `MMMM`.

В настоящее время библиотека функций состоит из следующих модулей:

`Access`	–	функции прямого доступа к частям выражений.
`Apply`	–	функция для вызова функций, переданных через параметры.
`Arithm`	–	функции целочисленной арифметики.
`Bit`	–	операции с цепочками битов.
`Box`	–	функции для работы с ящиками.
`Class`	–	предикаты для проверки принадлежности символов к различным классам символов.
`Compare`	–	предикаты для сравнения выражений.
`Convert`	–	функции для выполнения преобразований между различными типами данных.
`Dos`	–	функции для связи с операционной системой.
`StdIO`	–	объявления стандартных каналов ввода-вывода и функции для стандартного ввода-вывода.
`String`	–	функции для работы со строками.
`Table`	–	функции для работы с таблицами.
`Vector`	–	функции для работы с векторами.

В дальнейшем в библиотеку могут быть добавлены и другие модули.

3.2 Access: прямой доступ к выражениям

```
$func  Length  e.Exp = s.ExpLen;
$func? Left    s.Left  s.Len e.Exp = e.SubExp;
$func? Right   s.Right s.Len e.Exp = e.SubExp;
$func? Middle  s.Left  s.Right e.Exp = e.SubExp;
$func? L       s.Left  e.Exp = t.SubExp;
$func? R       s.Right e.Exp = t.SubExp;
```

Эти функции используются для прямого доступа к частям выражений. Их аргументы `s.Left`, `s.Right` и `s.Len` должны быть неотрицательными целыми числами. `e.Exp` может быть произвольным объектным выражением.

Если `s.Left`, `s.Right` или `s.Len` не являются неотрицательными целыми числами, результатом функций является ошибка
`$error(`$\mathcal{F}$ `"Invalid argument")`, где $\mathcal{F}$ – имя функции.

`Length` выдает длину выражения `e.Exp`, измеренную в термах. А именно, если объектное выражение $\mathcal{E}$ имеет вид $\mathcal{T}_1\ \mathcal{T}_2\ \ldots\ \mathcal{T}_n$, то n – его длина.

Например:

```
<Length >                =>   0
<Length A B C>           =>   3
<Length (A B) C (D E)>   =>   3
```

`Left` отбрасывает от `e.Exp` первые `s.Left` термов, а затем от того что останется берет первые `s.Len` термов и выдает полученное выражение в качестве результата.

`Right` отбрасывает от `e.Exp` последние `s.Right` термов, а затем от того что останется берет последние `s.Len` термов и выдает полученное выражение в качестве результата.

`Middle` отбрасывает от `e.Exp` первые `s.Left` и последние `s.Right` термов, и выдает полученное выражение в качестве результата.

`L` отбрасывает от `e.Exp` первые `s.Left` термов, и выдает первый терм полученного выражения в качестве результата.

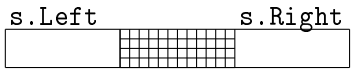
`R` отбрасывает от `e.Exp` последние `s.Right` термов, и выдает последний терм полученного выражения в качестве результата.

Если `e.Exp` имеет недостаточно большую длину, чтобы можно было выполнить требуемую операцию, результатом работы любой из вышеперечисленных функций является `$fail(0)`.

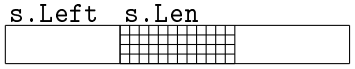
Например:

```
<Middle 2 3    A B C D E F>   =>   C
<Middle 2 3    A B C D>       =>   $fail(0)
<Middle 0 0    A B C>         =>   A B C
<Left   2 3    A B C D E F>   =>   C D E
<Left   2 3    A B C D>       =>   $fail(0)
<Left   0 0    A B C>         =>
<Right  2 3    A B C D E F>   =>   B C D
<Right  2 3    A B C D>       =>   $fail(0)
<Right  0 0    A B C>         =>
<L      2      A B C D E F>   =>   C
<L      2      A B>           =>   $fail(0)
<R      2      A B C D E F>   =>   D
<R      2      A B>           =>   $fail(0)
```

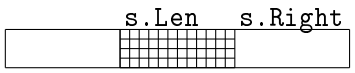
Операции `Middle`, `Left` и `Right` можно изобразить в виде следующей схемы:



`<Middle s.Left s.Right e.Exp>`



`<Left s.Left s.Len e.Exp>`



`<Right s.Right s.Len e.Exp>`

3.3 Apply: вызов функций переданных через параметры

```
$func? Apply s.Name e.Exp = e.Exp;
```

`Apply` применяет функцию, на которую указывает ссылка `s.Name` к выражению `e.Exp` и выдает результат этого вычисления.

3.4 Arithm: целочисленная арифметика

```
$func Add     s.Int1 s.Int2 = s.Int;
$func Sub     s.Int1 s.Int2 = s.Int;
$func Mult    s.Int1 s.Int2 = s.Int;
$func DivRem  s.Int1 s.Int2 = s.Quo s.Rem;
$func Div     s.Int1 s.Int2 = s.Quo;
$func Rem     s.Int1 s.Int2 = s.Rem;
$func Gcd     s.Int1 s.Int2 = s.Gcd;
```

Эти функции выполняют арифметические операции над целыми со знаком неограниченной разрядности. Каждый из аргументов арифметических функций должен представлять собой один символ-число.

Если хотя бы один из аргументов функций арифметики не является символом-числом, результатом является ошибка
`$error(`$\mathcal{F}$`"Invalid argument")`, где $\mathcal{F}$ – имя функции.

Если оба аргумента - числа, эти функции выдают следующие результаты.

`Add` выдает сумму двух аргументов, `Sub` – разность первого и второго аргументов, `Mult` – произведение двух аргументов, `Div` – результат деления нацело первого аргумента на второй, `Rem` – остаток от деления нацело первого аргумента на второй, `DivRem` – результат деления нацело и остаток от деления нацело первого аргумента на второй, `Gcd` – наибольший общий делитель двух аргументов.

Если результат одной из функций `Add`, `Sub` или `Mult` больше, чем допускается реализацией, результатом является ошибка
`$error(`$\mathcal{F}$`"Size limit exceeded")`, где $\mathcal{F}$ – имя функции.

Например:

```
<Add 3 5>       =>   8
<Add 3 -5>      =>   -2
<Sub 3 -5>      =>   8
<Mult -2 3>     =>   -6
<Div 5 2>       =>   2
<Rem 5 2>       =>   1
<DivRem 5 2>    =>   2 1
<Div 6 2>       =>   3
<Rem 6 2>       =>   0
<DivRem 6 2>    =>   3 0
```

Знаки частного и остатка определяются следующим образом. Если знаки делимого и делителя совпадают – частное положительно, если различны – отрицательно. Знак остатка совпадает со знаком делимого. Таким образом, всегда выполняется соотношение

$$\text{делимое} = (\text{частное} * \text{делитель}) + \text{остаток}$$

Например:

```
<Div 5 3>      =>   1
<Rem 5 3>      =>   2
<Div 5 -3>     =>   -1
<Rem 5 -3>     =>   2
<Div -5 3>     =>   -1
<Rem -5 3>     =>   -2
<Div -5 -3>    =>   1
<Rem -5 -3>    =>   -2
```

Попытка делить на ноль приводит к выдаче ошибки
`$error(`$\mathcal{F}$ `"Divide by zero")`, где $\mathcal{F}$ – имя функции.

Например:

```
<Div 5 0>      =>   $error(DIV "Divide by zero")
<Rem 5 0>      =>   $error(REM "Divide by zero")
<DivRem 5 0>   =>   $error(DIV-REM "Divide by zero")
```

Функция `Gcd`, если ее оба аргумента не равны нулю одновременно, выдает положительное целое число, которое является наибольшим общим делителем аргументов. Если же оба аргумента равны нулю, результатом является ошибка `$error(Gcd "Zero arguments")`. Например:

```
<Gcd 6 15>    =>   3
<Gcd -6 15>   =>   3
<Gcd 15 1>    =>   1
<Gcd 15 0>    =>   15
<Gcd 0 0>     =>   $error(Gcd "Zero arguments")
```

3.5 `Bit`: Операции с цепочками битов

Модуль `Bit` содержит функции, которые работают с цепочками двоичных цифр, представленными целыми числами.

Каждое целое число изображает цепочку двоичных цифр бесконечную влево, которая получится, если записать это число как двоичное число бесконечно большой разрядности, представленное в дополнительном коде. При этом в случае положительного числа эта цепочка содержит конечное число единиц, а в случае отрицательного – конечное число нулей. Например:

```
+3   ...000011
+2   ...000010
+1   ...000001
+0   ...000000
-1   ...111111
-2   ...111110
-3   ...111101
```

Разряды нумеруются справа налево, начиная с нуля.

```
$func  BitOr   s.Int1 s.Int2 = s.Int;
$func  BitAnd  s.Int1 s.Int2 = s.Int;
$func  BitXor  s.Int1 s.Int2 = s.Int;
```

`BitOr` выдает поразрядное “или” двух аргументов.

`BitAnd` выдает поразрядное “и” двух аргументов.

`BitXor` выдает поразрядное исключающее “или” двух аргументов.

```
$func  BitNot  s.Int = s.Int;
```

`BitNot` выдает поразрядное логическое отрицание аргумента.

```
$func  BitLeft   s.Int s.Shift = s.Int;
$func  BitRight  s.Int s.Shift = s.Int;
```

`BitLeft` выдает результат логического сдвига `s.Int` на `s.Shift` разрядов влево, если `s.Shift` неотрицательно, или на минус `s.Shift` разрядов вправо, если `s.Shift` отрицательно.

`BitRight` выдает результат логического сдвига `s.Int` на `s.Shift` разрядов вправо, если `s.Shift` неотрицательно, или на минус `s.Shift` разрядов влево, если `s.Shift` отрицательно.

```
$func? BitTest   s.Int s.Pos = ;
```

`BitTest` выдает неуспех, если разряд `s.Pos` числа `s.Int` равен нулю, в противном случае результатом является пустое выражение.

```
$func  BitSet    s.Int s.Pos = s.Int;
$func  BitClear  s.Int s.Pos = s.Int;
```

`BitSet` устанавливает разряд `s.Pos` числа `s.Int` в `1` и выдает полученное число в качестве результата.

`BitClear` устанавливает разряд `s.Pos` числа `s.Int` в `0` и выдает полученное число в качестве результата.

```
$func  BitLength  s.Int = s.Len;
```

`BitLength` выдает "длину" числа `s.Int`. А именно, если `s.Int` неотрицательно, выдается номер позиции самого правого нуля, слева от которого нет ни одной единицы. Если же число отрицательно, выдается номер самой правой единицы, слева от которой нет ни одного нуля.

Например:

```
<BitLength  3>     ==>  2
<BitLength  2>     ==>  2
<BitLength  1>     ==>  1
<BitLength  0>     ==>  0
<BitLength -1>     ==>  0
<BitLength -2>     ==>  1
<BitLength -3>     ==>  2
```

3.6 Box: работа с ящиками

```
$func Box     e.Exp = s.Box;
$func Get     s.Box = e.Exp;
$func Store   s.Box e.Exp = ;
```

`Box` создает новый ящик, помещает в него выражение `e.Exp` и выдает ссылку на этот ящик.

`Get` выдает содержимое ящика, на который ссылается `s.Box`.

`Store` заносит в ящик, на который ссылается `s.Box`, новое содержимое `e.Expr`.

3.7 `Class`: предикаты для распознавания классов символов

```
$func? IsBox     e.Exp = ;
$func? IsChannel e.Exp = ;
$func? IsChar    e.Exp = ;
$func? IsDigit   e.Exp = ;
$func? IsFunc    e.Exp = ;
$func? IsInt     e.Exp = ;
$func? IsLetter  e.Exp = ;
$func? IsString  e.Exp = ;
$func? IsTable   e.Exp = ;
$func? IsVector  e.Exp = ;
$func? IsWord    e.Exp = ;
```

Эти функции служат для проверки, что `e.Exp` является символом, принадлежащим к определенному множеству символов.

Если `e.Exp` – не символ, результатом является `$fail(0)`.

Если `e.Exp` – символ, выполняется проверка, что этот символ принадлежащит к соответствующему множеству. Если принадлежит – результатом является пустое выражение, если же не принадлежит – результатом является `$fail(0)`.

Соответствие между функциями-предикатами и множествами символов следующее:

`IsBox`	–	ссылки на ящики.
`IsChannel`	–	ссылки на каналы.
`IsChar`	–	символы-литеры.
`IsDigit`	–	символы-литеры, которые соответствуют десятичным цифрам.
`IsFunc`	–	ссылки на функции.
`IsInt`	–	целые числа.
`IsLetter`	–	символы-литеры, которые соответствуют латинским прописным и строчным буквам.
`IsString`	–	ссылки на строки.
`IsTable`	–	ссылки на таблицы.
`IsVector`	–	ссылки на векторы.
`IsWord`	–	слова.

3.8 Compare: сравнение выражений

```
$func? Eq  (e.Exp1)(e.Exp2) = ;
$func? Ne (e.Exp1)(e.Exp2) = ;
$func? Ge (e.Exp1)(e.Exp2) = ;
$func? Gt  (e.Exp1)(e.Exp2) = ;
$func? Le (e.Exp1)(e.Exp2) = ;
$func? Lt  (e.Exp1)(e.Exp2) = ;
```

Эти функции сравнивают два выражения `e.Exp1` и `e.Exp2` и проверяют, выполнено ли соответствующее отношение между ними: `Eq` – равно, `Ne` – не равно, `Ge` – больше или равно, `Gt` – больше чем, `Le` – меньше или равно, `Lt` – меньше чем.

Если условие выполнено, результат работы – пусто, иначе – `$fail(0)`.

```
$func Compare (e.Exp1)(e.Exp2)  = s.Res;   // '<', '>', '='
```

`Compare` сравнивает два выражения `e.Exp1` и `e.Exp2`. Результатом является `'<'`, если `e.Exp1` меньше чем `e.Exp2`, `'>'` если `e.Exp1` больше чем `e.Exp2`, и `'='`, если `e.Exp1` равно `e.Exp2`.

Объектные выражения сравниваются в соответствии со следующим отношением линейного порядка $<$.

Для любых двух выражений $\mathcal{E}'$ и $\mathcal{E}''$ всегда либо $\mathcal{E}' < \mathcal{E}''$, либо $\mathcal{E}' = \mathcal{E}''$, либо $\mathcal{E}'' < \mathcal{E}'$.

Два выражения $\mathcal{E}' = \mathcal{T}'_1 \ \ldots \ \mathcal{T}'_m$ и $\mathcal{E}'' = \mathcal{T}''_1 \ \ldots \ \mathcal{T}''_n$ сравниваются *лексикографически*, т.е. составляющие их термы сравниваются попарно слева направо, пока не будет найдена пара неравных термов $\mathcal{T}'_k$ и $\mathcal{T}''_k$. Если при этом $\mathcal{T}'_k < \mathcal{T}''_k$, то считается, что $\mathcal{E}' < \mathcal{E}''$.

Если окажется, что $\mathcal{E}'$ короче, чем $\mathcal{E}''$ и все термы из $\mathcal{E}'$ равны соответствующим термам из $\mathcal{E}''$, то считается, что $\mathcal{E}' < \mathcal{E}''$.

Более формально: для любых объектных выражений $\mathcal{E}$, $\mathcal{E}'$, $\mathcal{E}''$ и любых объектных термов $\mathcal{T}, \mathcal{T}', \mathcal{T}''$ имеет место

- Если $\mathcal{E}' < \mathcal{E}''$, то $\mathcal{T}\ \mathcal{E}' < \mathcal{T}\ \mathcal{E}''$.
- Если $\mathcal{T}' < \mathcal{T}''$, то $\mathcal{T}'\ \mathcal{E}' < \mathcal{T}''\ \mathcal{E}''$.
- $\square < \mathcal{T}\ \mathcal{E}$, где через $\square$ обозначено пустое объектное выражение.

Для объектных термов отношение порядка определяется следующим образом.

Все символы предшествуют термам вида $(\mathcal{E})$, т.е. всегда

$$\mathcal{S} < (\mathcal{E})$$

Сравнение термов вида $(\mathcal{E})$ сводится к сравнению их содержимого, т.е.

Если $\mathcal{E}' < \mathcal{E}''$, то $(\mathcal{E}') < (\mathcal{E}'')$.

Каждый символ принадлежит одному и только одному из следующих множеств:

символы-литеры
символы-слова
символы-числа
ссылки на ящики
ссылки на векторы
ссылки на строки
ссылки на таблицы

Эти множества мы будем называть типами символов. Если два символа принадлежат одному и тому же типу, они называются однотипными, в противном случае – разнотипными. Мы считаем что на множестве типов определено отношение порядка, и что типы были перечислены выше в порядке возрастания, т.е. множество символов-литер предшествует множеству символов-слов и т.д.

Если два символа $\mathcal{S}'$ и $\mathcal{S}''$ принадлежат двум различным типам $Type'$ и $Type''$, и при этом $Type' < Type''$, то считается, что $\mathcal{S}' < \mathcal{S}''$.

Однотипные символы сравниваются следующим образом.

Символы-литеры упорядочены в соответствии с их кодами ASCII.

Символы-слова преобразуются в цепочки символов-литер, соответствующие изображениям слов, а эти цепочки сравниваются как описано выше.

Символы-числа сравниваются в соответствии с естественным порядком на множестве чисел.

Отношение порядка на множествах ссылок на ящики, векторы, строки и таблицы зависит от реализации.

3.9 `Convert`: преобразования между различными типами данных

```
$func  ToLower       e.Char = e.Char;
$func  ToUpper       e.Char = e.Char;
$func  CharsToBytes  e.Char = e.Int;
$func  BytesToChars  e.Int  = e.Char;
$func  ToChars       e.Exp  = e.Char;
$func  ToWord        e.Exp  = s.Word;
$func? ToInt         e.Exp  = s.Int;
```

`ToLower` преобразует цепочку символов-литер в цепочку литер, в которой все прописные латинские буквы заменены на соответствующие строчные буквы.

`ToUpper` преобразует цепочку символов-литер в цепочку литер, в которой все строчные латинские буквы заменены на соответствующие прописные буквы.

`CharsToBytes` преобразует цепочку символов-литер в последовательность чисел, являющихся их кодами ASCII.

Если аргумент этих функций не является цепочкой символов- литер, результатом является `$error(`$\mathcal{F}$ `"Invalid argument")`, где $\mathcal{F}$ – имя функции.

Например:

```
<ToLower 'AbCd+'>     =>  'abcd+'
<ToLower 25>          =>  $error(TO-LOWER "Invalid argument")
<ToUpper 'AbCd+'>     =>  'ABCD+'
<ToUpper 25>          =>  $error(TO-UPPER "Invalid argument")
<CharsToBytes 'ABC'>  =>  65 66 67
```

`BytesToChars` преобразует цепочку целых чисел, каждое из которых лежит в диапазоне от 0 до 255, в цепочку символов-литер, имеющих соответствующие коды ASCII.

Например:

```
<BytesToChars 65 66 67>   =>   'ABC'
```

Функции `ToChars`, `ToWord` и `ToInt` получают на входе произвольное объектное выражение и начинают свою работу с того, что преобразуют его в цепочку литер. Это преобразование выполняется следующим образом. Символы-литеры переходят сами в себя, круглые скобки заменяются на литеры `'('` и `')'`, слова заменяются на цепочки литер, соответствующие их печатным именам, числа заменяются на их изображения в виде цепочек литер, ссылки на строки заменяются на содержимое строк, все прочие ссылки заменяются на некоторые цепочки литер, которые зависят от реализации.

Если размер цепочки литер, полученной в результате вышеописанного преобразования, больше, чем допускается реализацией, результатом работы функций является

`$error(`$\mathcal{F}$ `"Argument too large for conversion").`

Дальнейшая работа функций `ToChars`, `ToWord` и `ToInt` происходит следующим образом.

`ToChars` выдает полученную цепочку литер в качестве результата.

```
<ToChars "John">          =>   'John'
<ToChars 'John'>          =>   'John'
<ToChars 326>             =>   '326'
<ToChars -326>            =>   '-326'
<ToChars (-326) "John">   =>   '(-326)John'
```

`ToWord` преобразует полученную цепочку литер в слово с соответствующим печатным именем.

```
<ToWord "John">          =>   "John"
<ToWord 'John'>          =>   "John"
<ToWord 326>             =>   "326"
<ToWord -326>            =>   "-326"
<ToWord (-326) "John">   =>   "(-326)John"
```

`ToInt` преобразует полученную цепочку литер в целое число `s.Int`. Если эта цепочка литер не является правильным изображением целого числа, результатом является `$fail(0)`.

Например:

```
<ToInt '326'>      =>   326
<ToInt '+326'>     =>   326
<ToInt "-3" '26'>  =>   -326
<ToInt -32 006>    =>   -326
<ToInt 'John'>     =>   $fail(0)
```

3.10 `Dos`: связь с операционной системой

```
$func Arg    s.Int = e.Arg;
$func Args   = e.Args;
$func GetEnv e.VarName = e.Value;
$func Time   = e.String;
$func Exit   s.ReturnCode = ;
$func Delay  s.MSeconds = ;
$func Sleep  s.Seconds = ;
$func Random s.Max = s.Rand;  // 0 <= s.Rand < s.Max
$func Randomize = ;
```

Эти функции служат для связи с операционной системой.

Аргументы функций должны удовлетворять следующим ограничениям. `s.Int` должно быть целым неотрицательным числом, `e.VarName` – цепочкой литер и слов, `s.ReturnCode` – целым числом из интервала от 0 до 255. Если хотя бы одно из этих условий не выполнено, функции выдают результат `$error(`$\mathcal{F}$ `"Invalid argument")`, где $\mathcal{F}$ – имя функции.

`Arg` выдает аргумент командной строки с номером `s.Int` в виде цепочки символов-литер. Если он отсутствует, выдается пустое выражение.

`Args` выдает аргументы командной строки в виде последовательности, каждый элемент которой заключен в скобки и представляет собой цепочку символов-литер.

`GetEnv` получает значение переменной окружения с именем `e.VarName` и выдает его в виде цепочки символов-литер.

`Time` выдает текущую дату и время в виде объектного выражения вида

$$DD\ MMM\ YYYY\ HH{:}MM{:}SS.SS$$

где DD – день месяца, MMM – сокращенное название месяца (`"Jan"`, `"Feb"`, `"Mar"`, `"Apr"`, `"May"`, `"Jun"`, `"Jul"`, `"Aug"`, `"Sep"`, `"Oct"`, `"Nov"`, `"Dec"`), $YYYY$ – номер года, $HH{:}MM{:}SS.SS$ – часы, минуты, секунды и сотые доли секунды. DD, $YYYY$, HH, MM, SS представлены в виде чисел, MMM – слово, все разделители между ними – символы-литеры `' '`, `':'` и `'.'`.

`Exit` завершает выполнение программы, при этом устанавливается код завершения равный `s.ReturnCode`. Если программа завершается обычным способом, т.е. вычисление вызова главной функции `<Main>` завершается, то код завершения зависит от результата, выданного функцией `Main`. Если этот результат – объектное выражение, то код завершения равен 0, если же результат имеет вид `$error(`$\mathcal{E}$`)`, код завершения равен 100.

`Delay` задерживает исполнение программы на `s.MSeconds` миллисекунд. Точность, с которой измеряется время, не превышает одной сотой секунды, и не может быть лучше, чем точность часов в MSDOS.

`Sleep` задерживает исполнение программы на `s.Seconds` секунд. Точность, с которой измеряется время, не превышает одной сотой секунды, и не может быть лучше, чем точность часов в MSDOS.

`Random` выдает псевдослучайное целое число в интервале от `0` до `s.Max` минус `1`.

`Randomize` инициализирует генератор случайных чисел случайным начальным значением.

3.11 StdIO: стандартный ввод-вывод

```
$channel StdIn StdOut StdErr;
```

`StdIn`, `StdOut` и `StdErr` – стандартные каналы ввода-вывода. Они автоматически открываются перед началом работы программы и автоматически закрываются по окончании работы программы.

```
$func Channel = s.Channel;
```

`Channel` создает новый канал `s.Channel`.

```
$func? OpenFile      s.Channel e.FileName s.Mode = ;
$func  CloseChannel  s.Channel = ;
```

`OpenFile` открывает канал `s.Channel` и связывает его с файлом с именем `e.FileName`. `s.Mode` – это символ-слово, который указывает режим, в котором будет происходить работа с файлом: `"r"` или `"R"` – чтение, `"w"` или `"W"` – запись, `"a"` или `"A"` – добавление в конец файла. Если файл открыть не удается, результатом работы `OpenFile` является неуспех `$fail(0)`.

`CloseChannel` закрывает канал `s.Channel`.

```
$func? IsEof        s.Channel = ;
```

`IsEof` проверяет, что текущая позиция у файла, с которым связан канал `s.Channel`, находится в конце файла.

```
$func? Read       = t.Term;
$func? ReadChar   = s.Char;
$func? ReadLine   = e.Char;
$func  Write      e.Exp = ;
$func  WriteLn    e.Exp = ;
$func  Print      e.Exp = ;
$func  PrintLn    e.Exp = ;
```

`Read` читает из канала `&StdIn` очередное изображение терма. Если не осталось ни одного терма – выдает `$fail(0)`.

`ReadChar` читает из канала `&StdIn` очередную литеру. Если не осталось ни одной литеры – выдает `$fail(0)`.

`ReadLine` читает из канала `&StdIn` все литеры до ближайшего конца строки (включительно) и выдает их в качестве результата (не включая литеру конец строки). Если не осталось ни одной литеры – выдает `$fail(0)`.

`Write` пишет в канал `&StdOut` изображение выражения `e.Exp` (если `e.Exp` не содержит динамических символов, его можно ввести обратно, терм за термом, с помощью функции `Read`).

`WriteLn` делает то же, что и `Write`, но в конце, добавляет литеру конца строки.

`Print` преобразует выражение `e.Exp` в цепочку литер так же, как это делает функция `ToChars`, и выводит его в канал `&StdOut`.

`PrintLn` делает то же, что и `Print`, но в конце добавляет литеру конца строки.

```
$func? ReadCh       s.Channel = t.Term;
$func? ReadCharCh   s.Channel = s.Char;
$func? ReadLineCh   s.Channel = e.Char;
$func  WriteCh      s.Channel e.Exp = ;
$func  WriteLnCh    s.Channel e.Exp = ;
$func  PrintCh      s.Channel e.Exp = ;
$func  PrintLnCh    s.Channel e.Exp = ;
```

Эти функции отличаются от соответствующих функций без окончания `Ch` тем, что операции выполняются над каналом `s.Channel`.

3.12 `String`: работа со строками

```
$func  String         e.Source = s.String;
$func  StringInit     s.String s.Len s.Fill = ;
$func  StringFill     s.String s.Fill = ;
$func  StringLength   s.String = s.Len;
$func  StringRef      s.String s.Index = s.Char;
$func  StringSet      s.String s.Index s.Char = ;
$func  StringReplace  s.String e.Source = ;
$func  Substring      s.String s.Index s.Len = s.NewString;
$func  SubstringFill  s.String s.Index s.Len s.Fill = ;
```

Эти функции используются для создания строк, изменения их содержимого и доступа к их содержимому. Их аргументы должны удовлетворять следующим требованиям. `s.String` должен быть ссылкой на строку. `s.Index` и `s.Len` должны быть неотрицательными целыми числами. `s.Fill` может быть произвольным символом-литерой. `e.Source` должен быть последовательностью ссылок на строки, символов-слов и символов-литер.

Если хотя бы одно из этих условий не выполнено, результатом функций является `$error(`$\mathcal{F}$ `"Invalid argument")`, где $\mathcal{F}$ – имя функции.

В каждый момент времени строка содержит конечную (может быть пустую) последовательность символов-литер, которую мы будем называть содержимым строки. Если строка содержит последовательность из $n+1$ символов-литер $\mathcal{C}_0$, $\mathcal{C}_1$, ..., $\mathcal{C}_n$, мы будем говорить, что строка имеет длину $n+1$, а содержимое строки будем изображать в виде

$$\mathcal{C}_0\,\mathcal{C}_1\ \ldots\ \ldots\mathcal{C}_n$$

Таким образом, компоненты строки $\mathcal{C}_0, \mathcal{C}_1$, ..., $\mathcal{C}_n$ нумеруются начиная с нуля.

`String` создает новую строку и выдает ссылку на нее в качестве результата. При этом содержимое создаваемой строки формируется из `e.Source` следующим образом.

Пусть значение параметра `e.Source` имеет следующий вид: $\mathcal{S}_1\ \mathcal{S}_2\ \ldots\ \mathcal{S}_m$, где каждый из символов $\mathcal{S}_j$ является либо символом-литерой, либо символом-словом, либо ссылкой на строку. Тогда каждый из $\mathcal{S}_j$ преобразуется следующим образом.

Если $\mathcal{S}_j$ является символом-литерой $\mathcal{C}$, он остается без изменения.

Если $\mathcal{S}_j$ является символом-словом, он заменяется на цепочку литер, которая является печатным именем этого символа.

Если $\mathcal{S}_j$ является ссылкой на какую-то строку, он заменяется на содержимое этой строки (при этом состояние самой строки не изменяется).

Преобразованное таким образом значение параметра `e.Source` становится содержимым новой строки.

`StringInit` изменяет содержимое строки, на которую указывает `s.String`. Старое содержимое уничтожается и вместо него создается новое содержимое длины `s.Len`, причем каждая компонента строки принимает значение `s.Fill`.

`StringFill` изменяет содержимое строки, на которую указывает `s.String`. При этом длина строки не меняется, а каждая компонента строки принимает значение `s.Fill`.

`StringLength` выдает длину строки, на которую указывает `s.String`.

`StringRef` извлекает из строки, на который указывает `s.String`, содержимое ее компоненты с номером `s.Index`, которое и выдается в качестве результата.

`StringSet` изменяет содержимое строки, на которую указывает `s.String`. При этом длина строки не меняется, а компонента строки с номером `s.Index` принимает значение `s.Char`.

`StringReplace` изменяет содержимое строки, на которую указывает `s.String`. Старое содержимое уничтожается, а новое содержимое формируется из значения параметра `e.Source` точно таким же способом, как и в случае функции `String`.

`Substring` создает новую строку и выдает ссылку на нее в качестве результата. При этом содержимое создаваемой строки формируется следующим образом. Рассматривается содержимое исходной строки, т.е. строки, на которую указывает `s.String`. Пусть оно имеет вид $\mathcal{C}_0\ \mathcal{C}_1\ \ldots\ \mathcal{C}_n$. Тогда от него отбрасываются первые `s.Index` символов, а первые `s.Len` из оставшихся символов считаются содержимым новой строки.

В любом случае, в результате работы функции `Substring` состояние исходной строки не изменяется.

`SubstringFill` изменяет содержимое строки, на которую указывает `s.String`. При этом длина строки не меняется, а `s.Len` последовательно расположенных компонент строки, начиная с компоненты с номером `s.Index`, принимают значение `s.Char`.

Если размер исходной строки недостаточен для выполнения одной из вышеуказанных операций, в качестве результата выдается
`$error(`$\mathcal{F}$ `"Index out of range")`, где $\mathcal{F}$ – имя функции, а состояние строки не меняется.

Если при выполнении одной из вышеуказанных операций возникает необходимость создать содержимое строки, длина которого больше, чем допускается реализацией, в качестве результата выдается
`$error(`$\mathcal{F}$ `"Size limit  exceeded")`, где $\mathcal{F}$ – имя функции, а состояние строки не меняется.

3.13 Table: работа с таблицами

```
$func  Table         = s.Tab;
$func  Bind          s.Tab (e.Key) (e.Val) = ;
$func  Unbind        s.Tab e.Key = ;
$func? Lookup        s.Tab e.Key = e.Val;
$func? IsInTable     s.Tab e.Key = ;
$func  Domain        s.Tab = e.Domain ;
$func  TableCopy     s.Tab = s.TabCopy;
$func  ReplaceTable  s.TargetTable s.SourceTable = ;
```

`Table` создает новую пустую таблицу и выдает символ-ссылку на эту таблицу.

`Bind` связывает ключ `e.Key` со значением `e.Val` в таблице, на которую ссылается `s.Tab`.

`Unbind` удаляет из таблицы, на которую ссылается `s.Tab`, ключ `e.Key` и связанное с ним значение, если такой ключ был в таблице.

`Lookup` находит в таблице, на которую ссылается `s.Tab`, значение, связанное с ключом `e.Key` и выдает его. Если такого ключа в таблице нет - результатом является `$fail(0)`.

`IsInTable` проверяет, что ключ `e.Key` находится в таблице, на которую ссылается `s.Tab`.

`Domain` выдает список ключей, содержащихся в таблице, на которую ссылается `s.Tab`. А именно, если таблица содержит множество ключей $\{\mathcal{E}_1, \mathcal{E}_2, \ldots, \mathcal{E}_n\}$, `e.Domain` имеет вид

$$(\mathcal{E}_1)(\mathcal{E}_2)\ldots(\mathcal{E}_n)$$

где порядок расположения ключей описанием языка не определен.

`TableCopy` создает новую таблицу, копирует в него содержимое таблицы, на которую ссылается `s.Tab`, и выдает в качестве результата ссылку на новую таблицу.

`ReplaceTable` уничтожает прежнее содержимое таблицы, на которую ссылается `s.TargetTable`, и заменяет его на копию содержимого таблицы, на которую ссылается `s.SourceTable`.

3.14 Vector: работа с векторами

```
$func  Vector         e.Source = s.Vector;
$func  VectorToExp    s.Vector = e.Exp;
$func  VectorInit     s.Vector s.Len e.Fill = ;
$func  VectorFill     s.Vector e.Fill = ;
$func  VectorLength   s.Vector = s.Len;
$func  VectorRef      s.Vector s.Index = e.Exp;
$func  VectorSet      s.Vector s.Index e.Exp = ;
```

```
$func  VectorReplace  s.Vector e.Source = ;
$func  Subvector      s.Vector s.Index s.Len = s.NewVector;
$func  SubvectorFill  s.Vector s.Index s.Len e.Fill = ;
```

Эти функции используются для создания векторов, изменения их содержимого и доступа к их содержимому. Их аргументы должны удовлетворять следующим требованиям. `s.Vector` должен быть ссылкой на вектор. `s.Index` и `s.Len` должны быть неотрицательными целыми числами. `e.Fill` может быть произвольным объектным выражением. `e.Source` должен быть последовательностью ссылок на векторы и термов вида `(`$\mathcal{E}$`)`.

Если хотя бы одно из этих условий не выполнено, результатом функций является `$error(`$\mathcal{F}$ `"Invalid argument")`, где $\mathcal{F}$ – имя функции.

В каждый момент времени вектор содержит конечную (может быть, пустую) последовательность объектных выражений, которую мы будем называть содержимым вектора. Если вектор содержит последовательность из $n+1$ объектных выражений $\mathcal{E}_0$, $\mathcal{E}_1$, …, $\mathcal{E}_n$, мы будем говорить, что вектор имеет длину $n+1$, а содержимое вектора будем изображать в виде

`(`$\mathcal{E}_0$`)(`$\mathcal{E}_1$`)` … `(`$\mathcal{E}_n$`)`

Таким образом, компоненты вектора $\mathcal{E}_0$, $\mathcal{E}_2$, …, $\mathcal{E}_n$ нумеруются начиная с нуля.

`Vector` создает новый вектор и выдает ссылку на него в качестве результата. При этом содержимое создаваемого вектора формируется из `e.Source` следующим образом.

Пусть значение параметра `e.Source` имеет следующий вид: $\mathcal{T}_1$ $\mathcal{T}_2$ … $\mathcal{T}_m$, где каждый из объектных термов $\mathcal{T}_j$ либо имеет вид `(`$\mathcal{E}$`)`, либо является ссылкой на вектор. Тогда каждый из $\mathcal{T}_j$ преобразуется следующим образом.

Если $\mathcal{T}_j$ имеет вид `(`$\mathcal{E}$`)`, он остается без изменения.

Если $\mathcal{T}_j$ является ссылкой на какой-то вектор, он заменяется на содержимое этого вектора (при этом состояние самого вектора не изменяется).

Преобразованное таким образом значение параметра `e.Source` становится содержимым нового вектора.

`VectorToExp` выдает в виде объектного выражения содержимое вектора, на который указывает `s.Vector`.

`VectorInit` изменяет содержимое вектора, на который указывает `s.Vector`. Старое содержимое уничтожается и вместо него создается новое содержимое длины `s.Len`, причем каждая компонента вектора принимает значение `e.Fill`.

`VectorFill` изменяет содержимое вектора, на который указывает `s.Vector`. При этом длина вектора не меняется, а каждая компонента вектора принимает значение `e.Fill`.

`VectorLength` выдает длину вектора, на который указывает `s.Vector`.

`VectorRef` извлекает из вектора, на который указывает `s.Vector`, содержимое его компоненты с номером `s.Index`, которое и выдается в качестве результата.

`VectorSet` изменяет содержимое вектора, на который указывает `s.Vector`. При этом длина вектора не меняется, а компонента вектора с номером `s.Index` принимает значение `e.Exp`.

`VectorReplace` изменяет содержимое вектора, на который указывает `s.Vector`. Старое содержимое уничтожается, а новое содержимое формируется из значения параметра `e.Source` точно таким же способом, как и в случае функции `Vector`.

`Subvector` создает новый вектор и выдает ссылку на него в качестве результата. При этом содержимое создаваемого вектора формируется следующим образом. Рассматривается содержимое исходного вектора, т.е. вектора, на который указывает `s.Vector`. Пусть оно имеет вид `(`$\mathcal{E}_0$`)(`$\mathcal{E}_1$`) ... (`$\mathcal{E}_n$`)`. Тогда от него отбрасываются первые `s.Index` термов, а первые `s.Len` из оставшихся термов считаются содержимым нового вектора.

Состояние исходного вектора при этом не меняется.

`SubvectorFill` изменяет содержимое вектора, на который указывает `s.Vector`. При этом длина вектора не меняется, а `s.Len` последовательно расположенных компонент вектора, начиная с компоненты с номером `s.Index`, принимают значение `e.Exp`.

Если размер исходного вектора недостаточен для выполнения одной из вышеуказанных операций, в качестве результата выдается
`$error(`$\mathcal{F}$ `"Index out of range")`, где $\mathcal{F}$ – имя функции, а состояние вектора не меняется.

Если при выполнении одной из вышеуказанных операций возникает необходимость создать содержимое вектора, длина которого больше, чем допускается реализацией, в качестве результата выдается
`$error(`$\mathcal{F}$ `"Size limit exceeded")`, где $\mathcal{F}$ – имя функции, а состояние вектора не меняется.

Глава 4

Принципы реализации Рефала Плюс

Цель данной главы - показать, что для Рефала Плюс существует достаточно простая и естественная реализация, которая может рассматриваться как некоторое обобщение по отношению к реализациям Рефала-2 [КлР 1987], [Ром 1987б].

В принципе, материал данной главы может рассматриваться как еще одно описание семантики Рефала Плюс, основанное на компиляции, в противоположность к описанию семантики, изложенному в главе 2, основанному на "естественной операционной семантике".

В настоящее время существует несколько практических реализаций Рефала Плюс, каждая из которых может рассматриваться, как конкретизация или усовершенствование схемы реализации, изложенной в данной главе [АОППС 2004], [АО 2004].

Обзор и сравнительной анализ этих реализаций не входят в задачу данной книги.

4.1 Общая схема реализации

Программа, написанная на Рефале Плюс, компилируется в код виртуальной машины. Этот код в дальнейшем может как интерпретироваться, так и компилироваться.

В дальнейшем мы описываем семантику виртуального кода в терминах абстрактной виртуальной машины. Затем мы описываем алгоритм компиляции программ на Рефале Плюс в виртуальный код.

Способы отображения абстрактной виртуальной машины на традиционную фон-Неймановскую машину в данной книге не рассматриваются. В основном, можно использовать те же методы, которые применялись при реализации Рефала-2 [КлР 1987], [Ром 1987а], хотя у Рефала Плюс есть и некоторые особенности. Например, для Рефала Плюс не

очень подходит тот способ представления объектных выражений, который применялся в реализациях Рефала-2 [АбР 1988].

Не обсуждаются и некоторые дополнительные оптимизации, которые могут выполняться при компиляции с Рефала Плюс в виртуальный код, например, оптимизации, связанные с сокращением перебора при сопоставлении с образцом [Ром 1987а].

4.2 Виртуальный код

Программа, написанная в виртуальном коде, представляет собой последовательность команд, каждая из которых содержит код операции и, может быть, дополнительные операнды.

При записи программ в виртуальном коде мы будем применять следующие обозначения.

Пустую последовательность команд будем обозначать через []. Непустую последовательность команд, начинающуюся с команды X, за которой следует последовательность команд C, будем обозначать через $X : C$. Таким образом, последовательность команд X_1, X_2, ..., X_n записывается в виде $X_1 : X_2 : \ldots : X_n : []$. Такая запись удобна при описании команд виртуальной машины. Однако, когда мы будем приводить примеры программ в виртуальном коде и описывать алгоритм компиляции в виртуальный код, мы будем пользоваться также и альтернативной записью $X_1; X_2; \ldots; X_n$.

4.2.1 Состояние виртуальной машины

Состояние виртуальной машины представляет собой упорядоченную четверку

$$\langle C, S, D, E \rangle$$

где C - список исполняемых команд (Code), S - стек данных (Stack), D - свалка (Dump), E - свалка ошибок (Error dump).

Стек данных S используется для хранения объектных выражений, которые получаются в результате вычислений. Через этот стек передаются значения параметров функций и возвращаются результаты функций.

Свалка D представляет собой стек, содержащий упорядоченные пары вида (L, S), где L - последовательность команд, а S - стек данных. Свалка используется для сохранения текущего состояния стека данных (с целью его дальнейшего восстановления), и для запоминания последовательностей команд, которые (может быть) потребуется выполнять в дальнейшем.

Свалка ошибок E представляет собой стек, содержащий упорядоченные тройки вида (L, S, D), где L - последовательность команд, S - стек

данных, а D - свалка. Свалка ошибок предназначена для запоминания последовательности команд, текущего состояния стека данных и текущего состояния свалки. Эта информация используется в случае возникновения ошибок для восстановления состояния виртуальной машины.

Работа виртуальной машины описывается с помощью правил перехода из одного состояния в другое, которые имеют вид

$$\langle C, S, D, E\rangle \;\Rightarrow\; \langle C', S', D', E'\rangle$$

и означают, что если виртуальная машина видит последовательность команд C, стек данных S, свалку D и свалку ошибок E, то она переходит в новое состояние, которое содержит последовательность команд C', стек данных S', свалку D' и свалку ошибок E'.

Если переход возможен только при выполнении дополнительного условия P, правило перехода записывается в виде

$$\langle C, S, D, E\rangle \;\Rightarrow\; \{P\}\,\langle C', S', D', E'\rangle$$

4.2.2 Некоторые обозначения

При описании команд виртуальной машины мы будем использовать следующие обозначения.

Пустое объектное выражение будет обозначаться через $\square$.

Пустой стек будет обозначаться через $[]$, а непустой стек - через $X : S$, где X - вершина стека, а S - стек, который получиться из $X : S$ после выталкивания из него верхнего элемента X.

Стек $X_1 : \ldots : X_n : [\,]$, содержащий элементы $X_1, \ldots, X_n$, будет обозначаться также и через $[X_1 : \ldots : X_n]$.

Если S - стек, то его k-й элемент, считая от вершины, обозначается через S/k. А именно, если $S = X_0 : \ldots : X_k : S'$, то $S/k = X_k$.

4.2.3 Метки и адреса

Программы в виртуальном коде могут содержать метки, на которые может передаваться управление. Если требуется пометить меткой L последовательность команд C, мы записываем это следующим образом: `LABEL` $L; C$. После этого можно передавать управление на эту последовательность, например с помощью команды перехода `JUMP` L.

Если требуется пометить начало скомпилированного определения функции, мы будем писать `FUNSTART` L вместо `LABEL` L. Это может оказаться полезным для тех реализаций виртуальной машины, в которых существует различие между “длинными” и “короткими” переходами.

Смысл команды `JUMP` можно было бы определить с помощью следующего правила перехода:

$$\langle \texttt{JUMP}\ L : \ldots : \texttt{LABEL}\ L : C, S, D, E\rangle \;\Rightarrow\; \langle C, S, D, E\rangle$$

которое можно истолковать следующим образом: “найди в последовательности команд метку L и выполняй команды, которые идут вслед за ней”.

Мы знаем, однако, что при исполнении программ на обычных компьютерах никаких поисков меток не выполняется. Вместо этого, команда перехода содержит некоторый “адрес”, который прямо указывает, где находится следующая команда, подлежащая выполнению. Поэтому `LABEL` L - это “не настоящая” команда, а скорее директива для загрузчика, которая указывает, что во всех командах, вместо метки L следует подставить адрес команды, следующей за `LABEL` L, а саму директиву `LABEL` L - выкинуть из программы.

Таким образом, в реальном компьютере все фрагменты кода представляются их адресами, а метки в программе обозначают некоторые адреса. Перед выполнением команды, ее адрес всякий раз преобразуется в саму команду. Однако такая точка зрения неудобна при описании виртуальных команд, поэтому, при записи правил перехода мы будем считать, что все фрагменты кода представлены самими этими фрагментами, т.е. что все адреса как бы заранее преобразованы в соответствующие им фрагменты. С этой точки зрения, например, команда `JUMP` L содержит не адрес L, а непосредственно последовательность команд, на которую следует передать управление. При этом, команда `JUMP` описывается с помощью следующего правила перехода:

$$\langle \texttt{JUMP}\ L : C, S, D, E\rangle \ \Rightarrow\ \langle L, S, D, E\rangle$$

Мы видим, что принятое нами соглашение позволяет сделать описание команд более кратким и наглядным.

В дальнейшем фрагменты виртуального кода обозначаются через C или L.

4.2.4 Пустая команда

Команда `NOP` не выполняет никаких действий:

$$\langle \texttt{NOP} : C, S, D, E\rangle \ \Rightarrow\ \langle C, S, D, E\rangle$$

4.2.5 Команды управления неуспехами

Команда `ALT` L запоминает в свалке последовательность команд L и текущее состояние стека данных S. Если в дальнейшем возникает неуспех, состояние стека данных восстанавливается и начинает выполняться список команд L. Таким образом, `ALT` L создает развилку для перехвата неуспехов.

Команда `CUT` n выбрасывает со свалки n последних развилок, отменяя n перехватов неуспехов.

Команда FAIL всегда порождает неуспех: извлекает из свалки (L', S'), приводит стек данных в состояние S' и начинает выполнять последовательность команд L'.

$$\langle \texttt{ALT}\ L : C, S, D, E\rangle \ \Rightarrow\ \langle C, S, (L, S) : D, E\rangle$$
$$\langle \texttt{CUT}\ n : C, S, (L_1, S_1) : \ldots : (L_n, S_n) : D, E\rangle \ \Rightarrow\ \langle C, S, D, E\rangle$$
$$\langle \texttt{FAIL} : C, S, (L', S') : D, E\rangle \ \Rightarrow\ \langle L', S', D, E\rangle$$

4.2.6 Команды переходов и вызовов функций

Команда JUMP L передает управление на последовательность команд L.

Команда CALL a L вызывает функцию, имеющую a параметров. Значения этих параметров должны находиться в a верхних элементах стека данных. Последовательность команд, идущих за командой CALL , запоминается в свалке вместе с состоянием стека данных. Они используются в дальнейшем для возврата из функции.

Команда RETURN осуществляет успешный возврат из функции, а команда RETURNFAIL осуществляет возврат из функции с выработкой неуспеха.

Команда TAILCALL L осуществляет "хвостовой" вызов функции. Она может использоваться в тех случаях, когда сразу после команды CALL следует команда RETURN.

$$\langle \texttt{JUMP}\ L : C, S, D, E\rangle \ \Rightarrow\ \langle L, S, D, E\rangle$$
$$\langle \texttt{CALL}\ a\ L : C, \mathcal{E}_a : \ldots : \mathcal{E}_1 : S, D, E\rangle \ \Rightarrow$$
$$\quad \langle L, \mathcal{E}_a : \ldots : \mathcal{E}_1 : [\,], (C, S) : D, E\rangle$$
$$\langle \texttt{RETURN} : C, \mathcal{E}_b : \ldots : \mathcal{E}_1 : [\,], (C', S') : D, E\rangle \ \Rightarrow$$
$$\quad \langle C', \mathcal{E}_b : \ldots : \mathcal{E}_1 : S', D, E\rangle$$

$$\langle \texttt{RETURNFAIL} : C, S, (C', S') : D, E\rangle \ \Rightarrow\ \langle \texttt{FAIL} : [\,], S', D, E\rangle$$
$$\langle \texttt{TAILCALL}\ L : C, S, D, E\rangle \ \Rightarrow\ \langle L, S, D, E\rangle$$

Примечание. На первый взгляд TAILCALL $L \simeq$ JUMP L, и RETURNFAIL $\simeq$ CUT 1;FAIL. Но это справедливо только если адреса переходов между функциями и внутри одной функции представлены одинаково. Во многих реализациях виртуального кода это не так. Поэтому лучше предусмотреть в виртуальном коде различные команды.

4.2.7 Команды обработки ошибок

Команда PUSHTRAP L запоминает текущее состояние в свалке ошибок. Команда POPTRAP выбрасывает один элемент из свалки ошибок. Команда ERROR извлекает состояние из свалки ошибок и переводит машину в это состояние. При этом верхний элемент стека данных переносится на

вершину стека данных в новом состоянии. Таким способом передается сообщение об ошибке.

$$\langle \texttt{PUSHTRAP}\ L : C, S, D, E\rangle \ \Rightarrow\ \langle C, S, D, (L, S, D) : E\rangle$$
$$\langle \texttt{POPTRAP} : C, S, D, (L', S', D') : E\rangle \ \Rightarrow\ \langle C, S, D, E\rangle$$
$$\langle \texttt{ERROR} : C, \mathcal{E} : S, D, (L', S', D') : E\rangle \ \Rightarrow\ \langle L', \mathcal{E} : S', D', E\rangle$$

4.2.8 Команды построения объектных выражений

Команда `PUSHQ` $\mathcal{E}$ вталкивает в стек данных объектное выражение $\mathcal{E}$.

Команда `PUSH` m находит в стеке данных объектное выражение, находящееся на расстоянии m от вершины стека данных, и вталкивает его в стек данных.

Команда `BR` надевает скобки на объектное выражение, находящееся на вершине стека данных.

Команда `CONC` конкатенирует (приписывает друг к другу) два объектных выражения, находящихся в двух верхних позициях стека данных.

$$\langle \texttt{PUSHQ}\ \mathcal{E} : C, S, D, E\rangle \ \Rightarrow\ \langle C, \mathcal{E} : S, D, E\rangle$$
$$\langle \texttt{PUSH}\ m : C, S, D, E\rangle \ \Rightarrow\ \{S/m = \mathcal{E}\}\,\langle C, \mathcal{E} : S, D, E\rangle$$

$$\langle \texttt{BR} : C, \mathcal{E} : S, D, E\rangle \ \Rightarrow\ \langle C, (\mathcal{E}) : S, D, E\rangle$$
$$\langle \texttt{CONC} : C, \mathcal{E}'' : \mathcal{E}' : S, D, E\rangle \ \Rightarrow\ \langle C, \mathcal{E}'\,\mathcal{E}'' : S, D, E\rangle$$

4.2.9 Команды очистки стека

Команда `SQUEEZE` a n сдвигает а верхних элементов стека данных на n позиций в глубь стека. При этом из стека выкидываются n элементов.

Команда `SLIDE` n сдвигает верхний элемент стека данных на n позиций в глубь стека. При этом из стека выкидываются n элементов.

Команда `POP` n выкидывает из стека данных n верхних элементов.

$$\langle \texttt{SQUEEZE}\ a\ n : C, \mathcal{E}_1 : \ldots : \mathcal{E}_a : \mathcal{E}'_1 : \ldots : \mathcal{E}'_n : S, D, E\rangle \ \Rightarrow$$
$$\langle C, \mathcal{E}_1 : \ldots : \mathcal{E}_a : S, D, E\rangle$$
$$\langle \texttt{SLIDE}\ n : C, \mathcal{E} : \mathcal{E}'_1 : \ldots : \mathcal{E}'_n : S, D, E\rangle \ \Rightarrow\ \langle C, \mathcal{E} : S, D, E\rangle$$
$$\langle \texttt{POP}\ n : C, \mathcal{E}'_1 : \ldots : \mathcal{E}'_n : S, D, E\rangle \ \Rightarrow\ \langle C, S, D, E\rangle$$

4.2.10 Команды анализа объектных выражений

В этом разделе, для краткости, при записи условий в правилах перехода, используются следующие соглашения.

Условие вида $X = Y$, где Y содержит какие-то величины $x_1, \ldots, x_k$, которые не определены в левой части правила перехода, означает, что "существуют и могут быть однозначно определены такие $x_1, \ldots, x_k$, что выполнено $X = Y$". При этом $x_1, \ldots, x_k$ могут употребляться в правой части правила перехода.

Условие вида $X \neq Y$, где Y содержит какие-то величины $x_1, \ldots, x_k$ которые не определены в левой части правила перехода, означает, что "не существует таких $x_1, \ldots, x_k$, что выполнено $X = Y$".

Например, пусть $\mathcal{E}'$ и $\mathcal{E}''$ не встречаются в левой части правила. Тогда условие $S/m = (\mathcal{E}')\,\mathcal{E}''$ означает, что можно найти (и при том однозначно) такие объектные выражения $\mathcal{E}'$ и $\mathcal{E}''$, что m-й элемент стека S равен объектному выражению $(\mathcal{E}')\,\mathcal{E}''$. В то же время, условие $S/m \neq (\mathcal{E}')\,\mathcal{E}''$ означает, что не существует таких объектных выражений $\mathcal{E}'$ и $\mathcal{E}''$, для которых m-й элемент стека S равен объектному выражению $(\mathcal{E}')\,\mathcal{E}''$.

Описанные ниже команды выполняют анализ объектных выражений и разложение их на составляющие их подвыражения. При этом, если выражение имеет требуемую структуру, его подкомпоненты заносятся в стек данных. Если же выражение не имеет требуемой структуры, возникает неуспех, и выполняются те же действия, что и в случае команды `FAIL`. Первый операнд всех этих команд показывает расположение в стеке данных того выражения, которое требуется анализировать.

Например, команда `LS` m работает следующим образом. Рассматривается объектное выражение, которое находится в стеке данных на расстоянии m от вершины стека. Пусть это выражение равно $\mathcal{E}$. Тогда, если $\mathcal{E}$ пусто, возникает неуспех. Если же $\mathcal{E}$ не пусто и имеет вид $(\mathcal{E}')\,\mathcal{E}''$, то также возникает неуспех. И наконец, если $\mathcal{E}$ имеет вид $\mathcal{S}\,\mathcal{E}'$, то стек данных S переходит в состояние $\mathcal{E}' : \mathcal{S} : S$, и работа команды успешно завершается.

Команды анализа выражений распадаются на следующие группы: проверки условий, жесткое отщепление слева, жесткое отщепление справа, перебор слева и перебор справа.

4.2.10.1 Проверки условий

Команды из этой группы проверяют, что выражение удовлетворяет определенным требованиям. При этом состояние стека данных не изменяется (исключением является только команда `CB`).

$$\langle \texttt{EMPTY}\ m : C, S, D, E\rangle \;\Rightarrow\; \{S/m = \square\}\,\langle C, S, D, E\rangle$$
$$\langle \texttt{EMPTY}\ m : C, S, D, E\rangle \;\Rightarrow\; \{S/m \neq \square\}\,\langle \texttt{FAIL} : [\,], S, D, E\rangle$$

$$\langle \texttt{CV}\ m : C, S, D, E\rangle \;\Rightarrow\; \{S/m \neq \square\}\,\langle C, S, D, E\rangle$$
$$\langle \texttt{CV}\ m : C, S, D, E\rangle \;\Rightarrow\; \{S/m = \square\}\,\langle \texttt{FAIL} : [\,], S, D, E\rangle$$

$$\langle \texttt{CB}\ m : C, S, D, E\rangle \;\Rightarrow\; \{S/m = (\mathcal{E})\}\,\langle C, \mathcal{E} : S, D, E\rangle$$
$$\langle \texttt{CB}\ m : C, S, D, E\rangle \;\Rightarrow\; \{S/m \neq (\mathcal{E})\}\,\langle \texttt{FAIL} : [\,], S, D, E\rangle$$

$$\langle \texttt{CQ}\ m\ \mathcal{E} : C, S, D, E\rangle \;\Rightarrow\; \{S/m = \mathcal{E}\}\,\langle C, S, D, E\rangle$$
$$\langle \texttt{CQ}\ m\ \mathcal{E} : C, S, D, E\rangle \;\Rightarrow\; \{S/m \neq \mathcal{E}\}\,\langle \texttt{FAIL} : [\,], S, D, E\rangle$$

$$\langle \texttt{CS}\ m : C, S, D, E\rangle \ \Rightarrow\ \{S/m = \mathcal{S}\}\,\langle C, S, D, E\rangle$$
$$\langle \texttt{CS}\ m : C, S, D, E\rangle \ \Rightarrow\ \{S/m \neq \mathcal{S}\}\,\langle \texttt{FAIL} : [\,], S, D, E\rangle$$

$$\langle \texttt{CT}\ m : C, S, D, E\rangle \ \Rightarrow\ \{S/m = \mathcal{T}\}\,\langle C, S, D, E\rangle$$
$$\langle \texttt{CT}\ m : C, S, D, E\rangle \ \Rightarrow\ \{S/m \neq \mathcal{T}\}\,\langle \texttt{FAIL} : [\,], S, D, E\rangle$$

$$\langle \texttt{CEQ}\ m\ m' : C, S, D, E\rangle \ \Rightarrow\ \{S/m = S/m'\}\,\langle C, S, D, E\rangle$$
$$\langle \texttt{CEQ}\ m\ m' : C, S, D, E\rangle \ \Rightarrow\ \{S/m \neq S/m'\}\,\langle \texttt{FAIL} : [\,], S, D, E\rangle$$

4.2.10.2 Жесткое отщепление слева

Команды из этой группы отщепляют часть выражения с левого конца. Причем, если отщепление возможно, оно делается однозначно. Исходное выражение при этом разбивается на части, которые заносятся в стек данных.

$$\langle \texttt{LB}\ m : C, S, D, E\rangle \ \Rightarrow$$
$$\{S/m = (\mathcal{E}')\,\mathcal{E}''\}\,\langle C, \mathcal{E}' : (\mathcal{E}') : \mathcal{E}'' : S, D, E\rangle$$
$$\langle \texttt{LB}\ m : C, S, D, E\rangle \ \Rightarrow$$
$$\{S/m \neq (\mathcal{E}')\,\mathcal{E}''\}\,\langle \texttt{FAIL} : [\,], S, D, E\rangle$$

$$\langle \texttt{LQ}\ m\ \mathcal{E} : C, S, D, E\rangle \ \Rightarrow\ \{S/m = \mathcal{E}\,\mathcal{E}'\}\,\langle C, \mathcal{E}' : \mathcal{E} : S, D, E\rangle$$
$$\langle \texttt{LQ}\ m\ \mathcal{E} : C, S, D, E\rangle \ \Rightarrow\ \{S/m \neq \mathcal{E}\,\mathcal{E}'\}\,\langle \texttt{FAIL} : [\,], S, D, E\rangle$$

$$\langle \texttt{LS}\ m : C, S, D, E\rangle \ \Rightarrow\ \{S/m = \mathcal{S}\,\mathcal{E}\}\,\langle C, \mathcal{E} : \mathcal{S} : S, D, E\rangle$$
$$\langle \texttt{LS}\ m : C, S, D, E\rangle \ \Rightarrow\ \{S/m \neq \mathcal{S}\,\mathcal{E}\}\,\langle \texttt{FAIL} : [\,], S, D, E\rangle$$

$$\langle \texttt{LT}\ m : C, S, D, E\rangle \ \Rightarrow\ \{S/m = \mathcal{T}\,\mathcal{E}\}\,\langle C, \mathcal{E} : \mathcal{T} : S, D, E\rangle$$
$$\langle \texttt{LT}\ m : C, S, D, E\rangle \ \Rightarrow\ \{S/m \neq \mathcal{T}\,\mathcal{E}\}\,\langle \texttt{FAIL} : [\,], S, D, E\rangle$$

$$\langle \texttt{LEQ}\ m\ m' : C, S, D, E\rangle \ \Rightarrow$$
$$\{S/m = \mathcal{E}'\,\mathcal{E},\ \text{где}\ \mathcal{E}' = S/m'\}\,\langle C, \mathcal{E} : \mathcal{E}' : S, D, E\rangle$$
$$\langle \texttt{LEQ}\ m\ m' : C, S, D, E\rangle \ \Rightarrow$$
$$\{S/m \neq \mathcal{E}'\,\mathcal{E},\ \text{где}\ \mathcal{E}' = S/m'\}\,\langle \texttt{FAIL} : [\,], S, D, E\rangle$$

4.2.10.3 Жесткое отщепление справа

Команды из этой группы отщепляют часть выражения с правого конца. Причем, если отщепление возможно, оно делается однозначно. Исходное выражение при этом разбивается на части, которые заносятся в стек данных.

$$\langle \texttt{RB}\ m : C, S, D, E\rangle \ \Rightarrow$$
$$\{S/m = \mathcal{E}''\,(\mathcal{E}')\}\,\langle C, \mathcal{E}' : (\mathcal{E}') : \mathcal{E}'' : S, D, E\rangle$$
$$\langle \texttt{RB}\ m : C, S, D, E\rangle \ \Rightarrow$$
$$\{S/m \neq \mathcal{E}''\,(\mathcal{E}')\}\,\langle \texttt{FAIL} : [\,], S, D, E\rangle$$

$$\langle \texttt{RQ}\ m\ \mathcal{E} : C, S, D, E\rangle \ \Rightarrow\ \{S/m = \mathcal{E}'\,\mathcal{E}\}\,\langle C, \mathcal{E}' : \mathcal{E} : S, D, E\rangle$$
$$\langle \texttt{RQ}\ m\ \mathcal{E} : C, S, D, E\rangle \ \Rightarrow\ \{S/m \neq \mathcal{E}'\,\mathcal{E}\}\,\langle \texttt{FAIL} : [\,], S, D, E\rangle$$

$$\langle \texttt{RS}\ m : C, S, D, E\rangle \ \Rightarrow\ \{S/m = \mathcal{E}\,\mathcal{S}\}\,\langle C, \mathcal{E} : \mathcal{S} : S, D, E\rangle$$
$$\langle \texttt{RS}\ m : C, S, D, E\rangle \ \Rightarrow\ \{S/m \neq \mathcal{E}\,\mathcal{S}\}\,\langle \texttt{FAIL} : [\,], S, D, E\rangle$$

$$\langle \texttt{RT}\ m : C, S, D, E\rangle \ \Rightarrow\ \{S/m = \mathcal{E}\,\mathcal{T}\}\,\langle C, \mathcal{E} : \mathcal{T} : S, D, E\rangle$$
$$\langle \texttt{RT}\ m : C, S, D, E\rangle \ \Rightarrow\ \{S/m \neq \mathcal{E}\,\mathcal{T}\}\,\langle \texttt{FAIL} : [\,], S, D, E\rangle$$

$$\langle \texttt{REQ}\ m\ m' : C, S, D, E\rangle \ \Rightarrow$$
$$\{S/m = \mathcal{E}\,\mathcal{E}',\ \text{где}\ \mathcal{E}' = S/m'\}\,\langle C, \mathcal{E} : \mathcal{E}' : S, D, E\rangle$$
$$\langle \texttt{REQ}\ m\ m' : C, S, D, E\rangle \ \Rightarrow$$
$$\{S/m \neq \mathcal{E}\,\mathcal{E}',\ \text{где}\ \mathcal{E}' = S/m'\}\,\langle \texttt{FAIL} : [\,], S, D, E\rangle$$

4.2.10.4 Перебор слева

Команды из этой группы отщепляют часть выражения с левого конца и устанавливают развилку. Исходное выражение при этом разбивается на части, которые заносятся в стек данных. Если при выполнении последующих команд возникает неуспех, он перехватывается и делается попытка удлинить отщепленную часть выражения.

Команде LE в программе обязательно должна предшествовать команда LEI m или LVI m.

$$\langle \texttt{LEI}\ m : \texttt{LE} : C, S, D, E\rangle \ \Rightarrow$$
$$\{S/m = \mathcal{E}\ \text{и}\ \mathcal{E} : \square : S = S'\}\,\langle C, S', (\texttt{LE} : C, S') : D, E\rangle$$
$$\langle \texttt{LVI}\ m : \texttt{LE} : C, S, D, E\rangle \ \Rightarrow\ \{S/m = \mathcal{E}\}\,\langle \texttt{LE} : C, \mathcal{E} : \square : S, D, E\rangle$$

$$\langle \texttt{LE}\ : C, \mathcal{T}\,\mathcal{E}'' : \mathcal{E}' : S, D, E\rangle \ \Rightarrow$$
$$\{\mathcal{E}'' : \mathcal{E}'\,\mathcal{T} : S = S'\}\,\langle C, S', (\texttt{LE} : C, S') : D, E\rangle$$
$$\langle \texttt{LE}\ : C, \square : \mathcal{E} : S, D, E\rangle \ \Rightarrow\ \langle \texttt{FAIL} : [\,], S, D, E\rangle$$

4.2.10.5 Перебор справа

Команды из этой группы отщепляют часть выражения с правого конца и устанавливают развилку. Исходное выражение при этом разбивается на части, которые заносятся в стек данных. Если при выполнении последующих команд возникает неуспех, он перехватывается и делается попытка удлинить отщепленную часть выражения.

Команде RE в программе обязательно должна предшествовать команда REI m или RVI m.

$$\langle \texttt{REI}\ m : \texttt{RE} : C, S, D, E\rangle \ \Rightarrow$$
$$\{S/m = \mathcal{E}\ \text{и}\ \mathcal{E} : \square : S = S'\}\,\langle C, S', (\texttt{RE} : C, S') : D, E\rangle$$
$$\langle \texttt{RVI}\ m : \texttt{RE} : C, S, D, E\rangle \ \Rightarrow\ \{S/m = \mathcal{E}\}\,\langle \texttt{RE} : C, \mathcal{E} : \square : S, D, E\rangle$$

$\langle \texttt{RE} : C, \mathcal{E}'' \mathcal{T} : \mathcal{E}' : S, D, E \rangle \;\Rightarrow$
$\quad \{\mathcal{E}'' : \mathcal{T}\, \mathcal{E}' : S = S'\}\, \langle C, S', (\texttt{RE} : C, S') : D, E \rangle$
$\langle \texttt{RE} : C, \square : \mathcal{E} : S, D, E \rangle\, \langle \texttt{FAIL} : [\,], S, D, E \rangle$

4.3 Компиляция Рефал-программ в виртуальный код

При описании алгоритма компиляции Рефал-программ в виртуальный код будем использовать обозначения близкие к тем, что обычно используются в книгах по денотационной семантике [Шмд 1986].

4.3.1 Абстрактный синтаксис

FD	$\in$	определения функций
$Palt$	$\in$	образцовые распутья
Snt	$\in$	предложения
Q	$\in$	тропы
R	$\in$	хвосты
S	$\in$	источники
P	$\in$	образцы
Pe	$\in$	образцовые выражения
Pt	$\in$	образцовые термы
He	$\in$	жесткие выражения
Ht	$\in$	жесткие термы
F, Fe	$\in$	форматные выражения (форматы)
Ft	$\in$	форматные термы
Re	$\in$	результатные выражения
Rt	$\in$	результатные термы
V	$\in$	переменные
Vs	$\in$	s-переменные
Vt	$\in$	t-переменные
Vv	$\in$	v-переменные
Ve	$\in$	e-переменные

$\mathcal{E}$ $\in$ объектные выражения
$\mathcal{T}$ $\in$ объектные термы
$\mathcal{S}$ $\in$ объектные символы
$\mathcal{F}$ $\in$ имена функций

FD ::= $\mathcal{F}\ Palt$;
$Palt$ ::= \{Snt_1; ... Snt_n;} | {Snt_1; ... Snt_n;}
Snt ::= $P\ R$
Q ::= $S\ R$ | S :: $He\ R$ | S'' $iter S' :: $He\ R$ | S : Snt | R | S
R ::= , Q | # $S\ R$ | \? Q | \! Q | $fail | = Q | $error Q | $trap Q $with $Palt$
S ::= \{Q_1; ... Q_n;} | {Q_1; ... Q_n;} | S : $Palt$ | Re
P ::= $l Pe | $r Pe
Pe ::= Pt_1 ... Pt_n
Pt ::= $\mathcal{S}$ | (Pe) | V
He ::= Hts | $Hts'\ Vv\ Hts''$ | $Hts'\ Ve\ Hts''$
Hts ::= Ht_1 ... Ht_n
Ht ::= $\mathcal{S}$ | (He) | Vs | Vt
Fe ::= Fts | Fts' v Fts'' | Fts' e Fts''
Fts ::= Ft_1 ... Ft_n
Ft ::= $\mathcal{S}$ | (Fe) | s | t
Re ::= Rt_1 ... Rt_n
Rt ::= $\mathcal{S}$ | (Re) | V | <$\mathcal{F}$ Re>

4.3.2 Функции связанные с форматами

$\mathcal{A}[\![He]\!]$ - список переменных, входящих в He (с сохранением их взаимного порядка.

$\mathcal{A}[\![\ ]\!] = [\,]$
$\mathcal{A}[\![\mathcal{S}\ He]\!] = \mathcal{A}[\![He]\!]$
$\mathcal{A}[\![V\ He]\!] = [V] \wedge \mathcal{A}[\![He]\!]$
$\mathcal{A}[\![(He')\ He]\!] = \mathcal{A}[\![He']\!] \wedge \mathcal{A}[\![He]\!]$

где $\wedge$ обозначает конкатенацию списков.

$\mathcal{F}[\![He]\!]$ - формат жесткого выражения He, который получается из He отбрасыванием индексов всех переменных.

$\mathcal{F}[\![\ ]\!] =$
$\mathcal{F}[\![\mathcal{S}\ He]\!] = \mathcal{S}\ \mathcal{F}[\![He]\!]$
$\mathcal{F}[\![Vs\ He]\!] =$ s $\mathcal{F}[\![He]\!]$
$\mathcal{F}[\![Vt\ He]\!] =$ t $\mathcal{F}[\![He]\!]$
$\mathcal{F}[\![Vv\ He]\!] =$ v $\mathcal{F}[\![He]\!]$
$\mathcal{F}[\![Ve\ He]\!] =$ e $\mathcal{F}[\![He]\!]$
$\mathcal{F}[\![(He')\ He]\!] = (\ \mathcal{F}[\![He']\!]\ )\ \mathcal{F}[\![He]\!]$

4.3.3 Компиляция результатных выражений

Предположим, что нам требуется скомпилировать результатное выражение Re. В общем случае это выражение может содержать символы, скобки, переменные и вызовы функций. Кроме того, результат вычисления этого выражения должен удовлетворять некоторому формату $Fout$.

Мы начнем с рассмотрения более простой задачи. Будем предполагать, что Re не содержит вызовы функций и что его требуется вычислить следующим образом: вместо всех переменных подставить их значения, а получившееся объектное выражение поместить на вершину стека. При этом мы считаем, что значения всех переменных, входящих в Re уже известны и находятся на стеке.

Таким образом, для компиляции Re нам требуется знать следующую дополнительную информацию: среду ρ, связывающую переменные с их позициями в стеке и число d - номер последней занятой позиции в стеке (если стек пуст, то $d = -1$). При этом мы считаем, что позиции в стеке нумеруются от дна к вершине начиная с нуля. Следовательно, если стек содержит n объектных выражений и имеет вид

$$\mathcal{E}_{n-1} : \ldots : \mathcal{E}_1 : \mathcal{E}_0 : [\,]$$

то содержимым позиции i является объектное выражение $\mathcal{E}_i$.

Таким образом, $\rho(V)$ является позицией в стеке, в которой находится значение переменной V, а $(d - \rho(V))$ - расстояние от вершины стека до этой позиции.

Если заданы Re, ρ и d, то результат компиляции результатного выражения Re мы будем обозначать через $\mathbf{Re1}[\![Re]\!]\,\rho\,d$. Компилирующая функция **Re1** определяются следующими соотношениями:

$$\begin{array}{lcl}
\mathbf{Re1}[\![\,]\!]\ \rho\ d & = & \texttt{PUSHQ}\ \square \\
\mathbf{Re1}[\![Re\ Rt]\!]\ \rho\ d & = & \mathbf{Re1}[\![Re]\!]\ \rho\ d;\ \mathbf{Rt1}[\![Rt]\!]\ \rho\ (d+1);\ \texttt{CONC}
\end{array}$$

$$\begin{array}{lcl}
\mathbf{Rt1}[\![\mathcal{S}]\!]\ \rho\ d & = & \texttt{PUSHQ}\ \mathcal{S} \\
\mathbf{Rt1}[\![V]\!]\ \rho\ d & = & \texttt{PUSH}\ (d - \rho(V)) \\
\mathbf{Rt1}[\![(Re)]\!]\ \rho\ d & = & \mathbf{Re1}[\![Re]\!]\ \rho\ d;\ \texttt{BR}
\end{array}$$

Конечно, это не самый оптимальный способ компиляции, и позднее мы обсудим, возможные усовершенствования.

Теперь рассмотрим общий случай результатного выражения Re, содержащего вызовы функций. Будем считать, что задан формат $Fout$, которому должен удовлетворять результат вычисления этого выражения, и число r - позиция в стеке, куда следует поместить результат вычисления Re. При этом, если $Fout$ содержит b переменных, результатом вычисления Re должны быть b объектных выражений, соответствующих переменным формата. Эти выражения должны быть размещены в стеке в позициях r, $r+1$, $\ldots$, $r+b-1$.

Результат компиляции результатного выражения Re может быть описан с помощью функции **Re**, обращение к которой имеют следующий вид:

$$\mathbf{Re}[\![Re]\!]\ [\![Fout]\!]\ r\ \rho\ d$$

где ρ и d имеют тот же смысл, как и в случае функции **Re1**.

Компиляция Re состоит из следующих этапов.

Первым делом из Re извлекаются все вызовы функций и генерируется код, обеспечивающий вычисление этих вызовов. Сами вызовы при этом удаляются из Re и заменяются на выходные форматы этих функций. При этом переменные, входящие в форматы, снабжаются индексами, в результате чего они превращаются в жесткие выражения. Индексы переменных выбираются произвольно, но так, чтобы не возникло конфликтов с переменными из $\mathrm{dom}\,\rho$. (Эти новые переменные получают значения в результате вычисления вызовов функций в Re.) Затем, информация о новых переменных заносится в среду ρ.

Вся эта деятельность осуществляется компилирующей функцией **ExtractCallsRe**, обращение к которой имеет вид **ExtractCallsRe**$[\![Re]\!]$ ρ d, а результат работы имеет вид $(Ccalls, Re', \rho', d')$, где $Ccalls$ - виртуальный код, производящий вычисление вызовов функций из Re, Re' - результатное выражение, которое получается из Re в результате удаления вызовов функций, ρ' - среда ρ, пополненная сведениями о новых переменных и d' - значение d, исправленное с учетом того, что в результате исполнения кода $Ccalls$, в стеке могут оказаться занятыми дополнительные позиции.

Интересно отметить, что функция **ExtractCallsRe** рекурсивно обращается к функции Re, поскольку вычисление вызовов функций требует вычисления их аргументных выражений.

После извлечения вызовов функций из Re, получается результатное выражение Re', которое является частным случаем формата $Fout$ и может быть разбито на b результатных выражений, каждое из которых дает значение для одной из переменных формата $Fout$. Это делается с помощью функции **SplitRe**, обращение к которой имеет вид **SplitRe**$[\![Fout]\!]$ $[\![Re']\!]$, а результатом является список из b результатных выражений, каждое из которох теперь может быть скомпилировано функцией **Re1**.

И наконец, остается сгенерировать команду `SQUEEZE`, сдвигающую b

результатов в позицию r.

$$
\begin{array}{l}
\mathbf{Re}[\![Re]\!]\ [\![F]\!]\ r\ \rho\ d = \\
\quad \mathbf{let\ val}\ (Ccalls, Re', \rho', d') = \mathbf{ExtractCallsRe}[\![Re]\!]\ \rho\ d \\
\qquad \mathbf{val}\ [Re_1, \ldots, Re_b] = \mathbf{SplitRe}[\![F]\!]\ [\![Re']\!] \\
\quad \mathbf{in} \\
\qquad Ccalls; \\
\qquad \mathbf{Re1}[\![Re_1]\!]\ \rho'\ d'; \\
\qquad \ldots \\
\qquad \mathbf{Re1}[\![Re_b]\!]\ \rho'\ (d' + b - 1); \\
\qquad \texttt{SQUEEZE}\ b\ (d' + 1 - r) \\
\quad \mathbf{end}
\end{array}
$$

$$
\mathbf{ExtractCallsRe}[\![\,]\!]\ \rho\ d = (\texttt{NOP}, \Box, \rho, d)
$$

$$
\begin{array}{l}
\mathbf{ExtractCallsRe}[\![RtRe]\!]\ \rho\ d = \\
\quad \mathbf{let\ val}\ (C', Re', \rho', d') = \mathbf{ExtractCallsRt}[\![Rt]\!]\ \rho\ d \\
\qquad \mathbf{val}\ (C'', Re'', \rho'', d'') = \mathbf{ExtractCallsRe}[\![Re]\!]\ \rho'\ d' \\
\quad \mathbf{in}\ ((C'; C''),\ Re'\,Re'',\ \rho'',\ d'')\ \mathbf{end}
\end{array}
$$

$$
\mathbf{ExtractCallsRt}[\![\mathcal{S}]\!]\ \rho\ d = (\texttt{NOP}, \mathcal{S}, \rho, d)
$$

$$
\mathbf{ExtractCallsRt}[\![V]\!]\ \rho\ d = (\texttt{NOP}, V, \rho, d)
$$

$$
\begin{array}{l}
\mathbf{ExtractCallsRt}[\![(Re)]\!]\ \rho\ d = \\
\quad \mathbf{let\ val}\ (C', Re', \rho', d') = \mathbf{ExtractCallsRe}[\![Re]\!]\ \rho\ d \\
\quad \mathbf{in}\ (C',\ (Re'),\ \rho',\ d')\ \mathbf{end}
\end{array}
$$

$$
\begin{array}{l}
\mathbf{ExtractCallsRt}[\![<\mathcal{F}\ Re>]\!]\ \rho\ d = \\
\quad \mathbf{let\ val}\ Fin = \text{ входной формат функции } \mathcal{F} \\
\qquad \mathbf{val}\ Fout = \text{ выходной формат функции } \mathcal{F} \\
\qquad \mathbf{val}\ He = \text{ такое жесткое выражение, что} \\
\qquad\qquad Fout = \mathcal{F}[\![He]\!] \text{ и } \mathrm{dom}\,\rho \cap \mathcal{V}[\![He]\!] = \emptyset \\
\qquad \mathbf{val}\ [V'_1, \ldots, V'_a] = \mathcal{A}[\![Fin]\!] \\
\qquad \mathbf{val}\ [V_1, \ldots, V_b] = \mathcal{A}[\![He]\!] \\
\quad \mathbf{in} \\
\qquad (\mathbf{Re}[\![Re]\!]\ [\![Fin]\!]\ (d+1)\ \rho\ d;\ \texttt{CALL}\ a\ \mathcal{F}, \\
\qquad\quad He,\ \rho + \{V_1 = d+1, \ldots, V_b = d+b\},\ d+b) \\
\quad \mathbf{end}
\end{array}
$$

$$
\begin{array}{lcl}
\mathbf{SplitRe}[\![Ft\,Fe]\!]\ [\![Rt\,Re]\!] & = & \mathbf{SplitRt}[\![Ft]\!]\ [\![Rt]\!] \,^\wedge\, \mathbf{SplitRe}[\![Fe]\!]\ [\![Re]\!] \\
\mathbf{SplitRe}[\![Fe\,Ft]\!]\ [\![Re\,Rt]\!] & = & \mathbf{SplitRe}[\![Fe]\!]\ [\![Re]\!] \,^\wedge\, \mathbf{SplitRt}[\![Ft]\!]\ [\![Rt]\!] \\
\mathbf{SplitRe}[\![\mathtt{v}]\!]\ [\![Re]\!] & = & [Re] \\
\mathbf{SplitRe}[\![\mathtt{e}]\!]\ [\![Re]\!] & = & [Re] \\
\mathbf{SplitRe}[\![\,]\!]\ [\![\,]\!] & = & [\,]
\end{array}
$$

$$\begin{array}{lcl}
\mathbf{SplitRt}[\![\mathcal{S}]\!]\ [\![\mathcal{S}]\!] & = & [\,] \\
\mathbf{SplitRt}[\![\mathtt{s}]\!]\ [\![Rt]\!] & = & [Rt] \\
\mathbf{SplitRt}[\![\mathtt{t}]\!]\ [\![Rt]\!] & = & [Rt] \\
\mathbf{SplitRt}[\![(Fe)]\!]\ [\![(Re)]\!] & = & \mathbf{SplitRe}[\![Fe]\!]\ [\![Re]\!]
\end{array}$$

4.3.4 Генерация ошибок

В процессе работы рефал-программы во многих случаях может вырабатываться ошибка вида

$\mathtt{\$error}(\mathcal{F}$ `"Unexpected fail"`$)$

где $\mathcal{F}$ - имя функции, в которой возникла ошибка. Поэтому, результат компиляции каждой функции $\mathcal{F}$ содержит фрагмент виртуального кода вида

`LABEL` *Lerror*;
`PUSHQ` $\mathcal{F}$ `"Unexpected fail"; ERROR`

где *Lerror* - некоторая метка (для каждой функции - своя). Таким образом, чтобы сгенерировать ошибку, достаточно передать управление на *Lerror*.

Метка *Lerror* используется многими компилирующими функциями, и ее следовало бы (ради "идеологической чистоты") передавать этим функциям в качестве параметра. Тем не менее, чтобы не загромождать изложение, мы этого делать не будем, а будем считать, что *Lerror* - глобальная величина, которая не меняется на протяжении компиляции одного определения функции.

4.3.5 Компиляция троп, хвостов и источников

Результат компиляции троп, хвостов и источников может быть описан с помощью функций **Q**, **R** и **S** соответственно. Обращения к этим функциям имеют следующий вид:

$\mathbf{Q}[\![Q]\!]\ [\![Fout]\!]\ r\ k\ \rho\ d$
$\mathbf{R}[\![R]\!]\ [\![Fout]\!]\ r\ k\ \rho\ d$
$\mathbf{S}[\![S]\!]\ [\![Fout]\!]\ r\ k\ \rho\ d$

где *Fout*, r, ρ и d имеют тот же смысл, что и в случае функции *Re*, а k представляет собой *стек альтернатив*, содержимое которого всегда имеет следующий вид:

$[h_1 : \ldots : h_n]$

где $h_1, \ldots, h_n$ - целые неотрицательные числа, и $n \geq 1$. Стек k служит для подсчета сгенерированных команд `ALT`, и используется при генерации команд `CUT`.

А именно, если стек альтернатив имеет вид $h : k$ и порождается команда `ALT`, к его вершине прибавляется 1, и он принимает вид $(h+1) : k$.

Появление и исчезновение элементов в стеке альтернатив связано с заборами и отсечениями. А именно, если при компиляции тропы встретился забор `\?`, а стек альтернатив имеет вид k, то в стеке заводится новый элемент, и он принимает вид $0 : k$. Если же при компиляции тропы встречается отсечение `\!`, а стек имеет вид $h : k$, то генерируется команда `CUT` h, а стек принимает вид k. Благодаря этому, все альтернативы, скопившиеся после последнего забора `\?`, отменяются.

Стек альтернатив никогда не бывает пустым, поскольку в нем ведется учет команд `ALT`, сгенерированных "на нулевом уровне", т.е. еще до того как встретился первый забор, или после "закрытия" всех заборов.

Если встречается правая часть, т.е. тропа вида `=` Q, требуется отменить все альтернативы (даже те, что находятся на нулевом уровне). Поэтому, если стек альтернатив имеет вид $[h_1, \ldots, h_n]$, генерируется команда `CUT` $(h_1 + \ldots + h_n)$, а стек альтернатив принимает вид [0]. Таким образом, на всей предыстории "ставится крест" и начинается "новая жизнь".

При компиляции троп, хвостов и источников используются функции **Snt** и **Palt**, которые служат для компиляции предложений и образцовых распутий. Эти компилирующие функции описаны в разделах, посвященных компиляции предложений и образцовых распутий.

4.3.6 Компиляция троп

$$\begin{array}{l}
\mathbf{Q}[\![S\ R]\!]\ [\![F]\!]\ r\ k\ \rho\ d = \\
\quad \mathbf{S}[\![S]\!]\ [\,]\ (d+1)\ [0]\ \rho\ d;\ \mathbf{R}[\![R]\!]\ [\![F]\!]\ r\ k\ \rho\ d
\end{array}$$

$$\begin{array}{l}
\mathbf{Q}[\![S :: He\,R]\!]\ [\![F]\!]\ r\ k\ \rho\ d = \\
\quad \textbf{let val } [V_1, \ldots, V_a] = \mathcal{A}[\![He]\!] \\
\qquad \textbf{val } \rho' = \rho + \{V_1 = d+1, \ldots, V_a = d+a\} \\
\quad \textbf{in} \\
\qquad \mathbf{S}[\![S]\!]\ (\mathcal{F}[\![He]\!])\ (d+1)\ [0]\ \rho\ d; \\
\qquad \mathbf{R}[\![R]\!]\ [\![F]\!]\ r\ k\ \rho'\ (d+a) \\
\quad \textbf{end}
\end{array}$$

$$\mathbf{Q}[\![S''\ \texttt{\$iter}\ S' :: He\ R]\!]\ [\![F]\!]\ r\ (h:k)\ \rho\ d =$$
$$\quad \textbf{let val } [V_1,\ldots,V_a] = \mathcal{A}[\![He]\!]$$
$$\quad\quad \textbf{val } \rho' = \rho + \{V_1 = d+1,\ldots,V_a = d+a\}$$
$$\quad \textbf{in}$$
$$\quad\quad \mathbf{S}[\![S'']\!]\ (\mathcal{F}[\![He]\!])\ (d+1)\ [0]\ \rho\ d;\ \texttt{JUMP}\ L2;$$
$$\quad\quad \texttt{LABEL}\ L1;$$
$$\quad\quad\quad \mathbf{S}[\![S']\!]\ (\mathcal{F}[\![He]\!])\ (d+1)\ [0]\ \rho'\ (d+a);$$
$$\quad\quad \texttt{LABEL}\ L2;$$
$$\quad\quad \texttt{ALT}\ L1;$$
$$\quad\quad\quad \mathbf{R}[\![R]\!]\ [\![F]\!]\ r\ ((h+1):k)\ \rho'\ (d+a)$$
$$\quad \textbf{end}$$

$$\mathbf{Q}[\![S\ :\ Snt]\!]\ [\![F]\!]\ r\ k\ \rho\ d =$$
$$\quad \mathbf{S}[\![S]\!]\ [\![e]\!]\ (d+1)\ [0]\ \rho\ d;\ \mathbf{Snt}[\![Snt]\!]\ [\![e]\!]\ [\![F]\!]\ r\ k\ \rho\ (d+1)$$

$$\mathbf{Q}[\![R]\!]\ [\![F]\!]\ r\ k\ \rho\ d = \mathbf{R}[\![R]\!]\ [\![F]\!]\ r\ k\ \rho\ d$$

$$\mathbf{Q}[\![S]\!]\ [\![F]\!]\ r\ k\ \rho\ d = \mathbf{S}[\![S]\!]\ [\![F]\!]\ r\ k\ \rho\ d$$

4.3.7 Компиляция хвостов

$$\mathbf{R}[\![,\ Q]\!]\ [\![F]\!]\ r\ k\ \rho\ d = \mathbf{Q}[\![Q]\!]\ [\![F]\!]\ r\ k\ \rho\ d$$

$$\mathbf{R}[\![\texttt{\#}\ S\ R]\!]\ [\![F]\!]\ r\ k\ \rho\ d =$$
$$\quad \texttt{ALT}\ L1;$$
$$\quad\quad \mathbf{S}[\![S]\!]\ [\![\,]\!]\ (d+1)\ [0]\ \rho\ d;$$
$$\quad\quad \texttt{CUT}\ 1;\ \texttt{FAIL};$$
$$\quad \texttt{LABEL}\ L1;$$
$$\quad\quad \mathbf{R}[\![R]\!]\ [\![F]\!]\ r\ k\ \rho\ d$$

$$\mathbf{R}[\![\backslash\texttt{?}\ Q]\!]\ [\![F]\!]\ r\ k\ \rho\ d = \mathbf{Q}[\![Q]\!]\ [\![F]\!]\ r\ (0:k)\ \rho\ d$$

$$\mathbf{R}[\![\backslash\texttt{!}\ Q]\!]\ [\![F]\!]\ r\ (h:k)\ \rho\ d = \texttt{CUT}\ h;\ \mathbf{Q}[\![Q]\!]\ [\![F]\!]\ r\ k\ \rho\ d$$

$$\mathbf{R}[\![\texttt{\$fail}]\!]\ [\![F]\!]\ r\ k\ \rho\ d = \texttt{FAIL}$$

$$\mathbf{R}[\![\texttt{=}\ Q]\!]\ [\![F]\!]\ r\ [h_1,\ldots,h_n]\ \rho\ d =$$
$$\quad \texttt{CUT}\ (h_1+\ldots+h_n);\ \mathbf{Q}[\![Q]\!]\ [\![F]\!]\ r\ [0]\ \rho\ d$$

$$\mathbf{R}[\![\texttt{\$error}\ Q]\!]\ [\![F]\!]\ r\ k\ \rho\ d =$$
$$\quad \texttt{ALT}\ Lerror;$$
$$\quad\quad \mathbf{Q}[\![Q]\!]\ [\![e]\!]\ (d+1)\ [0]\ \rho\ d;$$
$$\quad\quad \texttt{ERROR}$$

$\mathbf{R}[\![\texttt{\$trap}\ Q\ \texttt{\$with}\ Palt]\!]\ [\![F]\!]\ r\ [h_1,\ldots,h_n]\ \rho\ d =$
$\texttt{PUSHTRAP}\ L1;\ \texttt{ALT}\ Lerror;$
$\quad \mathbf{Q}[\![Q]\!]\ [\![F]\!]\ r\ [0]\ \rho\ d;$
$\quad \texttt{CUT}\ (1+h_1+\ldots+h_n);\ \texttt{POPTRAP};\ \texttt{JUMP}\ L0;$
$\texttt{LABEL}\ L1;$
$\quad \mathbf{Palt}[\![Palt]\!]\ [\![e]\!]\ [\![F]\!]\ r\ [h_1,\ldots,h_n]\ \rho\ (d+1);$
$\texttt{LABEL}\ L0$

4.3.8 Компиляция источников

$\mathbf{S}[\![\backslash\{Q_1;\ \ldots\ Q_n;\ Q_0;\}]\!]\ [\![F]\!]\ r\ (h:k)\ \rho\ d =$
$\texttt{ALT}\ L1;$
$\quad \mathbf{Q}[\![Q_1]\!]\ [\![F]\!]\ r\ ((h+1):k)\ \rho\ d;\ \texttt{JUMP}\ L0;$
$\texttt{LABEL}\ L1;$
$\quad \mathbf{S}[\![\backslash\{\ \ldots\ Q_n;\ Q_0;\}]\!]\ [\![F]\!]\ r\ (h:k)\ \rho\ d;$
$\texttt{LABEL}\ L0$

$\mathbf{S}[\![\backslash\{Q_0;\}]\!]\ [\![F]\!]\ r\ k\ \rho\ d = \mathbf{Q}[\![Q_0]\!]\ [\![F]\!]\ r\ k\ \rho\ d$

$\mathbf{S}[\![\backslash\{\}]\!]\ [\![F]\!]\ r\ k\ \rho\ d = \texttt{FAIL}$

$\mathbf{S}[\![\{Q_1;\ \ldots\ Q_n;\ Q_0;\}]\!]\ [\![F]\!]\ r\ (h:k)\ \rho\ d =$
$\texttt{ALT}\ L1;$
$\quad \mathbf{Q}[\![Q_1]\!]\ [\![F]\!]\ r\ ((h+1):k)\ \rho\ d;\ \texttt{JUMP}\ L0;$
$\texttt{LABEL}\ L1;$
$\quad \mathbf{S}[\![\{\ \ldots\ Q_n;\ Q_0;\}]\!]\ [\![F]\!]\ r\ (h:k)\ \rho\ d;$
$\texttt{LABEL}\ L0$

$\mathbf{S}[\![\{Q_0;\}]\!]\ [\![F]\!]\ r\ (h:k)\ \rho\ d =$
$\texttt{ALT}\ Lerror;$
$\quad \mathbf{Q}[\![Q_0]\!]\ [\![F]\!]\ r\ ((h+1):k)\ \rho\ d$

$\mathbf{S}[\![\{\}]\!]\ [\![F]\!]\ r\ k\ \rho\ d = \texttt{JUMP}\ Lerror$

$\mathbf{S}[\![S:Palt]\!]\ [\![F]\!]\ r\ k\ \rho\ d =$
$\quad \mathbf{S}[\![S]\!]\ [\![e]\!]\ (d+1)\ [0]\ \rho\ d;$
$\quad \mathbf{Palt}[\![Palt]\!]\ [\![e]\!]\ [\![F]\!]\ r\ k\ \rho\ (d+1)$

$\mathbf{S}[\![Re]\!]\ [\![F]\!]\ r\ [h_1,\ldots,h_n]\ \rho\ d =$
$\quad \mathbf{Re}[\![Re]\!]\ [\![F]\!]\ r\ \rho\ d;\ \texttt{CUT}\ (h_1+\ldots+h_n)$

4.3.9 Компиляция предложений

Результат компиляции предложений может быть описан с помощью функции Snt, обращение к которой имеет следующий вид:

$\mathbf{Snt}[\![Snt]\!]\ [\![Fin]\!]\ [\![Fout]\!]\ r\ k\ \rho\ d$

где $Fout$, r, k, ρ и d имеют тот же смысл, что и в случае функций **Q**, **R** и **S**, а Fin - входной формат предложения.

Входной формат Fin описывает структуру объектных выражений, которые должны сопоставляться со входным образцом предложения. Переменные, входящие в Fin, соответствуют тем компонентам объектных выражений, которые могут меняться. Если Fin содержит a переменных, а d - номер последней занятой позиции в стеке, то перед началом исполнения предложения компоненты объектного выражения должны быть помещены в стек в позиции $d-a+1, \ldots, d$.

Компиляция предложения **Snt**, имеющего вид `$1` Pe R происходит следующим образом.

Первым делом образцовое выражение Pe разбивается на a образцовых выражений $Pe_1, \ldots, Pe_a$, каждое из которых соответствует одной из переменных, входящих в Fin. А именно, образцовому выражению Pe_j соответствует переменная формата V_j. При этом, компонента объектного выражения, соответствующая V_j должна находиться в стеке в позиции $d-a+j$. Поэтому, создается список упорядоченных троек $(d-a+j, V_j, Pe_j)$. Для большей легкости зрительного восприятия мы будем записывать этот список следующим образом:

$$[d-a+1 : V_1 : Pe_1] \ \ldots \ [d : V_a : Pe_a]$$

Таким образом, тройка $(d-a+j, V_j, Pe_j)$ записывается в виде $[d-a+j : V_j : Pe_j]$, а список этих троек записывается без наружных квадратных скобок и без разделяющих запятых.

Этот список затем обрабатывается функцией **PeInit**, целью которой является преобразовать тройки вида $[i : V : Pe]$ в пары вида $[i : Pe]$, где каждая пара $[i : Pe]$ означает, что образец Pe должен сопоставляться с объектным выражением, хранящимся в стеке в позиции i.

Функция $PeInit$ анализирует каждую тройку $[i : V : Pe]$. Если оказывается, что Pe - переменная, значение которой еще неизвестно (т.е. Pe не входит в область определения среды ρ), и при этом Pe имеет тот же тип, что и V - это означает, что Pe заведомо успешно сопоставляется с объектным выражением, хранящимся в позиции i. Поэтому тройка $[i : V : Pe]$ удаляется из списка, а среда ρ преобразуется в среду $\rho + \{Pe = i\}$. Если же одно из вышеуказанных условий нарушено, это означает, что для сопоставления Pe с объектным выражением заведомо требуется выполнять какие-то действия, поэтому тройка $[i : V : Pe]$ преобразуется в пару $[i : Pe]$.

Таким образом, результат работы функции **PeInit** имеет вид

$$[i_1 : Pe_1] \ \ldots \ [i_m : Pe_m]$$

и передается для дальнейшей обработки функции **PeL**, которая осуществляет компиляцию образцов при условии, что отождествление должно выполняться слева направо.

Случай отождествления справа налево "зеркально симметричен" случаю отождествления слева направо, поэтому мы его в дальнейшем рассматривать не будем.

$$
\begin{array}{l}
\mathbf{Snt}[\![\texttt{\$l}\ Pe\ R]\!]\ [\![Fin]\!]\ [\![Fout]\!]\ r\ k\ \rho\ d = \\
\quad \mathbf{let\ val}\ [V_1, \ldots, V_a] = \mathcal{A}[\![Fin]\!] \\
\qquad \mathbf{val}\ [Pe_1, \ldots, Pe_a] = \mathbf{SplitPe}[\![Fin]\!]\ [\![Pe]\!] \\
\qquad \mathbf{val}\ (\Delta, \rho') = \mathbf{PeInit}[\![\,]\!]\ [\![[d - a + 1 : V_1 : Pe_1]\ \ldots\ [d : V_a : Pe_a]]\!]\ \rho \\
\quad \mathbf{in\ PeL}[\![\Delta]\!]\ (\mathbf{R}[\![R]\!]\ [\![Fout]\!]\ r)\ k\ \rho'\ d\ \mathbf{end}
\end{array}
$$

$$
\mathbf{Snt}[\![\texttt{\$r}\ Pe\ R]\!]\ [\![Fin]\!]\ [\![Fout]\!]\ r\ k\ \rho\ d = \ldots
$$

$$
\mathbf{PeInit}[\![\Delta]\!]\ [\![\,]\!]\ \rho = (\Delta, \rho)
$$

$$
\begin{array}{l}
\mathbf{PeInit}[\![\Delta]\!]\ [\![[i : V : Pe]\ \Phi]\!]\ \rho = \\
\quad \mathbf{if}\ \mathcal{F}[\![Pe]\!] = V\ \mathbf{and}\ Pe \notin \mathrm{dom}\,\rho \\
\quad \mathbf{then\ PeInit}[\![\Delta]\!]\ [\![\Phi]\!]\ (\rho + \{Pe = i\}) \\
\quad \mathbf{else\ PeInit}[\![\Delta\ [i : Pe]]\!]\ [\![\Phi]\!]\ \rho
\end{array}
$$

SplitPe$[\![F]\!]\ [\![Pe]\!]$ — разбиение Pe на список образцов в соответствии с форматом F.

$$
\begin{array}{lcl}
\mathbf{SplitPe}[\![\,]\!]\ [\![\,]\!] & = & [\,] \\
\mathbf{SplitPe}[\![Ft\ Fe]\!]\ [\![Pt\ Pe]\!] & = & \mathbf{SplitPt}[\![Ft]\!]\ [\![Pt]\!] \mathbin{^\wedge} \mathbf{SplitPe}[\![Fe]\!]\ [\![Pe]\!] \\
\mathbf{SplitPe}[\![Fe\ Ft]\!]\ [\![Pe\ Pt]\!] & = & \mathbf{SplitPe}[\![Fe]\!]\ [\![Pe]\!] \mathbin{^\wedge} \mathbf{SplitPt}[\![Ft]\!]\ [\![Pt]\!] \\
\mathbf{SplitPe}[\![v]\!]\ [\![Pe]\!] & = & [Pe] \\
\mathbf{SplitPe}[\![e]\!]\ [\![Pe]\!] & = & [Pe]
\end{array}
$$

$$
\begin{array}{lcl}
\mathbf{SplitPt}[\![\mathcal{S}]\!]\ [\![\mathcal{S}]\!] & = & [\,] \\
\mathbf{SplitPt}[\![s]\!]\ [\![Pt]\!] & = & [Pt] \\
\mathbf{SplitPt}[\![t]\!]\ [\![Pt]\!] & = & [Pt] \\
\mathbf{SplitPt}[\![(Fe)]\!]\ [\![(Pe)]\!] & = & \mathbf{SplitPe}[\![Fe]\!]\ [\![Pe]\!]
\end{array}
$$

4.3.10 Компиляция образцов

Компиляция образцовых выражений (в случае отождествления слева направо) производится функцией **PeL**, обращение к которой имеет следующий вид:

$$
\mathbf{PeL}[\![\Delta]\!]\ c\ k\ \rho\ d
$$

где k, ρ и d имеют тот же смысл, что и в случае функции **Snt**, Δ имеет вид $[i_1 : Pe_1]\ \ldots\ [i_m : Pe_m]$, а c является "продолжением", т.е. функцией, которую надо вызвать после завершения компиляции всех образцовых выражений из Δ, передав ей k, ρ и d.

Определение функции PeL содержит правила вида

$$\mathbf{PeL}[\![\Delta]\!]\ c\ k\ \rho\ d\ \parallel\ C = \ldots$$

где C - некоторое условие. Такая запись означает, что данное правило применимо только если выполнено дополнительное условие C.

Во многих случаях оказывается, что можно применить сразу несколько правил из описания функции **PeL**. В таких случаях делается выбор одного из правил (либо произвольно, либо из каких-то дополнительных соображений). Конечно, во всяком реальном компиляторе этот выбор делается однозначно, и может диктоваться соображениями оптимизации.

Все правила распадаются на несколько групп.

4.3.11 Завершение компиляции образцовых выражений

$$\mathbf{PeL}[\![\ ]\!]\ c\ k\ \rho\ d = c\ k\ \rho\ d$$

4.3.12 "Закрытые" с двух сторон элементы образцов

$$\mathbf{PeL}[\![\Gamma\ [i :]\ \Delta]\!]\ c\ k\ \rho\ d =$$
$$\texttt{EMPTY}\ (d - i);\ \mathbf{PeL}[\![\Gamma\ \Delta]\!]\ c\ k\ \rho\ d$$

$$\mathbf{PeL}[\![\Gamma\ [i : (Pe)]\ \Delta]\!]\ c\ k\ \rho\ d =$$
$$\texttt{CB}\ (d - i);\ \mathbf{PeL}[\![\Gamma\ [d + 1 : Pe]\ \Delta]\!]\ c\ k\ \rho\ (d + 1)$$

$$\mathbf{PeL}[\![\Gamma\ [i : \mathcal{S}]\ \Delta]\!]\ c\ k\ \rho\ d =$$
$$\texttt{CQ}\ (d - i)\ \mathcal{S};\ \mathbf{PeL}[\![\Gamma\ \Delta]\!]\ c\ k\ \rho\ d$$

$$\mathbf{PeL}[\![\Gamma\ [i : V]\ \Delta]\!]\ c\ k\ \rho\ d\ \parallel\ V \in \mathrm{dom}\,\rho =$$
$$\texttt{CEQ}\ (d - i)\ (d - \rho(V));\ \mathbf{PeL}[\![\Gamma\ \Delta]\!]\ c\ k\ \rho\ d$$

$$\mathbf{PeL}[\![\Gamma\ [i : Vs]\ \Delta]\!]\ c\ k\ \rho\ d\ \parallel\ Vs \notin \mathrm{dom}\,\rho =$$
$$\texttt{CS}\ (d - i);\ \mathbf{PeL}[\![\Gamma\ \Delta]\!]\ c\ k\ (\rho + \{Vs = i\})\ d$$

$$\mathbf{PeL}[\![\Gamma\ [i : Vt]\ \Delta]\!]\ c\ k\ \rho\ d\ \parallel\ Vt \notin \mathrm{dom}\,\rho =$$
$$\texttt{CT}\ (d - i);\ \mathbf{PeL}[\![\Gamma\ \Delta]\!]\ c\ k\ (\rho + \{Vt = i\})\ d$$

$$\mathbf{PeL}[\![\Gamma\ [i : Vv]\ \Delta]\!]\ c\ k\ \rho\ d\ \parallel\ Vv \notin \mathrm{dom}\,\rho =$$
$$\texttt{CV}\ (d - i);\ \mathbf{PeL}[\![\Gamma\ \Delta]\!]\ c\ k\ (\rho + \{Vv = i\})\ d$$

$$\mathbf{PeL}[\![\Gamma\ [i : Ve]\ \Delta]\!]\ c\ k\ \rho\ d\ \parallel\ Ve \notin \mathrm{dom}\,\rho =$$
$$\mathbf{PeL}[\![\Gamma\ \Delta]\!]\ c\ k\ (\rho + \{Ve = i\})\ d$$

4.3.12.1 "Жесткие" элементы слева

$$\mathbf{PeL}[\![\Gamma\ [i : (Pe')\,Pe''\,]\ \Delta]\!]\ c\ k\ \rho\ d\ \parallel\ Pe'' \neq \square =$$
$$\texttt{LB}\ (d-i);\ \mathbf{PeL}[\![\Gamma\ [d+3 : Pe'][d+1 : Pe'']\ \Delta]\!]\ c\ k\ \rho\ (d+3)$$

$$\mathbf{PeL}[\![\Gamma\ [i : \mathcal{S}\,Pe]\ \Delta]\!]\ c\ k\ \rho\ d\ \parallel\ Pe \neq \square =$$
$$\texttt{LQ}\ (d-i)\ \mathcal{S};\ \mathbf{PeL}[\![\Gamma\ [d+2 : Pe]\ \Delta]\!]\ c\ k\ \rho\ (d+2)$$

$$\mathbf{PeL}[\![\Gamma\ [i : V\,Pe]\ \Delta]\!]\ c\ k\ \rho\ d\ \parallel\ V \in \mathrm{dom}\,\rho, Pe \neq \square =$$
$$\texttt{LEQ}\ (d-i)\ (d-\rho(V));\ \mathbf{PeL}[\![\Gamma\ [d+2 : Pe]\ \Delta]\!]\ c\ k\ \rho\ (d+2)$$

$$\mathbf{PeL}[\![\Gamma\ [i : Vs\,Pe]\ \Delta]\!]\ c\ k\ \rho\ d\ \parallel\ Vs \notin \mathrm{dom}\,\rho,\ Pe \neq \square =$$
$$\texttt{LS}\ (d-i);\ \mathbf{PeL}[\![\Gamma\ [d+2 : Pe]\ \Delta]\!]\ c\ k\ (\rho + \{Vs = d+1\})\ (d+2)$$

$$\mathbf{PeL}[\![\Gamma\ [i : Vt\,Pe]\ \Delta]\!]\ c\ k\ \rho\ d\ \parallel\ Vt \notin \mathrm{dom}\,\rho,\ Pe \neq \square =$$
$$\texttt{LT}\ (d-i);\ \mathbf{PeL}[\![\Gamma\ [d+2 : Pe]\ \Delta]\!]\ c\ k\ (\rho + \{Vt = d+1\})\ (d+2)$$

4.3.13 "Жесткие" элементы справа

$$\mathbf{PeL}[\![\Gamma\ [i : Pe''\,(Pe')]\ \Delta]\!]\ c\ k\ \rho\ d\ \parallel\ Pe'' \neq \square =$$
$$\texttt{RB}\ (d-i);\ \mathbf{PeL}[\![\Gamma\ [d+1 : Pe'']\ [d+3 : Pe']\ \Delta]\!]\ c\ k\ \rho\ (d+3)$$

$$\mathbf{PeL}[\![\Gamma\ [i : Pe\,\mathcal{S}]\ \Delta]\!]\ c\ k\ \rho\ d\ \parallel\ Pe \neq \square =$$
$$\texttt{RQ}\ (d-i)\ \mathcal{S};\ \mathbf{PeL}[\![\Gamma\ [d+2 : Pe]\ \Delta]\!]\ c\ k\ \rho\ (d+2)$$

$$\mathbf{PeL}[\![\Gamma\ [i : Pe\,V]\ \Delta]\!]\ c\ k\ \rho\ d\ \parallel\ V \in \mathrm{dom}\,\rho, Pe \neq \square =$$
$$\texttt{REQ}\ (d-i)\ (d-\rho(V));\ \mathbf{PeL}[\![\Gamma\ [d+2 : Pe]\ \Delta]\!]\ c\ k\ \rho\ (d+2)$$

$$\mathbf{PeL}[\![\Gamma\ [i : Pe\,Vs]\ \Delta]\!]\ c\ k\ \rho\ d\ \parallel\ Vs \notin \mathrm{dom}\,\rho, Pe \neq \square =$$
$$\texttt{RS}\ (d-i);\ \mathbf{PeL}[\![\Gamma\ [d+2 : Pe]\ \Delta]\!]\ c\ k\ (\rho + \{Vs = d+1\})\ (d+2)$$

$$\mathbf{PeL}[\![\Gamma\ [i : Pe\,Vt]\ \Delta]\!]\ c\ k\ \rho\ d\ \parallel\ Vt \notin \mathrm{dom}\,\rho, Pe \neq \square =$$
$$\texttt{RT}\ (d-i);\ \mathbf{PeL}[\![\Gamma\ [d+2 : Pe]\ \Delta]\!]\ c\ k\ (\rho + \{Vt = d+1\})\ (d+2)$$

4.3.13.1 "Открытые" ve-переменные слева

Следующие два правила компиляции имеет смысл применять только в тех случаях, когда ни одно из предыдущих правил не применимо. Требуется исправление стека альтернатив k, ибо команда `LE` неявно содержит в себе команду `ALT`.

$$\mathbf{PeL}[\![[i : Vv\,Pe]\ \Delta]\!]\ c\ (h : k)\ \rho\ d\ \parallel\ Vv \notin \mathrm{dom}\,\rho,\ Pe \neq \square =$$
$$\texttt{LVI}\ (d-i);\ \texttt{LE};$$
$$\mathbf{PeL}[\![[d+2 : Pe]\ \Delta]\!]\ c\ ((h+1) : k)\ (\rho + \{Vv = d+1\})\ (d+2)$$

$$\mathbf{PeL}[\![[i : VePe]\ \Delta]\!]\ c\ (h : k)\ \rho\ d\ \parallel\ Ve \notin \mathrm{dom}\,\rho,\ Pe \neq \square =$$
$$\texttt{LEI}\ (d-i);\ \texttt{LE};$$
$$\mathbf{PeL}[\![[d+2 : Pe]\ \Delta]\!]\ c\ ((h+1) : k)\ (\rho + \{Ve = d+1\})\ (d+2)$$

4.3.14 Компиляция образцовых распутий

Результат компиляции образцовых распутий может быть описан с помощью функции $Palt$, обращение к которой имеет следующий вид:

$$\mathbf{Palt}[\![Palt]\!]\ [\![Fin]\!]\ [\![Fout]\!]\ r\ k\ \rho\ d$$

где Fin, $Fout$, r, k, ρ и d имеют тот же смысл, что и в случае функции **Snt**.

$$\begin{array}{l}
\mathbf{Palt}[\![\backslash\{Snt_1\,;\ \ldots\ Snt_n\,;\ Snt_0\,;\}]\!]\ [\![Fin]\!]\ [\![Fout]\!]\ r\ (h:k)\ \rho\ d = \\
\quad \texttt{ALT}\ L1; \\
\quad\quad \mathbf{Snt}[\![Snt_1]\!]\ [\![Fin]\!]\ [\![Fout]\!]\ r\ ((h+1):k)\ \rho\ d;\ \texttt{JUMP}\ L0; \\
\quad \texttt{LABEL}\ L1; \\
\quad\quad \mathbf{Palt}[\![\backslash\{\ \ldots\ Snt_n\,;\ Snt_0\,;\}]\!]\ [\![Fin]\!]\ [\![Fout]\!]\ r\ (h:k)\ \rho\ d; \\
\texttt{LABEL}\ L0
\end{array}$$

$$\begin{array}{l}
\mathbf{Palt}[\![\backslash\{Snt_0\,;\}]\!]\ [\![Fin]\!]\ [\![Fout]\!]\ r\ (h:k)\ \rho\ d = \\
\quad \mathbf{Snt}[\![Snt_0]\!]\ [\![Fin]\!]\ [\![Fout]\!]\ r\ (h:k)\ \rho\ d
\end{array}$$

$$\mathbf{Palt}[\![\backslash\{\}]\!]\ [\![Fin]\!]\ [\![Fout]\!]\ r\ (h:k)\ \rho\ d = \texttt{FAIL}$$

$$\begin{array}{l}
\mathbf{Palt}[\![\{Snt_1\,;\ \ldots\ Snt_n\,;\ Snt_0\,;\}]\!]\ [\![Fin]\!]\ [\![Fout]\!]\ r\ (h:k)\ \rho\ d = \\
\quad \texttt{ALT}\ L1; \\
\quad\quad \mathbf{Snt}[\![Snt_1]\!]\ [\![Fin]\!]\ [\![Fout]\!]\ r\ ((h+1):k)\ \rho\ d;\ \texttt{JUMP}\ L0; \\
\quad \texttt{LABEL}\ L1; \\
\quad\quad \mathbf{Palt}[\![\{\ \ldots\ Snt_n\,;\ Snt_0\,;\}]\!]\ [\![Fin]\!]\ [\![Fout]\!]\ r\ (h:k)\ \rho\ d; \\
\quad \texttt{LABEL}\ L0
\end{array}$$

$$\begin{array}{l}
\mathbf{Palt}[\![\{Snt_0\,;\}]\!]\ [\![Fin]\!]\ [\![Fout]\!]\ r\ (h:k)\ \rho\ d = \\
\quad \texttt{ALT}\ Lerror; \\
\quad\quad \mathbf{Snt}[\![Snt_0]\!]\ [\![Fin]\!]\ [\![Fout]\!]\ r\ ((h+1):k)\ \rho\ d
\end{array}$$

$$\mathbf{Palt}[\![\{\}]\!]\ [\![Fin]\!]\ [\![Fout]\!]\ r\ (h:k)\ \rho\ d = \texttt{JUMP}\ Lerror$$

4.3.15 Компиляция определения функции

Результат компиляции определений функций может быть описан с помощью функции **FD**, обращение к которой имеет следующий вид:

$$\mathbf{FD}[\![FDec;FD]\!]$$

где $FDec$ - объявление функции, а FD - ее определение.

FD$[\![$`$func?` $\mathcal{F}$ Fin `=` $Fout$`;` $\mathcal{F}$ $Palt$`;`$]\!] =$
let val $[V_1, \ldots, V_a] = \mathcal{A}[\![Fin]\!]$ **in**
`FUNSTART` $\mathcal{F}$`;`
`ALT` $Lfail$`;`
Palt$[\![Palt]\!]$ $[\![Fin]\!]$ $[\![Fout]\!]$ 0 [0] {} $(a-1)$`;`
`CUT` 1`;` `RETURN`;
`LABEL` $Lfail$`;`
`RETURNFAIL`;
`LABEL` $Lerror$`;`
`PUSHQ` $\mathcal{F}$ `"Unexpected fail";` `ERROR`
end

FD$[\![$`$func` $\mathcal{F}$ Fin `=` $Fout$`;` $\mathcal{F}$ $Palt$`;`$]\!] =$
let val $[V_1, \ldots, V_a] = \mathcal{A}[\![Fin]\!]$ **in**
`FUNSTART` $\mathcal{F}$`;`
`ALT` $Lerror$`;`
Palt$[\![Palt]\!]$ $[\![Fin]\!]$ $[\![Fout]\!]$ 0 [0] {} $(a-1)$`;`
`CUT` 1`;` `RETURN`;
`LABEL` $Lerror$`;`
`PUSHQ` $\mathcal{F}$ `"Unexpected fail";` `ERROR`
end

Литература

[АбР 1988] С.М.Абрамов, С.А.Романенко. Представление объектных выражений массивами при реализации языка Рефал. — М.:ИПМ им.М.В.Келдыша АН СССР, 1988, препринт N 186. — 27 с.

[АОППС 2004] С.М.Абрамов, А.Ю.Орлов, Л.В.Парменова, С.М.Пономарева, А.Ф.Слепухин. Новый подход к реализации системы программирования Рефал Плюс // Труды международной конференции "Программные системы: теория и приложения", ИПС РАН, г.Переславль-Залесский, май 2004 / Под редакцией С.М.Абрамова. В двух томах. — М.: Физматлит, 2004. — Т. 1, 530 с., ил. — ISBN 5-94052-067-9

[АО 2004] С.М.Абрамов, А.Ю. Орлов. Компиляция в императивные языки синтаксического отождествления языка Рефал // Труды международной конференции "Программные системы: теория и приложения", ИПС РАН, г.Переславль-Залесский, май 2004 / Под редакцией С.М.Абрамова. В двух томах. — М.: Физматлит, 2004. — Т. 1, 530 с., ил. — ISBN 5-94052-067-9

[Апт 1983] K.R.Apt. Formal Justification of a Proof System for Communicating Sequential Processes — Journal Assoc. Comput. Machin., 30(1), pp.197-216, 1983.

[АХУ 1979] А.Ахо, Дж.Хопкрофт, Дж.Ульман. Построение и анализ вычислительных алгоритмов: Пер. с англ. — М.:Мир, 1979. — 536 с.

[БзР 1977] Базисный Рефал и его реализация на вычислительных машинах. — М.:ЦНИПИАСС, 1977. — 258 с.

[БьД 1982] D.Bjorner, C.B.Jones. Formal Specification and Software Development. — Prentice-Hall International, London, 1982.

[Вир 1977] Н.Вирт. Систематическое программирование. Введение: Пер. с англ. — М.:Мир, 1977. — 184 с.

[Вир 1985] Н.Вирт. Алгоритмы + структуры данных = программы: Пер. с англ. — М.:Мир, 1985. — 406 с.

[Гер 500] Антология мировой философии в четырех томах. Том 1, часть 1. — М.:Мысль, 1969. — с.275.

[Кис 1987] В.Л.Кистлеров. Принципы построения языка алгебраических вычислений FLAC. — Препринт, М.:Институт проблем управления, 1987. — 39 с.

[КлР 1986] Ан.В.Климов, С.А.Романенко. Система программирования Рефал-2 для ЕС ЭВМ. Описание библиотеки функций. — М.:ИПМ им.М.В.Келдыша АН СССР, 1986, препринт N 200. — 38 с.

[КлР 1987] Ан.В.Климов, С.А.Романенко. Система программирования Рефал-2 для ЕС ЭВМ. Описание входного языка. — М.:ИПМ им.М.В.Келдыша АН СССР, 1987. — 52 с.

[КРТ 1972] Ан.В.Климов, С.А.Романенко, В.Ф.Турчин. Компилятор с языка Рефал. — М.:ИПМ АН СССР, 1972. — 74 с.

[Плоткин 1983] G.D.Plotkin. An Operational Semantics for CSP, in: D.Bjorner (ed.), Formal Description of Programming Concepts II, — North-Holland, Amsterdam, pp.199-223.

[Ром 1987а] С.А.Романенко. Реализация Рефала-2. — М.:ИПМ им.М.В.Келдыша АН СССР, 1977. — 191 с.

[Ром 1987б] С.А.Романенко. Рефал-4 - расширение Рефала-2, обеспечивающее выразимость результатов прогонки. — М.:ИПМ им.М.В.Келдыша АН СССР, 1987, препринт N 147. — 27 с.

[Ром 1987в] С.А.Романенко. Прогонка для программ на Рефале-4. — М.:ИПМ им.М.В.Келдыша АН СССР, 1987, препринт N 211. — 19 с.

[Ром 1987г] С.А.Романенко. Генератор компиляторов, порожденный самоприменением специализатора, может иметь ясную и естественную структуру. — М.:ИПМ им.М.В.Келдыша АН СССР, 1987, препринт N 26. — 35 с.

[Ром 1988а] С.А.Романенко. Мета-мета-вычисления и специализация программ. — В сб.: Тезисы докладов Всесоюзной школы-семинара "Семиотические аспекты формализации интеллектуальной деятельности" в г.Боржоми, 22-30 апреля 1988 г. — М.:ВИНИТИ, 1988, с.65-68.

[Ром 1988б] S.A.Romanenko. A Compiler Generator Produced by a Self-Applicable Specializer Can Have a Surprisingly Natural and Understandable Structure. — In D.Bjorner, A.P.Ershov and N.D.Jones, editors, Partial Evaluation and Mixed Computation, — North-Holland, 1988, pages 445-463.

[Тур 1966] В.Ф.Турчин. Метаязык для формального описания алгоритмических языков. — В сб.: Цифровая вычислительная техника и программирование. — М.:Сов. Радио, 1966, с.116-124.

[Тур 1971] В.Ф.Турчин. Программирование на языке Рефал. — М.:ИПМ АН СССР, 1971, препринты N 41, N 43, N 44, N 48, N 49.

[Тур 1986] V.F.Turchin. The concept of a supercompiler. — ACM Transactions on Programming Languages and Systems, Vol.8, No.3, July 1986, pp.292-325.

[Тур 1989] V.F.Turchin. Refal-5, Programming Guide and Reference Manual. — New England Publishing Co., Holyoke, 1989.

[Уор 1980] D.H.D.Warren. Logic Programming and Compiler Writing. — Software – Practice and Experience, Vol.10, 97-125 (1980).

[Хен 1983] П.Хендерсон. Функциональное программирование. Применение и реализация: Пер. с англ. — М.:Мир, 1983. — 349 с.

[Шмд 1986] D.A.Schmidt. Denotational Semantics. — Allyn and Bacon, Boston, 1986.

Приложение А

Алфавитный указатель функций

```
$func  Add         s.Int1 s.Int2 = s.Int;              Arithm
$func? Apply       s.Name e.Exp = e.Exp;               Apply
$func  Arg         s.Int = e.Arg;                      Dos
$func  Bind        s.Tab (e.Key) (e.Val) = ;           Table
$func  BitAnd      s.Int1 s.Int2 = s.Int;              Bit
$func  BitClear    s.Int s.Pos = s.Int;                Bit
$func  BitLeft     s.Int s.Shift = s.Int;              Bit
$func  BitLength   s.Int = s.Len;                      Bit
$func  BitNot      s.Int = s.Int;                      Bit
$func  BitOr       s.Int1 s.Int2 = s.Int;              Bit
$func  BitRight    s.Int s.Shift = s.Int;              Bit
$func  BitSet      s.Int s.Pos = s.Int;                Bit
$func? BitTest     s.Int s.Pos = ;                     Bit
$func  BitXor      s.Int1 s.Int2 = s.Int;              Bit
$func  Box         e.Exp       = s.Box;                Box
$func  BytesToChars e.Char = e.Int;                    Convert
$func  Channel     = s.Channel;                        StdIo
$func  CharsToBytes e.Int  = e.Char;                   Convert
$func  CloseChannel  s.Channel = ;                     StdIo
$func  Compare     (e.Exp1)(e.Exp2)  = s.Res;          Compare
$func  Div         s.Int1 s.Int2 = s.Quo;              Arithm
$func  DivRem      s.Int1 s.Int2 = s.Quo s.Rem;        Arithm
$func? Domain      s.Tab = e.KeyList ;                 Table
$func? Eq          (e.Exp1)(e.Exp2) = ;                Compare
$func  Exit        s.ReturnCode = ;                    Dos
$func  Gcd         s.Int1 s.Int2 = s.Gcd;              Arithm
$func? Ge          (e.Exp1)(e.Exp2) = ;                Compare
```

```
$func  Get         s.Box = e.Exp;                          Box
$func  GetEnv      e.VarName = e.Value;                    Dos
$func? Gt          (e.Exp1)(e.Exp2) = ;                    Compare
$func? IsBox        e.Exp = ;                              Class
$func? IsChannel    e.Exp = ;                              Class
$func? IsChar       e.Exp = ;                              Class
$func? IsDigit      e.Exp = ;                              Class
$func? IsEof        s.Channel = ;                          StdIo
$func? IsFunc       e.Exp = ;                              Class
$func? IsInTable    s.Tab e.Key = ;                        Table
$func? IsInt        e.Exp = ;                              Class
$func? IsLetter     e.Exp = ;                              Class
$func? IsString     e.Exp = ;                              Class
$func? IsTable      e.Exp = ;                              Class
$func? IsVector     e.Exp = ;                              Class
$func? IsWord       e.Exp = ;                              Class
$func? L            s.Left  e.Exp = t.SubTerm;             Access
$func? Le           (e.Exp1)(e.Exp2) = ;                   Compare
$func? Left         s.Left  s.Len e.Exp = e.SubExp;        Access
$func  Length       e.Exp = s.ExpLen;                      Access
$func? Lt           (e.Exp1)(e.Exp2) = ;                   Compare
$func? Lookup       s.Tab e.Key = e.Val;                   Table
$func? Middle       s.Left  s.Right e.Exp = e.SubExp;      Access
$func  Mult         s.Int1 s.Int2 = s.Int;                 Arithm
$func? Ne           (e.Exp1)(e.Exp2) = ;                   Compare
$func? OpenFile    s.Channel e.FileName s.Mode = ;         StdIo
$func  Print       e.Expr = ;                              StdIo
$func  PrintCh     s.Channel e.Expr = ;                    StdIo
$func  PrintLn     e.Expr = ;                              StdIo
$func  PrintLnCh   s.Channel e.Expr = ;                    StdIo
$func? R           s.Right e.Exp = t.SubTerm;              Access
$func? Read         = t.Term;                              StdIo
$func? ReadCh       s.Channel = t.Term;                    StdIo
$func? ReadChar     = s.Char;                              StdIo
$func? ReadCharCh   s.Channel = s.Char;                    StdIo
$func? ReadLine     = e.Char;                              StdIo
$func? ReadLineCh   s.Channel = e.Char;                    StdIo
$func  Rem         s.Int1 s.Int2 = s.Rem;                  Arithm
$func  ReplaceTable  s.TargetTab s.SourceTab = ;           Table
$func? Right       s.Right s.Len e.Exp = e.SubExp;         Access
$func  Store       s.Box e.Exp = ;                         Box
$func  String         e.Source = s.String;                 String
$func  StringFill     s.String s.Fill = ;                  String
$func  StringInit     s.String s.Len s.Fill = ;            String
```

```
$func  StringLength  s.String = s.Len;                    String
$func  StringRef     s.String s.Index = s.Char;           String
$func  StringReplace s.String e.Source = ;               String
$func  StringSet     s.String s.Index s.Char = ;          String
$func  Sub        s.Int1 s.Int2 = s.Int;                  Arithm
$func  Substring  s.String s.Index s.Len = s.NewString;  String
$func  SubstringFill s.String s.Index s.Len s.Fill =;    String
$func  Subvector     s.Vector s.Index s.Len = s.Vector;   Vector
$func  SubvectorFill s.Vector s.Index s.Len e.Fill =;    Vector
$func  Table     = s.Tab;                                Table
$func  TableCopy s.Tab = s.TabCopy ;                     Table
$func  Time      = e.String;                             Dos
$func  ToChars   e.Exp  = e.Char;                        Convert
$func? ToInt     e.Char = s.Int;                         Convert
$func  ToLower   e.Char = e.Char;                        Convert
$func  ToUpper   e.Char = e.Char;                        Convert
$func  ToWord    e.Char = s.Word;                        Convert
$func  Unbind    s.Tab e.Key = ;                         Table
$func  Vector        e.Source = s.Vector;                Vector
$func  VectorFill    s.Vector e.Fill = ;                 Vector
$func  VectorInit    s.Vector s.Len e.Fill = ;           Vector
$func  VectorLength  s.Vector = s.Len;                   Vector
$func  VectorRef     s.Vector s.Index = e.Exp;           Vector
$func  VectorReplace s.Vector e.Source = ;               Vector
$func  VectorSet     s.Vector s.Index e.Exp = ;          Vector
$func  VectorToExp   s.Vector = e.Exp;                   Vector
$func  Write       e.Expr = ;                            StdIo
$func  WriteCh     s.Channel e.Expr = ;                  StdIo
$func  WriteLn     e.Expr = ;                            StdIo
$func  WriteLnCh   s.Channel e.Expr = ;                  StdIo
```