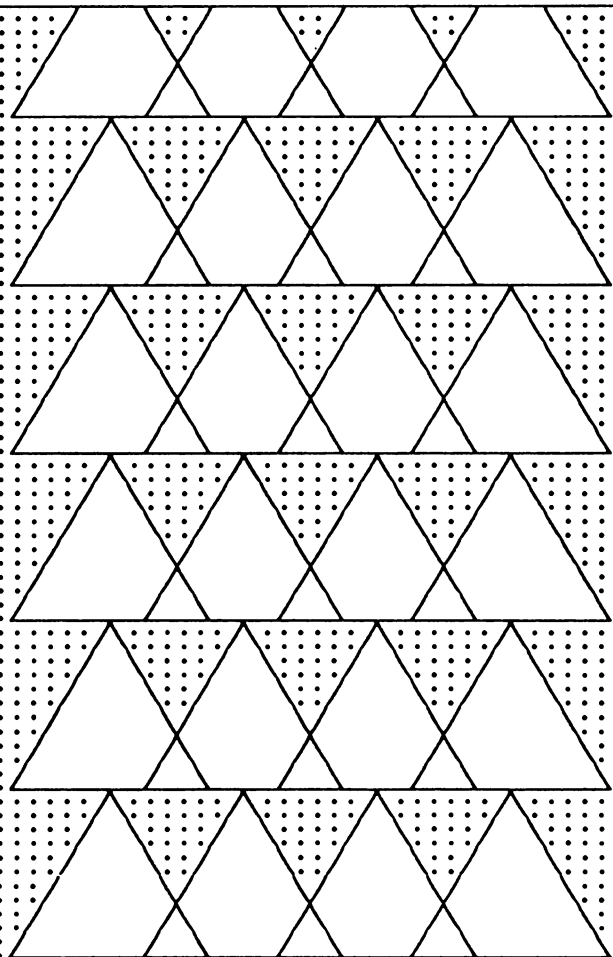


Р.Ф. ГУРИН, С.А. РОМАНЕНКО

ЯЗЫК ПРОГРАММИРОВАНИЯ

РЕФАЛ ПЛЮС



Рутен Ф.Гурин, Сергей А.Романенко

**ЯЗЫК ПРОГРАММИРОВАНИЯ
РЕФАЛ ПЛЮС**

"ИНТЕРТЕХ"

Москва 1991

УДК 681.3

Гурин Р.Ф., Романенко С.А.

Язык программирования Рефал Плюс. — М.: ИНТЕРТЕХ, 1991. — 183 с.

Описывается язык программирования Рефал Плюс, предназначенный для обработки символьной информации. Рассматриваются методы программирования на Рефале Плюс, которые иллюстрируются многочисленными примерами программ.

Для программистов, работающих в областях, связанных с обработкой символьной информации, таких как компьютерная алгебра, конструирование компиляторов и интерпретаторов, искусственный интеллект.

Тираж 1000

(с) Р.Ф.Гурин, С.А.Романенко, 1991.

ББК 32.973
Г95

СОДЕРЖАНИЕ

Введение	6
Глава 1. Программирование на рефале плюс	11
1. Первый пример	11
2. Структуры данных	12
2.1. Объектные выражения (12). 2.2. Представление данных в виде объектных выражений (14). 2.3. Объекты и значения (15). 2.4. Сборка мусора (17).	
3. Вычисление и анализ объектных выражений	19
3.1. Результатные выражения (19). 3.2. Переменные (21). 3.3. Форматы функций (22). 3.4. Образцы (23). 3.5. Тропы, хвосты и источники (26). 3.6. Огражденные тропы (27). 3.7. Результатные выражения в качестве источников (27). 3.8. Правые части (27).	
4. Функции определяемые в программе	29
4.1. Определения функций (29). 4.2. Локальные переменные (31). 4.3. Рекурсия (33).	
5. Неуспехи и аварии	36
5.1. Неуспех при вычислении результатных выражений и троп (36). 5.2. Перестройки (37). 5.3. Перехват неуспехов (40). 5.4. Управление перехватом неуспехов (42). 5.5. Смысл правых частей (44). 5.6. Откатные и безоткатные функции (45).	
6. Логические условия	47
6.1. Проверки и предикаты (47). 6.2. Ветвления (48). 6.3. Логические связки (49). 6.4. Пример: программа дифференцирования (51). 6.5. Пример: сравнение множеств (54).	
7. Использование селекторов прямого доступа	57
8. Функции, выдающие несколько результатов	58
8.1. Сквозной просмотр выражений (58). 8.2. Быстрая сортировка (60).	
9. Итеративные циклы	61
10. Перебор с возвратом	64
10.1. Задача о ферзях (64). 10.2. Задача о цепочках (68).	
11. Пример: компилятор для простого императивного языка	71
11.1. Входной язык компилятора (71). 11.2. Выходной язык (74). 11.3. Общая структура компилятора (76).	

- 11.4. Модули компилятора и их интерфейсы (78).
 11.5. Головной модуль компилятора (80). 11.6. Сканер (81).
 11.7. Синтаксический анализатор (85).
 11.8. Генератор кода (94). 11.9. Модуль работы со словарем (101).

Глава 2. Синтаксис и семантика рефала плюс	104
1. Нотация для записи синтаксиса	104
2. Естественный метод описания семантики	106
3. Лексическая структура программы	108
3.1. Комментарии (109). 3.2. Лексемы (109). 3.3. Ключевые слова (109). 3.4. Символы-литеры (110). 3.5. Символы-слова (111). 3.6. Символы-числа (113). 3.7. Переменные (113). 3.8. Нормализация потока лексем (114).	
4. Объекты и значения	115
5. Объектные выражения	116
5.1. Синтаксис объектных выражений (116). 5.2. Статические и динамические символы (117). 5.3. Символические имена выражений (118). 5.4. Устранение символических имен выражений (118).	
6. Значения переменных и среды	118
7. Результатные выражения	120
7.1. Синтаксис (120). 7.2. Вычисление результатных выражений (120). 7.3. Примеры (122).	
8. Образцы	123
8.1. Синтаксис (123). 8.2. Сопоставление с образцом (124). 8.3. Примеры (125).	
9. Жесткие выражения	126
9.1. Синтаксис (126). 9.2. Сопоставление с жестким выражением (126). 9.3. Примеры (127).	
10. Тропы	128
10.1. Синтаксис (128). 10.2. Вычисление троп (129). 10.3. Условия (130). 10.4. Присваивания (131). 10.5. Поиски (132). 10.6. Перестройки (133). 10.7. Огражденные тропы (134). 10.8. Отрицания условий (134). 10.9. Заборы (135). 10.10. Отсечения (135). 10.11. Тупики (136). 10.12. Правые части (136). 10.13. Аварии (137). 10.14. Перехваты аварий (137). 10.15. Распутья (138). 10.16. Выборы (139). 10.17. Результатные выражения как источники (140).	
11. Определения функций	140
12. Объявления	142
12.1. Объявления констант (142). 12.2. Объявления ящиков, векторов, строк, таблиц и каналов (142).	

12.3.Объявления функций (143). 12.4.Директивы отладки (144).	
13.Контекстные ограничения	145
13.1.Устранение избыточных конструкций (145).	
13.2.Ограничения, налагаемые объявлениями функций (146).	
13.3.Ограничения на использование ссылок на функции (150).	
13.4.Ограничения на использование переменных (150).	
13.5.Ограничения на использование отсечений (152).	
14.Модули	154
15.Исполнение программы	155
Глава 3. Библиотека функций	156
1.Использование библиотечных функций	156
2.ACCESS: прямой доступ к выражениям	157
3.APPLY: вызов функций переданных через параметры	159
4.ARITHM: целочисленная арифметика	159
5.BOX: работа с ящиками	161
6.CLASS: предикаты для распознавания классов символов	162
7.COMPARE: сравнение выражений	163
8.CONVERT: преобразования между различными типами данных	165
9.DOS: связь с операционной системой	167
10.STDIO: стандартный ввод-вывод	168
11.STRING: работа со строками	170
12.TABLE: работа с таблицами	173
13.VECTOR: работа с векторами	174
Литература	177
Алфавитный указатель функций	181

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 66], [Тур 71]. Впоследствии, после появления достаточно эффективных реализаций [КРТ 72], [Бэр 77], [Ром 87а] 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 77], Рефала-2 [КлР 86], [КлР 87], [Ром 87а], Рефала-4 [Ром 87б], [Ром 87в], Рефала-5 [Тур 89], и языков FLAC [Кис 87] и RL [Ром 87г], [Ром 88а], [Ром 88б].

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 порождаться 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* (co-arity).

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 88], позволяющем свести копирование выражения к копированию пары указателей на его концы.

Дешевизна операции копирования позволяет применять функциональный стиль программирования 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.

В настоящее время имеется реализация Рефала Плюс для персональных компьютеров IBM PC XT/AT (или совместимых с ними). За справками обращаться: 123103, Москва, набережная Новикова—Прибоя, д. 9, кор.2, СП "ИНТЕРТЕХ", тел. 262—33—22, 262—08—33, факс 273—19—37, телекс 412176 INTEC.

Глава I

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 2 "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 3 "Библиотека функций" и к разделу "Алфавитный указатель функций".

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.

2. Структуры данных

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 -999999999999999

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.

имеют место соотношения

« $p+q$ »	=	("плюс" « p » « q »)
« $p-q$ »	=	("минус" « p » « q »)
« $p \times q$ »	=	("умн" « p » « q »)
« p/q »	=	("дел" « p » « q »)
« p^q »	=	("степ" « p » « q »)

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))

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 *име-*

нем 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:

$$R \longrightarrow \boxed{C}$$

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.

2.4. Сборка мусора

В языке Рефал Плюс имеется возможность породить объек-

ты 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, имеющих в этот момент.

На рисунке 1.1 схематически изображены значения переменных, а также объекты и их содержимое. Звездочки (*) изображают некоторые элементы выражений, которые не являются символами-ссылками. Для удобства изложения, все объекты на рисунке пронумерованы. Символы-ссылки обозначены темными кружками (●).

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.

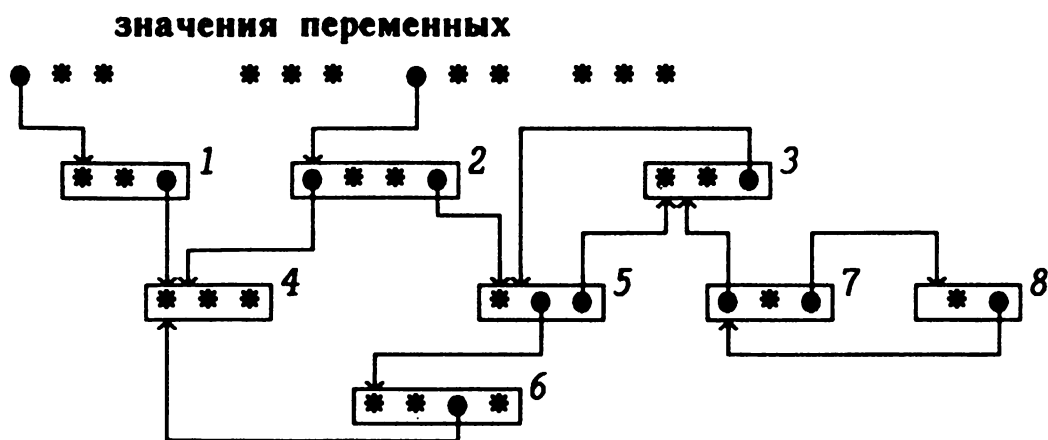


Рис.1.1. Объекты и ссылки.

3. Вычисление и анализ объектных выражений

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:

$$\langle "+" \langle "*" sX sY \rangle 3 \rangle$$

Угловые скобки обозначают вызов функции. Причем первый символ-слово после левой угловой скобки обозначает имя функции, а все остальное соответствует аргументом 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:

$$\langle "*" sX \langle "+" sA sB \rangle \rangle$$

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

$$\langle "+" \langle "*" sX sA \rangle sB \rangle$$

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

$$\{V_1 = 0e_1, \dots, V_n = 0e_n\}$$

будет изображать среду, в которой переменные V_1, \dots, V_n имеют значения $0e_1, \dots, 0e_n$ 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 Oe' и Oe'' соответственно, то результатом вычисления $Re' Re''$ является объектное выражение $Oe' Oe''$.

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 Oe , то результатом вычисления (Re) является (Oe) .

Например, результатом вычисления результатного выражения

$$sX '+' sY (eZ)$$

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

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

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 произ-

вольные объектные выражения.

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

3.3. 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 "+" 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 "+" имеет вид:

$$\$func "+" sX sY = sZ;$$

В общем случае объявление функции *Fname* имеет вид

$$\$func Fname F_{in} = F_{out};$$

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 результирующем 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

```
<"+" 2 <"+" 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 "+". А именно, заменяем <"+" sX sY> на выходной формат функции "+", в результате чего получается <"+" 2 s>. Отсюда видно, что аргумент наружного вызова соответствует входному формату функции "+". С другой стороны, если мы напишем в программе результатное выражение

```
<"+" 2 <"+" sX sY> 3>
```

оно будет расцениваться компилятором как ошибочное, поскольку аргумент наружного вызова функции "+" 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.

3.4. 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 выраже-

ний, 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 Oe и образец P , то всегда можно сопоставить Oe с P и установить, имеет ли Oe структуру, предписанную образцом P . Если да, то мы говорим, что Oe успешно сопоставляется с P , в противном случае мы говорим, что результатом сопоставления является *неуспех*.

В результате успешного сопоставления Oe с P переменные, входящие в P , связываются с соответствующими частями выражения Oe . Таким образом, результатом сопоставления Oe 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:

$$\{e1 = , \quad sX = A, \quad e2 = B C\}$$

$$\{e1 = A, \quad sX = B, \quad e2 = C\}$$

$$\{e1 = A B, \quad sX = C, \quad e2 = \}$$

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' и ρ'' — два варианта сопоставления Oe с P . Рассмотрим все вхождения переменных в P . Если ρ' и

ρ'' не совпадают, они приписывают некоторым переменным различные значения. Найдем в P самое первое слева вхождение переменной которому ρ' и ρ'' приписывают разные значения и сравним длину этих значений. Если значение, приписываемое средой ρ' , короче, чем значение, приписываемое средой ρ'' , то считается, что ρ' "предшествует" ρ'' , т.е. имеет приоритет перед ρ'' , в противном случае считается, что ρ'' "предшествует" ρ' .

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

$$\begin{aligned} \{e1 = , eX = , sA = A1, eY = A2 A3, e2 = (B1 B2)\} \\ \{e1 = , eX = A1, sA = A2, eY = A3, e2 = (B1 B2)\} \\ \{e1 = , eX = A1 A2, sA = A3, eY = , e2 = (B1 B2)\} \\ \{e1 = (A1 A2 A3), eX = , sA = B1, eY = B2, e2 = \} \\ \{e1 = (A1 A2 A3), eX = B1, sA = B2, eY = , e2 = \} \end{aligned}$$

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:

$$\begin{aligned} \{e1 = (A1 A2 A3), eX = B1, sA = B2, eY = , e2 = \} \\ \{e1 = (A1 A2 A3), eX = , sA = B1, eY = B2, e2 = \} \\ \{e1 = , eX = A1 A2, sA = A3, eY = , e2 = (B1 B2)\} \\ \{e1 = , eX = A1, sA = A2, eY = A3, e2 = (B1 B2)\} \\ \{e1 = , eX = , sA = A1, eY = A2 A3, e2 = (B1 B2)\} \end{aligned}$$

3.5. Тропы, хвосты и источники

Из результатных выражений и образцов строятся более крупные синтаксические единицы — *тропы*. Если результатные выражения можно считать аналогом арифметических выражений, то тропы представляют собой аналог операторов, таких как, например, оператор присваивания, оператор цикла и др.

Вычисление тропы выполняется в некоторой среде и приводит к получению некоторого значения.

В Рефале Плюс весьма экономно используются ключевые слова и разделители, в связи с чем могут возникать синтаксические проблемы. Так, например, тропы могут начинаться с результатных выражений. Пусть Q — одна из таких троп, а Re — некоторое результатное выражение. Попробуем записать Q сразу вслед за Re . В результате получается двусмысленная конструкция $Re\ Q$ в которой невозможно понять, где кончается Re и где начинается Q .

Чтобы преодолеть эту трудность, мы можем потребовать, чтобы во всех сомнительных случаях тропа отделялась от предшествующего результатного выражения каким-то разделителем (например — запятой) следующим образом: $Re\ ,\ Q$. Это, конечно, устраняет неоднозначность, но может приводить к загромождению программы, поскольку, в большинстве случаев, Q все же может быть однозначно отделено от предшествующей конструкции и без дополнительных разделителей.

Таким образом, среди всех троп нужно выделить подкласс "хороших" троп, которые легко отделяются от предшествующих конструкций. Такие тропы мы называем *хвостами*.

Другим важным подклассом троп являются *источники*. Одним из основных синтаксических свойств источника является то, что он заведомо не может содержать запятую на нулевом уровне фигурных скобок.

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

В следующих разделах мы пока ограничимся рассмотрением

только нескольких простейших примеров троп, хвостов и источников.

3.6. Огражденные тропы

Если Q — некоторая тропа, то мы всегда можем превратить ее в хвост, приписав перед ней запятую. В результате получается *огражденная тропа*

$$, Q$$

которая с синтаксической точки зрения является хвостом, а во всех прочих отношениях эквивалентна тропе Q .

3.7. Результатные выражения в качестве источников

Всякое результатное выражение Re является источником, а значит — и тропой. Приписав к нему запятую, получаем хвост

$$, Re$$

Вычисление источников вида Re сводится к вычислению результатного выражения Re . Если при этом получается объектное выражение Oe , то Oe считается результатом источника.

3.8. Правые части

Всякая конструкция вида

$$= Q$$

где Q — некоторая тропа, называется *правой частью*. С синтаксической точки зрения правые части являются хвостами, а значит — и тропами.

Вычисление правой части $= Q$ сводится к вычислению тропы Q . Если при этом получается объектное выражение Oe , то Oe считается результатом всей правой части.

На первый взгляд кажется, что различие между тропами вида Re и тропами вида $= Re$ — чисто синтаксическое, но

это не так. Тонкая разница между ними проявляется в том случае, если вычисление Re приводит к *неуспеху*, и к этому вопросу мы вернемся позднее.

4. Функции определяемые в программе

4.1. Определения функций

Рефал-программа состоит из *определений функций*, каждое из которых может иметь один из двух следующих видов:

$$\begin{aligned} \text{Fname} \setminus \{ \text{Snt}_1; \text{Snt}_2; \dots \text{Snt}_n; \}; \\ \text{Fname} \{ \text{Snt}_1; \text{Snt}_2; \dots \text{Snt}_n; \}; \end{aligned}$$

где Fname — имя функции, а $\text{Snt}_1, \text{Snt}_2, \dots, \text{Snt}_n$ — *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 Fname вида

$$\langle \text{Fname Re} \rangle$$

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

Например, определим функцию Sumsq , вычисляющую сумму квадратов двух чисел. В общепринятых обозначениях ее определение записывается в виде

$$\text{Sumsq}(X, Y) = X * X + Y * Y$$

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

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

```
Sumsq
```

```
{
  sX sY = <"+" <"*" sX sX> <"*" 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

```
$func Fname Fin = Fout;
```

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

При записи определений функций разрешается использовать следующие сокращения.

Если в предложении Snt_j хвост R_j состоит из одной запятой, его разрешается опускать, в результате чего предложение Snt_j принимает вид P_j .

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

```
Fname \{ Snt; };
```

его разрешается записывать в сокращенном виде следующим образом:

```
Fname Snt;
```

Например, определение функции Sumsq может быть записано следующим образом:

```
Sumsq sX sY = <"+" <"*" sX sX> <"*" sY sY>>;
```

4.2. Локальные переменные

Определим функцию Sq-Sub1, вычитающую из аргумента единицу и возводящую полученное число в квадрат:

$$\text{Sq-Sub1}(X) = (X-1)*(X-1)$$

Простейшее определение на Рефале имеет вид:

```
$func Sq-Sub1 sX = sZ;
```

```
Sq-Sub1 sX = <"*" <"-" sX 1> <"-" sX 1>>;
```

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

```
$func Sq-Sub1 sX = sZ;
```

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

```
Sq-Sub1 sX = <Sq <"-" sX 1>>;
```

```
Sq      sY = <"*" 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, это можно сделать с помощью тропы вида

$S :: \text{He } R$

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 *Oe*, то переменные из *He* связываются с соответствующими частями *Oe*, после чего вычисляется хвост *R*, и полученный результат считается результатом всей конструкции.

Теперь мы можем записать определение функции *Sq-Sub1* следующим образом:

```
$func Sq-Sub1 sX = sZ;
```

```
Sq-Sub1 sX =  
  <"-" sX 1> :: sY,  
  <"*" sY sY>;
```

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, <"+" 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 А, вторая — литеру В, а третья — литеру С.

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

4.3. 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

$$Ot_1 \ Ot_2 \ \dots \ Ot_n$$

где Ot_1, Ot_2, \dots, Ot_n — некоторые объектные термы, то на выходе должно получиться объектное выражение

$$Ot_n \ \dots \ Ot_2 \ Ot_1$$

Если бы длина входного выражения была ограничена, например, если бы мы знали, что $n \leq 3$, 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

$$0t_1 \ 0t_2 \ \dots \ 0t_n$$

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

$$0t_2 \ \dots \ 0t_n$$

и перевернуть это более короткое выражение. Получится

$$0t_n \ \dots \ 0t_2$$

Теперь ясно, что если к этому выражению приписать справа $0t_1$, мы как раз и получим желаемый результат

$$0t_n \ \dots \ 0t_2 \ 0t_1$$

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 Oe на два непустых выражения Oe_1 и Oe_2 меньшего размера, таких, что

$$Oe = Oe_1 Oe_2$$

Теперь мы можем перевернуть каждое из выражений Oe_1 и Oe_2 по-отдельности. Пусть при этом получатся выражения Oe'_1 и Oe'_2 соответственно. Тогда ясно, что

$$Oe'_2 Oe'_1$$

равно результату обращения исходного выражения Oe .

Если Рефал реализован на многопроцессорной машине таким образом, что обращение выражений Oe_1 и Oe_2 можно выполнять одновременно, оказывается, что выгодно, чтобы Oe_1 и Oe_2 были приблизительно одинаковой длины. Тогда получаем следующий вариант описания функции, в котором используются библиотечные функции из модулей ACCESS и ARITHM (см. главу 3):

```
$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>>;
};
```

5. Неуспехи и аварии

5.1. Неудача при вычислении результатных выражений и троп

До сих пор мы считали, что результатом вычисления тропы Q является некоторое объектное выражение Oe . 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 $\$error(Oe)$, где Oe — некоторое объектное выражение, которое является сообщением об ошибке. Обычно Oe 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 `$error(0e)` обладают следующим свойством. Пусть нам требуется вычислить некоторую конструкцию. Тогда, если при вычислении некоторой составной части этой конструкции получается результат `$error(0e)`, дальнейшее вычисление конструкции прекращается и результатом вычисления всей конструкции считается `$error(0e)`. Единственным исключением из этого правила является конструкция `$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.

5.2. Перестройки

Рассмотрим следующую задачу: пусть нам задано некоторое объектное выражение `0e`, которое заведомо содержит на нулевом уровне не менее двух символов-литер '+' . Требуется разбить это выражение `0e` на три части `0eX`, `0eA` и `0eY`, такие, что `0e = 0eX '+' 0eA '+' 0eY`, и при этом `0eX` и `0eY` не содержат '+' на нулевом уровне скобок. Функцию, решающую данную задачу назовем "++". Тогда, например

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

Таким образом, требуется найти в `0e` самый левый '+' и самый правый '+'. Самый левый '+' легко найти с помощью сопоставления `0e` 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 "++"      eZ      = (eX)(eA)(eY);
$func "++Aux" (eX)(eP) = (eX)(eA)(eY);
"++"          $l eX '+' eP = <"++Aux" (eX)(eP)>;
"++Aux"      $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 $0e$, то $0e$ 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 $\rho_1, \rho_2, \dots, \rho_n$ — 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 O_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 "++", не использующее вспомогательную функцию "++Aux":

```
$func "++" eZ = (eX)(eA)(eY);
```

```
"++"
```

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

```
eP : $r eA '+' eY
```

```
= (eX)(eA)(eY);
```

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

$$\backslash \{ Q_1; Q_2; \dots Q_n; \}$$

где Q_1, Q_2, \dots, 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 $0e$, оно считается результатом всего распутья. Если же результатом вычисления тропы Q_1 является неуспех, то делается попытка вычислить $\backslash \{ Q_2; \dots, Q_n; \}$, 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

$$1 :: sX, \\ \backslash \{ sX : 0 = 1; sX : 1 = 0; \}$$

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:

$$\{ Q_1; Q_2; \dots Q_n; \}$$

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

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

Довольно часто встречаются тропы следующего вида:

$$S : Ve, \{ Ve : Snt_1; Ve : Snt_2; \dots, Ve : Snt_n; \}$$

$$S : Ve, \{ Ve : Snt_1; Ve : Snt_2; \dots, Ve : Snt_n; \}$$

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:

$$S : \{ Snt_1; Snt_2; \dots Snt_n; \}$$

$$S : \{ Snt_1; Snt_2; \dots Snt_n; \}$$

называемая *выбором*, при этом конструкции $\{ Snt_1; Snt_2; \dots Snt_n; \}$ и $\{ Snt_1; Snt_2; \dots Snt_n; \}$ 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 .

Так, например, 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:

$$sX : \{ 0 = 1; 1 = 0; \} :: sY = <"+" sX sY>$$

При этом сначала вычисляется источник $sX : \{ 0 = 1; 1 = 0; \}$, затем полученный результат присваивается переменной sY , после чего вычисляется тропа $= <"+" sX sY>$.

5.4. Управление перехватом неудач

Как мы видели выше, Рефал Плюс предоставляет довольно богатый набор конструкций для перехвата неудач. Однако, иногда возникает желание породить неудачу 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 неудач. А именно, всякое отсечение $\{ ! Q \}$ должно находиться внутри некоторого забора $\{ ? \dots \{ ! Q \} \dots \}$. 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 $\{ ? \dots \{ ! Q \} \dots \}$.

В частности, если X — неудача, то неудачей заканчивается и вычисление всей конструкции $\{ ? \dots \{ ! Q \} \dots \}$.

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 — '-' 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.

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.

Чтобы объяснить смысл правых частей, нам придется предварительно ввести некоторые понятия.

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

Если некоторая конструкция не является вассалом объемлющей конструкции, мы будем говорить, что она является *сувереном*. Например, в тропе $S :: \text{He } R$ источник S является сувереном, ибо его результат не считается результатом всей конструкции (хотя и может использоваться в R).

А именно, если определение функции имеет вид fname Palt , то образцовое распутье Palt является сувереном.

Если тропа имеет один из следующих видов: $S R$, $S :: \text{He } R$, $S : P R$ или $\# S R$, то источник S является сувереном.

Если источник имеет вид $S : \text{Palt}$, то источник S является сувереном.

Рассмотрим теперь некоторую конструкцию вместе со всеми объемлющими ее суверенами. Среди этих суверенов всегда существует наименьший, и мы будем называть его *покровителем* рассматриваемой конструкции. В частности, всякий суверен является своим собственным покровителем.

Теперь мы можем описать семантику правых частей.

Пусть покровителем правой части $= Q$ является некоторая объемлющая конструкция $\dots = Q \dots$. Тогда первым делом вычисляется тропа Q . Если это вычисление завершилось с

некоторым результатом X , то X считается результатом всего покровителя $\dots = Q \dots$.

В частности, если X — неуспех, то и значением покровителя $\dots = Q \dots$ будет неуспех, даже если внутри него расставлены ловушки, предназначенные для перехвата неуспехов.

Например, результатом вычисления тропы

$$\backslash \{ A \ B \ C : \$1 \ e \ sX \ e, \ sX : B, \ sX \} :: sY, \ sY$$

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

$$\backslash \{ A \ B \ C : \$1 \ e \ sX \ e = sX : B, \ sX \} :: sY, \ sY$$

вычисление которой терпит неудачу.

На использование заборов, отсечений и правых частей накладываются следующие ограничения.

И отсечение $\backslash ! Q$, и соответствующий ему забор $\backslash ? \dots \backslash ! Q \dots$, должны иметь общего покровителя.

На пути от $\backslash !$ к соответствующему ему $\backslash ?$ не должно быть ни одного $=$.

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 $Fname$ — безоткатная, то вычисление вызова $\langle Fname \ Re \rangle$ не может закончиться получением неуспеха. Если же функция $Fname$ — откатная, то вычисление вызова $\langle Fname \ Re \rangle$ вообще говоря может закончиться получением неуспеха в качестве результата.

До сих пор мы считали, что объявления функций имеют вид

$$\text{\$func Fname } F_{in} = F_{out};$$

однако это верно только в том случае, если функция Fname — безоткатная. Если же Fname — откатная, ее объявление должно иметь вид

$$\text{\$func? Fname } F_{in} = F_{out};$$

Теперь мы можем более точно описать семантику определений функций. Пусть определение функции Fname имеет вид

$$\text{Fname Palt}$$

где Palt — образцовое распутье вида $\setminus\{P_1 R_1; P_2 R_2; \dots P_n R_n;\}$ или $\{P_1 R_1; P_2 R_2; \dots P_n R_n;\}$, и пусть требуется вычислить ее вызов $\langle \text{Fname Re} \rangle$. Тогда прежде всего вычисляется результатное выражение Re. Если при этом получается неуспех, то вызов функции Fname не выполняется и считается, что результатом вычисления $\langle \text{Fname Re} \rangle$ является неуспех. Предположим теперь, что в результате вычисления Re получилось объектное выражение Oe. Тогда выполняется вызов функции, а именно, вычисляется источник

$$\text{Oe : Palt}$$

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 $\langle \text{Fname Re} \rangle$. Если же X является неуспехом, то дальнейшие действия зависят от того, является ли функция Fname откатной.

Если Fname — откатная, и X является неуспехом, то результатом вызова $\langle \text{Fname Re} \rangle$ считается неуспех.

Если Fname — безоткатная, и X является неуспехом, то этот неуспех "перехватывается" и преобразуется в ошибку $\text{\$error(Fname "Unexpected fail")}$.

6. Логические условия

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 "<" проверяет, что ее первый аргумент меньше второго. А именно, результатом вычисления вызова вида <"<" (Oe') (Oe'')> является пустое выражение, если объектное выражение Oe' "меньше" чем объектное выражение Oe''. Если же это не так — результатом является неуспех.

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

$$\$func? Fname F_{in} = ;$$

Дальше мы рассмотрим различные способы использования и комбинирования условий.

6.2. Ветвления

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

$$\backslash? \{S \backslash! Q'; \backslash! Q'';\}$$

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 $\backslash!$. 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

$$\backslash? \{S \backslash! = Q'; \backslash! = Q'';\}$$

его можно записать более кратко следующим образом:

$$\{S = Q'; = Q'';\}$$

(что мы и будем делать в дальнейшем).

Рассмотрим следующий пример. Определим функцию Min-Oe, которая сравнивает два объектных выражения Oe' и Oe'' и выдает Oe', если Oe' предшествует Oe''. Если же Oe' равно Oe'' или Oe'' предшествует Oe', то Min-Oe выдает Oe''.

$$\$func \text{Min-Oe} (eX)(eY) = e.\text{Min-X-Y};$$

$$\text{Min-Oe} (eX)(eY) =$$

$$\{$$

$$\text{"<" } (eX)(eY) >$$

$$= eX;$$

$$= eY;$$

$$\};$$

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 $\backslash\{Q;\}$. После этого ветвление может быть записано следующим образом:

$$\{ \backslash\{Q;\} \backslash! Q'; \backslash! Q''; \}$$

6.3. Логические связи

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 заданы тропами: Q' и Q'' достаточно вычислить тропу

$$\backslash \{ Q'; \}, Q''$$

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 заданы тропами: Q' и Q'' достаточно вычислить тропу

$$\backslash \{ Q'; Q''; \}$$

● Логическое "НЕ"

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

$$\# \backslash \{ Q; \}$$

которая является сокращенной записью для тропы $\# \backslash \{ Q; \}, .$

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.

6.4. Пример: программа дифференцирования

Опишем функцию, которая будет выполнять символьное дифференцирование формул [Хен 83]. 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 $D_x(e)$ — результат дифференцирования e по x . Тогда правила дифференцирования можно записать в виде

$$\begin{aligned} D_x(x) &= 1 \\ D_x(y) &= 0 && \text{(где } y \text{ не равно } x) \\ D_x(i) &= 0 \\ D_x(e' + e'') &= D_x(e') + D_x(e'') \\ D_x(e' * e'') &= e' * D_x(e'') + e'' * D_x(e') \end{aligned}$$

Прежде чем писать программу дифференцирования, нужно выбрать для формул представление в виде объектных выражений. Обозначим через « e » представление формулы e . Выберем следующее представление

$$\begin{aligned} \langle x \rangle &= x \\ \langle i \rangle &= i \\ \langle e' + e'' \rangle &= (\text{Sum } \langle e' \rangle \langle e'' \rangle) \\ \langle e' * e'' \rangle &= (\text{Prod } \langle e' \rangle \langle e'' \rangle) \end{aligned}$$

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

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

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:

$$\begin{array}{l} 0 + e'' \longrightarrow e'' \\ e' + 0 \longrightarrow e' \end{array}$$

$$\begin{array}{lcl}
 0 * e'' & \longrightarrow & 0 \\
 e' * 0 & \longrightarrow & 0 \\
 1 * e'' & \longrightarrow & e'' \\
 e' * 1 & \longrightarrow & e'
 \end{array}$$

(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>>;
  };
  };

```

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 $\{0s_1, 0s_2, \dots, 0s_n\}$ можно изобразить в виде объектного выражения

$$0s_1 \ 0s_2 \ \dots \ 0s_n$$

При этом оказывается, что одному и тому же множеству могут соответствовать несколько его изображений. Например, мно-

жество $\{John, Mary\}$ 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 Os , мы будем его изображать в виде символа Os . Если же элемент множества — множество X , мы будем изображать его в виде термина вида $\langle X \rangle$, где $\langle X \rangle$ — изображение множества X . Таким образом, множество $\{A, \{A, B\}, \{A\}\}$ можно изобразить, например, в виде выражения $A\ (A\ B)\ (A)$.

Теперь постараемся определить функцию-предикат $Eqset?$, 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.

A 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:

$$A = B \Leftrightarrow A \subseteq B \ \& \ B \subseteq A$$

$$A \subseteq B \Leftrightarrow (\forall x \in A)(x \in B)$$

Поэтому мы определим не одну, а четыре функции-предиката: $Eqset?$ будет проверять, что ее аргументы представляют одно и то же множество, $Subset?$ будет проверять, что множество, представляемое первым аргументом, является подмножеством множества, представляемого вторым аргументом,

$E1?$ будет проверять, что ее первый аргумент представляет элемент, входящий в множество, представляемое вторым аргументом, а $Eq1?$ 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 $Eq1?$ вынуждена вызвать функцию $Eqset?$. Таким образом, оказывается, что функция $Eqset?$ в конечном итоге рекурсивно определяется через себя.

```
$func? Eqset? (eA)(eB) = ;
$func? Subset? (eA)(eB) = ;
$func? E1? tX (eA) = ;
$func? Eq1? tX tY =;
```

$Eqset?$

```
(eA)(eB) = <Subset? (eA)(eB)><Subset? (eB)(eA)>;
```

$Subset?$ (eA)(eB) =

```
eA :
```

```
{
```

```
= ;
```

```
tX eR = <E1? tX (eB)><Subset? (eR)(eB)>;
```

```
};
```

$E1?$ tX (eA) =

```
eA : tY eR,
```

```
\{ <Eq1? tX tY>; <E1? tX (eR)>; };
```

$Eq1?$ tX tY =

```
\{
```

```
tX tY : s s
```

```
= tX : tY;
```

```
tX tY : (eA)(eB)
```

```
= <Eqset? (eA)(eB)>;
```

```
};
```

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 разбивается 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 79].

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 79], 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,
  {
    <"<=" (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)
  = {
    <"<=" (sA)(sB)>
      = sA <Merge (eXRest)(eY)>;
      = sB <Merge (eX)(eYRest)>;
  };
};

```

8. Функции выдающие несколько результатов

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

$\langle \text{NMB } A (B A) C A \rangle \Rightarrow 1 (2 3) 4 5$

Основная трудность здесь заключается в том, что встре-

тив 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 NMB-Exp и NMB-Term):

```
$func NMB      e.Exp      = e.Exp;
$func NMB-Exp  e.Exp sN   = e.Exp sN;
$func NMB-Term t.Exp sN   = t.Exp sN;
```

```
NMB e.Exp =
  <NMB-Exp e.Exp 1> :: e.Exp s,
  e.Exp;
```

```
NMB-Exp e.Exp sN =
  e.Exp :
  {
  = sN;
  tX e.Rest =
    <NMB-Term tX sN> :: tX sN,
    <NMB-Exp e.Rest sN> :: e.Rest sN,
    tX e.Rest sN;
  };
```

```
NMB-Term tX sN =
  tX :
  {
  s =
    sN <"+" sN 1>;
```

```
(eE) =
  <NMB-Exp eE sN> :: eE sN,
  (eE) sN;
};
```

8.2. Быстрая сортировка

Следующий пример — так называемая *быстрая сортировка* [АХУ 79], 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 $Q_1 Q_2 Q_3$, 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 Split-Aux
           sX (eS1)(eS2)(eS3) eS = (eS1)(eS2)(eS3);

QSort  eS =
{
  eS :
    = ;
  eS : s
    = eS;
```

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

```
Split sX eS
    = <Split-Aux sX ()()() eS>;
```

```
Split-Aux sX (eS1)(eS2)(eS3) eS =
  eS :
  {
  =
    (eS1)(eS2)(eS3);
  sY eRest =
    {
    <"<" (sY)(sX)>
      = <Split-Aux sX (eS1 sY)(eS2)(eS3) eRest>;
    <">" (sY)(sX)>
      = <Split-Aux sX (eS1)(eS2)(eS3 sY) eRest>;
      = <Split-Aux sX (eS1)(eS2 sY)(eS3) eRest>;
    };
  };
```

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:

$$S'' \text{ \$iter } S' :: \text{He } R$$

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 $S'' \text{ \$iter } S' :: He R$ эквивалентен тропе

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

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

$$\begin{aligned} S'' &:: He, \{ R; \\ & \quad S' :: He, \{ R; \\ & \quad \quad S' \text{ \$iter } S' :: He R; \\ & \quad \quad \quad \}; \} \end{aligned}$$

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

$$\begin{aligned} S'' &:: He, \{ R; \\ & \quad S' :: He, \{ R; \\ & \quad \quad S' :: He, \{ R; \end{aligned}$$

```
...
... };;}
```

Рассмотрим пример использования конструкции поиска — определение функции факториал:

```
$func Fact sN = sFact;

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

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

```
$func Fact sN = sFact;
$func Fact-Aux sR sK = sFact;

Fact sN
  = <Fact-Aux 1 sN>;

Fact-Aux sR sK =
{
  sK : 0
  = sR;
  = <Fact-Aux <"*" sR sK> <"-" sK 1>>;
};
```

То же самое изображается с помощью конструкции поиска следующим образом.

```
$func Fact sN = sFact;
```



```

Fact sN =
  1 sN
  $iter <"*" sR sK> <"-" sK 1>
    :: sR sK,
sK : 0,
  = sR;

```

10. Перебор с возвратом

10.1. Задача о ферзях

Рассмотрим классическую задачу о восьми ферзях [Хен 83]. 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).

	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

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 тropy

```
\{
<"+" sI sJ> :: sN1, <"+" sI1 sJ1> :: sN2, sN1 : sN2;
<"-" sI sJ> :: sN1, <"-" sI1 sJ1> :: sN2, sN1 : sN2;
}
```

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

$$I_1 I_2 \dots I_n$$

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

	1	2	3	4
1		●		
2				●
3	●			
4			●	

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, ..., k . Будем кодировать частично построенную позицию как последовательность чисел

$$I_1 I_2 \dots I_k$$

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

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

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

```
Attack? sI sJ ePos =.
```

```
ePos : $r eRest e, eRest : e sJ1,
```

```
<Length eRest> :: sI1,
```

```
\{
```

```
sI1 : sI;
```

```
sJ1 : sJ;
```

```

<"+" sI sJ> :: sN1, <"+" sI1 sJ1> :: sN2, sN1 : sN2;
<"-" sI sJ> :: sN1, <"-" sI1 sJ1> :: sN2, sN1 : sN2;
};

```

Отметим, что проверку $I'=I$ можно и убрать, поскольку в нашей программе функция `Attack?` 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 `Next-Queen?`, которая пытается добавить к частично построенной позиции нового ферзя, перебирая различные горизонтали. Если ферзя удастся поставить, но этот ферзь не последний, делается попытка поставить следующего ферзя и т.д. Если очередного ферзя поставить не удастся, происходит возврат назад и делается попытка переставить на новое место предыдущего ферзя.

```

$func? Next-Queen? sI sN ePos = ePos;

```

```

Next-Queen? sI sN ePos =
  1 $iter \{ <"<" (sJ) (sN)> = <"+" sJ 1>; }
  :: sJ,
  # <Attack? sI sJ ePos>,
  ePos sJ :: ePos,
  \? {
  sI : sN
  \! ePos;
  \! <Next-Queen? <"+" sI 1> sN ePos>;
  };

```

Некоторые тонкости в определении функции `Next-Queen?` заслуживают пояснения.

Во-первых, конструкция поиска делает попытку вычислить свое тело, последовательно пробуя для переменной J значения $1, 2, \dots, n$ 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 `Attack?`, а значит потерпит неудачу отрицание этого условия, либо очередного ферзя можно поставить на поле (I, J) , но не удастся разместить следующих ферзей, и тогда терпит неудачу рекурсивный вызов функции `Next-Queen?`.

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
  = <Next-Queen? 1 sN >;
```

10.2. Задача о цепочках:

Рассмотрим следующую задачу [Вир 77]. Требуется найти объектное выражение Oe , обладающее следующими свойствами.

- Oe не содержит скобок, а всякий входящий в него символ является одним из символов 1, 2 или 3.
- Oe имеет заданную длину len .
- Не существует таких объектных выражений Oe_a , Oe_b и Oe_c , что Oe_c — не пусто и при этом

$$Oe = Oe_a Oe_c Oe_c Oe_b$$
 т.е. Oe 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 $Oe_a Oe_c Oe_c Oe_b$, где Oe_c — непустое, достаточно после добавления каждого числа про-

верить, что не образовалось выражение вида

$$0e_a 0e_c 0e_c$$

Для распознавания выражений такого вида определим предикат `Unacceptable?`.

```
$func? Unacceptable? e.String = ;
Unacceptable? e.String =
  <Div <Length e.String> 2> :: s.Max,
  {
    s.Max : 0
      = $fail;
      = 1
      $iter \{ <"<" (sK) (s.Max)> = <"+" 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
        \{ <"<" (s.Digit) (3)> = <"+" s.Digit 1>; }
        :: s.Digit,
```

```

e.String s.Digit :: e.String,
# <Unacceptable? e.String>,
  <Extend? s.Len e.String>;
};

```

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

```
$func? Find-String? s.Len = e.String;
```

```
Find-String? s.Len
= <Extend? s.Len >;
```

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 80].

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

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 с помощью следующей грамматики (см. главу 2, 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 лек-

сем в одну, разделять их не обязательно.

Не всякая последовательность лексем является синтаксически правильной программой. Поэтому, после того как входная последовательность литер разбита на лексем, 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 St₁ ELSE St₂

проверяет условие Cond. Затем, если оно выполнено, то выпол-

няется 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
```

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 | MUL | DIV | LOAD | STORE |
$     ADDC | SUBC | MULC | 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 неза-
нятыми 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 опера-
ции. 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, MUL и DIV имеют вид Op, Addr; и
вычисляют соответственно сумму, разность, произведение и
целую часть частного двух чисел. При этом первое число бе-
рется из сумматора, второе — из ячейки с адресом Addr, а
результат записывается в сумматор.

Инструкции ADDC, SUBC, MULC и DIVC имеют вид Op, Int; и
вычисляют соответственно сумму, разность, произведение и
целую часть частного двух чисел. При этом первое число бе-
рется 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	MUL, 19;		
007	SUB, 21;	014	STORE, 20;		

Слева от директив выписаны соответствующие им адреса.

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 имеется 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.

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 Init-Scanner s.Channel = ;
$func Read-Token = s.TokenClass s.TokenInfo;
$func Term-Scanner = ;

```

Из него экспортируются три функции.

Функция `Init-Scanner` служит для инициализации сканера. Параметр `s.Channel` является ссылкой на канал, из которого сканер начинает читать литеры. Прежде чем обращаться к `Init-Scanner`, следует открыть этот канал на чтение.

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

Функция `Read-Token` 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: CMPPRS.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(0e)`, где `0e` — сообщение об ошибке.

Модуль CMPGEN имеет следующую интерфейсную часть:

```

**
** File: CMPGEN.RFI
**

$func Gen-Code    t.Program = t.Code;
$func Write-Code t.Code = ;

```

Из него экспортируются две функции.

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

Функция `Write-Code` 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 Make-Dic      = s.Dic;
$func Lookup-Dic   s.Key s.Dic = s.Ref;
$func Allocate-Dic s.Dic s.StartAddr = s.FreeAddr;

```

Из него экспортируются четыре функции.

Функция `Make-Dic` создает новый пустой словарь и выдает ссылку на него.

Функция `Lookup-Dic` 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 `Allocate-Dic` 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.

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.Ch1,  
  <Open-File s.Ch1 e.FileName "r">,  
  <Parse s.Ch1> :: t.AProgram,  
  <Close-Channel s.Ch1>,  
  <Gen-Code t.AProgram> :: t.Code,  
  <Write-Code t.Code>;
```

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 `CMPCSN` имеет следующую реализацию:

```

**
** File: CMPCSN.RF
**

$use STUDIO CLASS CONVERT BOX;

$func Scan-Token
    s.Ch1 e.Line =
        s.TokenKey s.TokenInfo (e.Line1);
$func Scan-Id-Rest
    (e.Id-Chars) e.Chars =
        s.TokenKey s.Word (e.Rest);
$func Scan-Int-Rest
    (e.Int-Chars) e.Chars =
        s.TokenKey s.Int (e.Rest);
$func? Blank?          s.Char = ;
$func? One-Char-Token? s.Char = ;

```

```
$func? Compound-Token?   s.Char e.Line = s.Word e.Rest;
$func? KeyWord?          s.Word = ;
```

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

```
$box Scan-Ch1 Scan-Line;
```

```
Init-Scanner s.Ch1 =           ** Инициализация сканера.
  <Store &Scan-Ch1 s.Ch1>,     ** Прячем канал в ящик.
  <Store &Scan-Line >;       ** Текущая строка – пустая.
```

```
Term-Scanner =                ** Завершение работы.
  <Store &Scan-Ch1 >,         ** Забываем канал
  <Store &Scan-Line >;       ** и текущую строку.
```

```
Read-Token =                  ** Чтение лексемы.
  <? &Scan-Ch1> : s.Ch1,
  <? &Scan-Line> :: e.Line,
  <Scan-Token s.Ch1 e.Line>
                                :: s.TokenKey s.TokenInfo (e.Line),
  <Store &Scan-Line e.Line>,
  = s.TokenKey s.TokenInfo;
```

```
Scan-Token s.Ch1 e.Line =
  e.Line :
  {
  =                               ** Остаток строки –
    {                             ** пустой. Читаем
    <Read-Line! s.Ch1> :: e.Line   ** следующую.
      = <Scan-Token s.Ch1 e.Line>;
      = Key Eof ();               ** Конец файла.
    };
  s.Char e.Rest =                ** Изучаем очередную
  {                               ** литеру.
  <Blank? s.Char>
    = <Scan-Token s.Ch1 e.Rest>;
```

```

<Letter? s.Char>
  = <Scan-Id-Rest (s.Char) e.Rest>;
<Digit? s.Char>
  = <Scan-Int-Rest (s.Char) e.Rest>;
<One-Char-Token? s.Char>
  = Key <To-Word s.Char> (e.Rest);
<Compound-Token? s.Char e.Rest> :: s.Word e.Rest
  = Key s.Word (e.Rest);
  = Char s.Char (e.Rest);      ** Неопознанная литера.
};
};

```

** Отщепляем остаток идентификатора.

```

Scan-Id-Rest (e.Id-Chars) e.Rest =
{
  e.Rest : s.Char e.Rest1,
    \{<Letter? s.Char>; <Digit? s.Char>;}
  = <Scan-Id-Rest (e.Id-Chars s.Char) e.Rest1>;
  = <To-Word <To-Upper e.Id-Chars>> : s.Word,
    {<Keyword? s.Word> = Key; = Name;} :: s.TokenKey,
  = s.TokenKey s.Word (e.Rest);
};

```

** Отщепляем остаток целого числа.

```

Scan-Int-Rest (e.Int-Chars) e.Rest =
{
  e.Rest : s.Char e.Rest1, <Digit? s.Char>
  = <Scan-Int-Rest (e.Int-Chars s.Char) e.Rest1>;
  = Value <To-Int e.Int-Chars> (e.Rest);
};

```

```

Blank? s.Char =      ** "Межевая" литера?
' \n\t' : e s.Char e;

```

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

Compound-Token? ** 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;
};

KeyWord? ** Идентификатор – ключевое слово?

\{
DO ; ELSE ; IF ; READ ; THEN ; WHILE ; WRITE ;
};

11.7. Синтаксический анализатор

Задача синтаксического анализатора, находящегося в модуле SMPPRS, — 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:

\$ Предложение = Подлежащее Сказуемое.

\$ Подлежащее = "кошки" | "собаки".

\$ Сказуемое = "спят" | "едят".

Пусть нам дана последовательность лексем

"собаки" "едят"

и мы хотим узнать, является ли она правильным предложением.

Для этого мы должны узнать, может ли эта цепочка быть порождена нетерминалом Предложение. Но из грамматики следует, что множество цепочек, порождаемых нетерминалом Предложение, совпадает с множеством цепочек, порождаемых цепочкой двух нетерминалов Подлежащее Сказуемое. Таким образом, задача сводится к тому, чтобы проверить, нельзя ли от входной цепочки отщепить начало, которое порождается нетерминалом Подлежащее, а затем проверить, что остаток цепочки может быть порожден нетерминалом Сказуемое.

Как отщепить от цепочки начало, порождаемое нетерминалом Подлежащее? Очевидно, что для этого достаточно проверить, что цепочка лексем начинается с лексемы "кошки" или лексемы "собаки".

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 85].

Руководствуясь вышеизложенными соображениями мы можем определить функцию "Предложение", отщепляющую от цепочки символов начальную цепочку, выводимую из нетерминала Предло-

жение, либо терпящую неудачу, если это невозможно.

```
$func? "Предложение" e.Token = e.Rest;
$func? "Подлежащее" e.Token = e.Rest;
$func? "Сказуемое" e.Token = e.Rest;
$func? Token? s? e.Token = e.Rest;
```

```
"Предложение" eZ =
  <"Подлежащее" eZ> :: eZ,
  <"Сказуемое" eZ> :: eZ,
  = eZ;
```

```
"Подлежащее" eZ =
  \{
  <Token? "собаки" eZ> :: eZ = eZ;
  <Token? "кошки" eZ> :: eZ = eZ;
  };
```

```
"Сказуемое" eZ =
  \{
  <Token? "спят" eZ> :: eZ = eZ;
  <Token? "едят" 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 $\$error(Oe)$ где Oe 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.Comp-Oper t.Expr t.Expr).
$ t.Expr =
$     (Const s.Value) |
$     (Name s.Name) |
$     (Op t.Oper t.Expr t.Expr).
$ s.Comp-Oper = Eq | Ne | Gt | Ge | Lt | Le.
$ s.Oper = Add | Sub | Div | Mul.
$ s.Name = s.Word.
```

\$ s.Value = s.Int.

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

$$(S_{\text{KeyWord}} \text{Ot}_1 \text{Ot}_2 \dots \text{Ot}_n)$$

где ключевое слово S_{KeyWord} — это символ-слово, являющееся названием конструкции, а объектные термы $\text{Ot}_1, \text{Ot}_2, \dots, \text{Ot}_n$ — 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 Statement-Seq	sC sI	= sC sI tX;
\$func Rest-St-Seq	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 Rest-Expr	sC sI tX1	= sC sI tX;
\$func Term	sC sI	= sC sI tX;
\$func Rest-Term	sC sI tX1	= sC sI tX;
\$func Factor	sC sI	= sC sI tX;
\$func Comp-Op	sC sI	= sC sI s.Comp-Oper;
\$func? Add-Op?	sC sI	= sC sI s.Oper;
\$func? Mul-Op?	sC sI	= sC sI s.Oper;
\$func? Token?	sI? sC sI	= sC sI ;
\$func Accept	sI? sC sI	= sC sI ;
\$func? Name?	sC sI	= sC sI s.Name;
\$func? Value?	sC sI	= sC sI s.Value;

```

Parse s.Ch1 =
  <Init-Scanner s.Ch1>,
  <Program <Read-Token>> :: sC sI t.Program,
  <Term-Scanner>,
  {
    ** Проверяем, что остаток
    sC sI : Key Eof      ** программы пуст.
    = t.Program;
    = $error sC sI " instead of Eof after the program";
  };

```

```

Program sC sI =
  ** Программа.
  <Statement-Seq sC sI> :: sC sI tX,
  = sC sI (Program tX);

```

```

Statement-Seq sC sI =
  ** Последовательность
  <Statement sC sI> :: sC sI tX0, ** операторов.
  = <Rest-St-Seq sC sI tX0>;

```

```

Rest-St-Seq sC sI tX0 =
  \? {
  <Token? ";" sC sI> :: sC sI \!
    <Statement-Seq 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 \!
    <Statement-Seq 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,
    <Comp-Op 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,
    = <Rest-Expr sC sI t.X0>;

```

```

Rest-Expr sC sI t.X1 =
    \? {
    <Add-Op? sC sI> :: sC sI s.Op \!

```

```

<Term sC sI> :: sC sI t.X2,
    = <Rest-Expr sC sI (Op s.Op t.X1 t.X2)>;
\!
    = sC sI t.X1;
};

```

```

Term sC sI = ** Слагаемое.
    <Factor sC sI> :: sC sI t.X0,
    = <Rest-Term sC sI t.X0>;

```

```

Rest-Term sC sI t.X1 =
    \? {
    <Mul-Op? sC sI> :: sC sI s.Op \!
        <Factor sC sI> :: sC sI t.X2,
        = <Rest-Term 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;
    };

```

```

Comp-Op sC sI = ** Операция отношения.
    {
    sC : Key,

```

```

( "=" Eq) ("<>" Ne) ("<=" Le) ("<" Lt) (">=" Ge) (">" Gt)
: e (sI s.Op) e
  = <Read-Token> s.Op;
  = $error "Invalid comparison operation: " sC sI;
};

```

Add-Op? Key sI, ** Аддитивная операция.

```

( "+" Add) ("-" Sub) : e (sI s.Op) e
  = <Read-Token> s.Op;

```

Mul-Op? Key sI, ** Мультипликативная операция.

```

( "*" Mul) ("/" Div) : e (sI s.Op) e
  = <Read-Token> s.Op;

```

** Эта функция пытается отщепить ключевое слово sI?.

** Если это не удастся, то результатом является неуспех.

```

Token? sI? Key sI? = <Read-Token>;

```

** Эта функция пытается отщепить ключевое слово sI?.

** Если это не удастся, то результатом является ошибка.

Accept

```

{
  sI? Key sI? = <Read-Token>;
  sI? sC sI = $error sC sI " instead of " Key sI?;
};

```

** Имя переменной.

```

Name? Name sI = <Read-Token> sI;

```

** Значение.

```

Value? Value sI = <Read-Token> sI;

```

11.8. Генератор кода

При генерации машинного кода порождается его промежуточное представление в виде объектного термина, 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 | Mul | Load | Store |
$       Addc | Subc | Divc | Mulc | 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 на содержимое ящиков.

Модуль CMPGEN имеет следующую реализацию:

```

**
** File: CMPGEN.RF
**

```

```
$use STDIO CLASS ARITHM BOX;
```

```
$use CMPDIC;
```

```
$func Enc-Program      t.Program s.Dic = t.Code;
$func Enc-St           t.St s.Dic = t.Code;
$func Enc-Test         t.Test s.Label s.Dic = t.TestC;
$func Unless-Op        s.Op = s.Jump-If;
$func Enc-Expr         t.Expr s.Dic = t.ExprC;
$func Enc-Sub-Expr     t.Expr sN s.Dic = t.ExprC;
$func Literal-Op       s.Op = s.OpCode;
$func Memory-Op        s.Op = s.OpCode;
$func Assemble         t.Code s.StartAddr = s.FreeAddr;
$func Assemble-Seq    e.CodeSeq s.Addr = s.FreeAddr;
$func Dereference      t.Code = t.Target;
$func Dereference-Seq e.CodeSeq = e.CodeSeqD;
$func Write-Code-Seq  e.CodeSeq = ;
```

```
** Генерация кода из абстрактной программы.
```

```
Gen-Code t.Program =
```

```
  ** Создается пустой словарь.
```

```
<Make-Dic> :: s.Dic,
```

```
  ** Компилируется абстрактная программа.
```

```
<Enc-Program t.Program s.Dic> :: t.Code,
```

```
  ** Распределяется память под инструкции.
```

```
<Assemble t.Code 1> :: s.FreeAddr,
```

```
  ** Распределяется память под переменные.
```

```
<Allocate-Dic s.Dic s.FreeAddr> :: s.EndAddr,
```

```
  ** Метки заменяются на адреса.
```

```
<Dereference t.Code> :: t.CodeD,
```

```
  ** Генерируется директива BLOCK.
```

```
<"-" s.EndAddr s.FreeAddr> :: s.BlockLength,
```

```
(Seq t.CodeD (Block s.BlockLength)) :: t.Target,
= t.Target;
```


** Компиляция программы.

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

** Компиляция оператора.

```
Enc-St (s.KeyWord e.Info) s.Dic =
  (s.KeyWord e.Info) :
  {
  (Assign sX t.Expr) =
    <Lookup-Dic sX s.Dic> :: s.Addr,
    <Enc-Expr 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,
    <Enc-Test t.Test s.L1 s.Dic> :: t.TestC,
    <Enc-St t.Then s.Dic> :: t.ThenC,
    <Enc-St 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,
    <Enc-Test t.Test s.L2 s.Dic> :: t.TestC,
    <Enc-St t.Do s.Dic> :: t.DoC,
    = (Seq
```

```

        (Label s.L1)
          t.TestC
          t.DoC
          (Instr Jump s.L1)
        (Label s.L2)
      );
(Read s.X) =
  <Lookup-Dic s.X s.Dic> :: s.Addr,
  = (Instr Read s.Addr);
(Write t.Expr) =
  <Enc-Expr t.Expr s.Dic> :: t.ExprC,
  = (Seq t.ExprC (Instr Write 0));
(Seq t.St1 t.St2) =
  <Enc-St t.St1 s.Dic> :: t.StC1,
  <Enc-St t.St2 s.Dic> :: t.StC2,
  = (Seq t.StC1 t.StC2);
(Skip) =
  = (Seq );
};

```

**** Компиляция условия.**

```

Enc-Test (Test s.Op t.Arg1 t.Arg2) s.Label s.Dic =
  <Enc-Expr (Op Sub t.Arg1 t.Arg2) s.Dic> :: t.ExprC,
  <Unless-Op s.Op> :: s.Jump-If,
  = (Seq t.ExprC (Instr s.Jump-If s.Label));

```

Unless-Op **** 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.

Enc-Expr t.Expr s.Dic

= <Enc-Sub-Expr t.Expr 0 s.Dic>;

Enc-Sub-Expr (s.KeyWord e.Info) sN s.Dic =

(s.KeyWord e.Info) :

{

(Const sC) =

= (Instr Loadc sC);

(Name sX) =

<Lookup-Dic sX s.Dic> :: s.Addr,

= (Instr Load s.Addr);

(Op s.Op t.Expr1 t.Expr2) =

t.Expr2 :

{

(Const sC2) =

<Enc-Sub-Expr t.Expr1 sN s.Dic> :: t.Expr1C,

<Literal-Op s.Op> :: s.OpCode,

= (Seq t.Expr1C (Instr s.OpCode sC2));

(Name sX2) =

<Enc-Sub-Expr t.Expr1 sN s.Dic> :: t.Expr1C,

<Memory-Op s.Op> :: s.OpCode,

<Lookup-Dic sX2 s.Dic> :: s.Addr,

= (Seq t.Expr1C (Instr s.OpCode s.Addr));

(Op e) =

<Lookup-Dic sN s.Dic> :: s.Addr,

<Enc-Sub-Expr t.Expr2 sN s.Dic> :: t.Expr2C,

<"+" sN 1> :: sN1,

<Enc-Sub-Expr t.Expr1 sN1 s.Dic> :: t.Expr1C,

<Memory-Op s.Op> :: s.OpCode,

= (Seq

```

        t.Expr2C
        (Instr Store s.Addr)
        t.Expr1C
        (Instr s.OpCode s.Addr)
    );
};
};

```

```

Literal-Op                                ** Генерация имен инструкций
{                                           ** с непосредственным операндом.
Add = Addc; Sub = Subc;
Mul = Mulc; Div = Divc;
};

```

```

Memory-Op                                  ** Генерация имен инструкций
{                                           ** с операндом-адресом.
Add = Add; Sub = Sub;
Mul = Mul; Div = Div;
};

```

**** Распределение памяти под инструкции.**

```

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

```

```

Assemble-Seq e.CodeSeq s.A0 =
e.CodeSeq :

```

```

{
t.Code e.Rest =
  <Assemble t.Code s.A0> :: s.A1,
  = <Assemble-Seq e.Rest s.A1>;
=
  = s.A0;
};

```

**** Замена меток на их значения.**

```

Dereference t.Code =
  t.Code :
  {
  (Seq e.CodeSeq) =
    (Seq <Dereference-Seq e.CodeSeq>);
  (Instr s.Instr s.Value) =
    {
    <Int? s.Value>
      = t.Code;
    <Box? s.Value>
      = (Instr s.Instr <? s.Value>);
    };
  (Label s.Label) =
    (Label <? s.Label>);
  };

```

Dereference-Seq

```

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

```

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

Write-Code

```

{
(Seq e.CodeSeq) =
  <Write-Code-Seq 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">;
};

```

Write-Code-Seq

```

{
t.Code e.CodeSeq =
  <Write-Code t.Code><Write-Code-Seq e.CodeSeq>;
= ;
};

```

11.9. Модуль работы со словарем

Словари реализованы в виде бинарных деревьев [АХУ 79]. 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.**

```
Make-Dic = <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.**

```
Lookup-Dic s.Key s.Dic =
```

```
<? 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)> :
```

```
{
```

```
'<' = <Lookup-Dic s.Key s.DicL>;
```

```
'>' = <Lookup-Dic 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.**

```
Allocate-Dic s.Dic s.A =
```

```
<? s.Dic> :
```

```
{
```

```
= s.A;
```

```
s.Key s.Ref s.DicL s.DicR =
```

```
<Allocate-Dic s.DicL s.A> :: s.A,
```

```
<Store s.Ref s.A>,
```

```

    <"+" s.A 1> :: s.A,
      = <Allocate-Dic s.DicR s.A>;
};

```

```

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

```


Глава 2

СИНТАКСИС И СЕМАНТИКА РЕФАЛА ПЛЮС

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:

$$A = B \mid C.$$

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.

$(A B)(C D)$	$A C, A D, B C, B D$
$A\{B C$	$A B C, A C$
$A \{B A\}$	$A, A B A, A B A B A, A B A B A B A, \dots$
$\{A B\} C$	$C, A C, B C, A A C, A B C, B B C, B A C, \dots$

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. 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:

$$\rho, \sigma \vdash E \Rightarrow \text{true}, \sigma'$$

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

$$\rho, \sigma \vdash \text{if } E \text{ then } E' \text{ else } E'' \Rightarrow X, \sigma''$$

$$\rho, \sigma \vdash E \Rightarrow \text{false}, \sigma'$$

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

$$\rho, \sigma \vdash \text{if } E \text{ then } E' \text{ else } E'' \Rightarrow X, \sigma''$$

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

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.

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 литеру *. При этом весь остаток строки игнорируется. Второй способ — заключить комментарий в пару "комментарных скобок" /* и */. Например:

* Это — комментарий.
 * И это — комментарий.
 /* И это — тоже! */

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.

3.3. Ключевые слова

\$ КлючевоеСлово =
 \$ "\$box" | "\$channel" | "\$const" | "\$error" |
 \$ "\$fail" | "\$func" | "\$func?" |

```

$      "$iter" | "$l" | "$r" | "$string" |
$      "$stable" | "$trace" | "$traceall" |
$      "$strap" | "$use" | "$vector" | "$with" |
$      "#" | "&" | "," | ":" | "::" | ";" | "=" |
$      "\?" | "\!".

```

Если ключевое слово начинается с литеры **\$**, не делается различия между прописными и строчными буквами. Например, ниже одно и то же ключевое слово записано тремя разными способами:

```
$func $Func $FUNC
```

3.4. Символы–литеры

```

$      ИзображениеЦепочкиЛитер =
$      "' ' { ИзображениеЛитеры | ПереходНаНовуюСтроку } "' .
$      ИзображениеЛитеры =
$      ИзображениеОбычнойЛитеры | ИзображениеОсобойЛитеры.
$      ИзображениеОбычнойЛитеры =
$      любая литера ASCII кроме апострофа ('),
$      кавычки ("), обратной косой (\) и конца строки.
$      ИзображениеОсобойЛитеры =
$      "\n" | "\t" | "\v" | "\b" | "\r" | "\f" |
$      "\\ " | "\'" | '\ "' .
$      ПереходНаНовуюСтроку =
$      "\ " КонецСтроки .

```

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

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

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

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

Последовательность из нескольких символов–литер может быть записана в виде одной последовательности изображений литер ASCII, 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'
```

Внутри изображения цепочки символов–литер литера \ и следующая за ней литера перевода строки (LF) игнорируются, что дает возможность переносить цепочки с одной строки на другую. Например:

```
'A\
BC'
```

3.5. Символы–слова

```
$   ИзображениеСлова =
$   Идентификатор |
```



```

$      '"" { ИзображениеЛитеры | ПереходНаНовуюСтроку } '"" .
$      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".
$      Цифра = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7"
$          | "8" | "9".

```

Символ–слово соответствует цепочке литер кода ASCII и записывается в виде цепочки изображений этих литер ASCII, заключенной в двойные кавычки. Изображения литер ASCII 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 ASCII: прописные буквы, строчные буквы, цифры, восклицательный знак, вопросительный знак, знак минус.

Во-вторых, этот символ—слово должен начинаться с прописной буквы, восклицательного знака или вопросительного знака.

Если символ—слово записан без окружающих двойных кавычек, то все строчные буквы считаются изображениями соответствующих прописных букв, т.е. "поднимаются в верхний регистр".

Например, ниже приведено три изображения одного и того же символа—слова:

```
I-do-not-like-swimming!
I-DO-NOT-LIKE-SWIMMING!
"I-DO-NOT-LIKE-SWIMMING!"
```

3.6. Символы—числа

\$ ИзображениеЧисла = ["+" | "-"] Цифра { Цифра }.

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

```
123   +121   -123   -123456789012345678901234567890
```

3.7. Переменные

\$ Переменная =

\$ s-переменная | t-переменная | e-переменная |

\$ v-переменная.

\$ s-переменная = "s" ["."] ИндексПеременной.

\$ t-переменная = "t" ["."] ИндексПеременной.

\$ v-переменная = "v" ["."] ИндексПеременной.

- § е-переменная = "e" ["."] ИндексПеременной.
- § УказательТипаПеременной = "s" | "t" | "v" | "e".
- § ИндексПеременной = ХвостИдентификатора.

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

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

Различие между прописными и строчными буквами в записи переменных несущественно. Например, eI, e.I, 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.

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	⇒	Литера ₁ Литера ₂ ... Литера _n
ИзображениеСлова	⇒	Слово
ИзображениеЧисла	⇒	Число

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

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.

5. Объектные выражения

5.1. Синтаксис объектных выражений

§ ОбъектноеВыражение = { ОбъектныйТерм }.

§ ОбъектныйТерм = Символ | "(" ОбъектноеВыражение ")".

В дальнейшем мы будем обозначать объектные выражения через Oe , объектные термы — через Ot , а символы — через Os .

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 исполнению 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.

5.3. Символические имена выражений

§ ИмяВыражения = "&" Слово.

Объектным выражениям в программе на Рефале Плюс могут присваиваться символические имена, в качестве которых используются символы-слова. Если символ-слово Os обозначает объектное выражение Oe , то конструкция вида $\& Os$ означает, что в этом месте должно быть подставлено объектное выражение Oe .

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.

5.4. Устранение символических имен выражений

Если слово Os является символическим именем объектного выражения Oe , то все использования этого имени в программе можно устранить, заменив каждую конструкцию $\&Os$ на Oe .

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.

6. Значения переменных и среды

Вычисление любой конструкции Рефала Плюс требует знания значений переменных, входящих в эту конструкцию. Эта информация о значениях переменных естественным образом представляется в виде функции с конечной областью определения, ста-

вящей в соответствие каждой переменной из ее области определения значение этой переменной.

А именно, пусть ρ — среда с областью определения $\{V_1, \dots, V_n\}$. Тогда $\rho(V_j) = 0e_j$ является значением переменной V_j . Такую среду мы будем обозначать через $\{V_1 = 0e_1, \dots, V_n = 0e_n\}$. В частности, пустая среда будет обозначаться через $\{\}$.

Область определения среды ρ мы будем обозначать через $\text{dom}[\rho]$. Таким образом, $\text{dom}[\{V_1 = 0e_1, \dots, V_n = 0e_n\}] = \{V_1, \dots, V_n\}$.

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.

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

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

Например

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

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

7. Результатные выражения

7.1. Синтаксис

```

$   РезультатноеВыражение =
$       { РезультатныйТерм | ИмяВыражения }.
$   РезультатныйТерм =
$       СтатическийСимвол | Переменная |
$       "(" РезультатноеВыражение ")" |
$       ВызовФункции.
$   ВызовФункции =
$       "<" ИмяФункции АргументВызова ">".
$   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 $Fname$.

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 Oe , либо неуспех $\$fail(0)$, либо ошибка $\$error(Oe)$, где Oe — 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 $\langle \text{Fname } Re \rangle$, то вычисляется результатное выражение Re . Если в результате получится объектное выражение Oe , то функция Fname применяется к Oe .

Запись $\sigma \vdash \langle \text{Fname } Oe \rangle \Rightarrow X, \sigma'$ означает, что результатом применения функции Fname к объектному выражению Oe является X , причем если вычисление началось в глобальном состоянии σ , оно закончится в глобальном состоянии σ' .

$$\rho, \sigma \vdash \Rightarrow , \sigma$$

$$\rho, \sigma \vdash Re \Rightarrow Oe', \sigma'$$

$$\rho, \sigma' \vdash Rt \Rightarrow Oe'', \sigma''$$

$$\rho, \sigma \vdash Re Rt \Rightarrow Oe' Oe'', \sigma''$$

$$\rho, \sigma \vdash Re \Rightarrow Oe', \sigma'$$

$$\rho, \sigma' \vdash Rt \Rightarrow \$fail(0), \sigma''$$

$$\rho, \sigma \vdash Re Rt \Rightarrow \$fail(0), \sigma''$$

$$\rho, \sigma \vdash Re \Rightarrow Oe', \sigma'$$

$$\rho, \sigma' \vdash Rt \Rightarrow \$error(Oe''), \sigma''$$

$$\rho, \sigma \vdash Re Rt \Rightarrow \$error(Oe''), \sigma''$$

$$\rho, \sigma \vdash Re \Rightarrow \$fail(0), \sigma'$$

$$\rho, \sigma \vdash Re Rt \Rightarrow \$fail(0), \sigma'$$

$$\rho, \sigma \vdash Re \Rightarrow \$error(Oe'), \sigma'$$

$$\rho, \sigma \vdash Re Rt \Rightarrow \$error(Oe'), \sigma'$$

$$\rho, \sigma \vdash 0s \Rightarrow 0s, \sigma$$

$$\rho, \sigma \vdash V \Rightarrow 0e, \sigma$$

где $0e = \rho(V)$.

$$\rho, \sigma \vdash Re \Rightarrow 0e, \sigma'$$

$$\rho, \sigma \vdash (Re) \Rightarrow (0e), \sigma'$$

$$\rho, \sigma \vdash Re \Rightarrow \$fail(0), \sigma'$$

$$\rho, \sigma \vdash (Re) \Rightarrow \$fail(0), \sigma'$$

$$\rho, \sigma \vdash Re \Rightarrow \$error(0e), \sigma'$$

$$\rho, \sigma \vdash (Re) \Rightarrow \$error(0e), \sigma'$$

$$\rho, \sigma \vdash Re \Rightarrow 0e, \sigma'$$

$$\sigma' \vdash \langle Fname\ 0e \rangle \Rightarrow X, \sigma''$$

$$\rho, \sigma \vdash \langle Fname\ Re \rangle \Rightarrow X, \sigma''$$

$$\rho, \sigma \vdash Re \Rightarrow \$fail(0), \sigma'$$

$$\rho, \sigma \vdash \langle Fname\ Re \rangle \Rightarrow \$fail(0), \sigma'$$

$$\rho, \sigma \vdash Re \Rightarrow \$error(0e), \sigma'$$

$$\rho, \sigma \vdash \langle Fname\ Re \rangle \Rightarrow \$error(0e), \sigma'$$

7.3. Примеры

Ниже приведены примеры результатных выражений:

(A B) C D

t.Head e.Tail

While t.Condition Do t.Statement

<"*" sN <Factorial <"-" sN 1>>

Следующие результатные выражения являются результатными термами:

(A B)
 t.Head
 <"*" sN <Factorial <"-" sN 1>>

Пусть $\rho = \{sM = 2, sN = 3, eA = A B C, tB = (D E F)\}$,
 и пусть "+" — имя функции, выполняющей сложение целых чисел,
 а "*" — имя функции, выполняющей умножение целых чисел,
 т.е. в частности имеет место

$$\sigma \vdash \langle "+" \ 3 \ 100 \rangle \Rightarrow 103, \sigma$$

$$\sigma \vdash \langle "*" \ 2 \ 103 \rangle \Rightarrow 206, \sigma$$

для любого глобального состояния σ , ибо функции "+" и "*" не изменяют глобальное состояние.

Тогда имеем

$$\rho, \sigma \vdash eA (eA tB) tB \Rightarrow A B C (A B C (D E F)) (D E F), \sigma$$

$$\rho, \sigma \vdash \langle "*" \ sM \langle "+" \ sN \ 100 \rangle \rangle \Rightarrow 206, \sigma$$

для любого глобального состояния σ .

8. Образцы

8.1. Синтаксис

\$ Образец = УказательНаправления ОбразцовоеВыражение.

\$ УказательНаправления = ["\$l" | "\$r"].

\$ ОбразцовоеВыражение =

\$ { ОбразцовыйТерм | ИмяВыражения }.

\$ 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 "\$1".

В дальнейшем мы будем обозначать образцы через P , образцовые выражения — через P_e , а образцовые термы — через P_t .

8.2. Сопоставление с образцом

Мы говорим, что среда ρ' является результатом сопоставления объектного выражения O_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 O_e .

В этом случае среду ρ' мы будем также называть вариантом сопоставления O_e с P в среде ρ , а множество таких вариантов сопоставления будем обозначать через $M(\rho, O_e, P)$.

На множестве $M(\rho, O_e, P)$ вводится отношение порядка следующим образом.

Пусть $M(\rho, O_e, P)$ содержит два различных варианта сопоставления ρ' и ρ'' . Рассматриваются все вхождения переменных в P .

Если P имеет указатель направления $\$1$, 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 $\rho_1, \rho_2, \dots, \rho_n$ будем записывать в виде $[\rho_1, \rho_2, \dots, \rho_n]$, пустую последовательность — в виде $[]$.

Запись $[\rho_0]^\wedge[\rho_1, \dots, \rho_n]$ будет обозначать $[\rho_0, \rho_1, \dots, \rho_n]$.

Запись $\rho \vdash 0e : P \Rightarrow [\rho_1, \dots, \rho_n]$ будет означать, что $M(\rho, 0e, P) = \{\rho_1, \dots, \rho_n\}$ и при этом ρ_i предшествует ρ_j для всех $i < j$.

8.3. Примеры

Ниже приведены примеры образцов:

```
t.Head e.Tail
eX (eY)
eA '+' eB
$1 eA '+' eB
$r eA '+' eB
```

Далее приведены примеры сопоставления с образцом:

```
{ } \vdash A ( ) C D E : $1 sX tY tZ e1
\Rightarrow [ {sX = A, tY = ( ), tZ = C, e1 = D E} ]
```

```
{ } \vdash 1 2 3 : $1 eA eB \Rightarrow [
    {eA = , eB = 1 2 3},
    {eA = 1, eB = 2 3},
    {eA = 1 2, eB = 3},
    {eA = 1 2 3, eB = } ]
```

$$\{ \} \vdash 1 \ 2 \ 3 : \$r \ eA \ eB \Rightarrow [$$

{eA = 1 2 3,	eB =	},
{eA = 1 2,	eB = 3	},
{eA = 1,	eB = 2 3	},
{eA = ,	eB = 1 2 3	}]

$$\{eA = 1 \ 2\} \vdash \$l \ 1 \ 2 \ 3 \ 4 \ 5 : eA \ eB$$

$$\Rightarrow [\{eA = 1 \ 2, \ eB = 3 \ 4 \ 5\}]$$

9. Жесткие выражения

9.1. Синтаксис

\$ ЖесткоеВыражение =
\$ { ЖесткийКрай } |
\$ { ЖесткийКрай } e-переменная { ЖесткийКрай } |
\$ { ЖесткийКрай } v-переменная { ЖесткийКрай } .
\$ ЖесткийКрай = { ЖесткийТерм | ИмяВыражения } .
\$ 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.

Все переменные имеющие индексы и входящие 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 .

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 Oe либо сопоставление Oe с He невозможно, либо может быть сделано только одним способом. Таким образом, имеет место либо $\{ \} \vdash Oe : He \Rightarrow []$, либо $\{ \} \vdash Oe : He \Rightarrow [\rho]$.

Запись $\rho \vdash Oe :: He \Rightarrow \rho'$ будет означать, что $\{\} \vdash Oe$: $He \Rightarrow \rho''$ и $\rho' = \rho + \rho''$, т.е. что ρ' получается из ρ следующим образом. Сначала Oe 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' .

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} \vdash X Y Z :: sY eA
 \Rightarrow {sX = XXX, eA = Y Z, sY = X}

{sX = XXX, eA = A B C} \vdash X Y Z :: eA sY
 \Rightarrow {sX = XXX, eA = X Y, sY = Z}

10. Тропы

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.

А именно, 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.

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 и еще не закрытых отсечениями "\!".

Если вычисление тропы заканчивается, то результатом вычисления является либо объектное выражение Oe , либо неуспех $\$fail(k)$, где целое неотрицательное k характеризует "степень серьезности" неуспеха, либо ошибка $\$error(Oe)$, где Oe — сообщение об ошибке.

Если тропа находится на уровне m , а результатом ее вычисления является неуспех $\$fail(k)$, "серьезность неуспеха" всегда удовлетворяет условию $0 \leq k \leq m+1$. 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:

$$Q \Rightarrow Q'$$

10.3. Условия

Тропа $S R$ означает, что следует вычислить источник S , а затем, в случае успеха, вычислять хвост R .

$$\frac{\rho, 0, \sigma \vdash S \Rightarrow \sigma' \quad \rho, m, \sigma' \vdash R \Rightarrow X, \sigma''}{\rho, m, \sigma \vdash S R \Rightarrow X, \sigma''}$$

$$\frac{\rho, 0, \sigma \vdash S \Rightarrow \text{\$fail}(k), \sigma'}{\rho, m, \sigma \vdash S R \Rightarrow \text{\$fail}(0), \sigma'}$$

$$\frac{\rho, 0, \sigma \vdash S \Rightarrow \text{\$error}(0e), \sigma'}{\rho, m, \sigma \vdash S R \Rightarrow \text{\$error}(0e), \sigma'}$$

10.4. Присваивания

Тропа $S :: \text{He } R$ 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.

$$S :: \text{He} \Rightarrow S :: \text{He} ,$$

$$\frac{\rho, 0, \sigma \vdash S \Rightarrow 0e, \sigma' \quad \rho \vdash 0e :: \text{He} \Rightarrow \rho' \quad \rho', m, \sigma' \vdash R \Rightarrow X, \sigma''}{\rho, m, \sigma \vdash S :: \text{He } R \Rightarrow X, \sigma''}$$

$$\frac{\rho, 0, \sigma \vdash S \Rightarrow \text{\$fail}(k), \sigma'}{\rho, m, \sigma \vdash S :: \text{He } R \Rightarrow \text{\$fail}(0), \sigma'}$$

$$\frac{\rho, 0, \sigma \vdash S \Rightarrow \text{\$error}(0e), \sigma'}{\rho, m, \sigma \vdash S :: \text{He } R \Rightarrow \text{\$error}(0e), \sigma'}$$

Например, результатом выполнения тропы

$$100 :: sN, \langle "+" sN 1 \rangle :: sN = sN$$

является число 101.

10.5. Поиски

Тропа $S'' \text{ \$iter } S' :: He R$ 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.

$$\begin{aligned} S'' \text{ \$iter } S' &\Rightarrow S'' \text{ \$iter } S' :: , \\ S'' \text{ \$iter } S' R &\Rightarrow S'' \text{ \$iter } S' :: R \\ S'' \text{ \$iter } S' :: He &\Rightarrow S'' \text{ \$iter } S' :: He , \end{aligned}$$

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 $0e$ 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.

$$\begin{aligned} S'' \text{ \$iter } S' :: He R &\Rightarrow \\ S'' :: He , \{ R; S' \text{ \$iter } S' :: He R; \} \end{aligned}$$

Например, если значения переменных eA и eB еще не определены, перестройка

$eX : \$1 eA eB,$
 $\langle \text{Writeln } eA \rangle, \langle \text{Writeln } eB \rangle \$fail$

эквивалентна поиску

$()(eX)$
 $\$iter \ \{ eB : t1 e2 = (eA t1)(e2); \}$
 $:: (eA)(eB),$
 $\langle \text{Writeln } eA \rangle, \langle \text{Writeln } eB \rangle \$fail$

10.6. Перестройки

Тропа $S : P R$ имеет следующий смысл. Следует вычислить источник S и сопоставить полученное объектное выражение Oe 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 $\lambda, m, \sigma \vdash Q \Rightarrow X, \sigma'$ означает, что если тропа Q находится на уровне m , то результатом ее вычисления для списка сред λ является X .

$$\frac{\rho, m, \sigma \vdash Q \Rightarrow Oe, \sigma'}{[\rho]^\lambda, m, \sigma \vdash Q \Rightarrow Oe, \sigma'}$$

$$\frac{\begin{array}{l} \rho, m, \sigma \vdash Q \Rightarrow \$fail(0), \sigma' \\ \lambda, m, \sigma' \vdash Q \Rightarrow X, \sigma'' \end{array}}{[\rho]^\lambda, m, \sigma \vdash Q \Rightarrow X, \sigma''}$$

$$\frac{\rho, m, \sigma \vdash Q \Rightarrow \$fail(k+1), \sigma'}{[\rho]^\lambda, m, \sigma \vdash Q \Rightarrow \$fail(k+1), \sigma'}$$

$$\frac{\rho, m, \sigma \vdash Q \Rightarrow \$error(Oe), \sigma'}{[\rho]^\lambda, m, \sigma \vdash Q \Rightarrow \$error(Oe), \sigma'}$$

$$[], m, \sigma \vdash Q \Rightarrow \text{\$fail}(0), \sigma$$

Теперь мы можем описать семантику перестройки.

$$S : P \Rightarrow S : P ,$$

$$\rho, 0, \sigma \vdash S \Rightarrow 0e, \sigma'$$

$$\rho \vdash 0e : P \Rightarrow \lambda$$

$$\lambda, m, \sigma' \vdash R \Rightarrow X, \sigma''$$

$$\rho, m, \sigma \vdash S : P R \Rightarrow X, \sigma''$$

$$\rho, 0, \sigma \vdash S \Rightarrow \text{\$fail}(k), \sigma'$$

$$\rho, m, \sigma \vdash S : P R \Rightarrow \text{\$fail}(0), \sigma'$$

$$\rho, 0, \sigma \vdash S \Rightarrow \text{\$error}(0e), \sigma'$$

$$\rho, m, \sigma \vdash S : P R \Rightarrow \text{\$error}(0e), \sigma'$$

Например, вычисление приведенной ниже тропы заканчивается неуспехом. При этом печатается цепочка литер 'CBA':

'ABC' : $\text{\$r e sX e}, \langle \text{Print sX} \rangle \text{\$fail}$

10.7. Огражденные тропы

Вычисление хвоста $\# S R$, Q всегда дает тот же результат, что и вычисление тропы Q .

$$\# S R, Q \Rightarrow Q$$

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.

$$\# S \Rightarrow \# S ,$$

$$\rho, 0, \sigma \vdash S \Rightarrow , \sigma'$$

$$\rho, m, \sigma \vdash \# S R \Rightarrow \text{\$fail}(0), \sigma'$$

$$\rho, 0, \sigma \vdash S \Rightarrow \text{\$fail}(k), \sigma'$$

$$\rho, m, \sigma' \vdash R \Rightarrow X, \sigma''$$

$$\rho, m, \sigma \vdash \# S R \Rightarrow X, \sigma''$$

$$\rho, 0, \sigma \vdash S \Rightarrow \text{\$error}(0e), \sigma'$$

$$\rho, m, \sigma \vdash \# S R \Rightarrow \text{\$error}(0e), \sigma'$$

10.9. Заборы

Хвост $\backslash? Q$ 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.

$$\rho, m+1, \sigma \vdash Q \Rightarrow 0e, \sigma'$$

$$\rho, m, \sigma \vdash \backslash? Q \Rightarrow 0e, \sigma'$$

$$\rho, m+1, \sigma \vdash Q \Rightarrow \text{\$fail}(0), \sigma'$$

$$\rho, m, \sigma \vdash \backslash? Q \Rightarrow \text{\$fail}(0), \sigma'$$

$$\rho, m+1, \sigma \vdash Q \Rightarrow \text{\$fail}(k+1), \sigma'$$

$$\rho, m, \sigma \vdash \backslash? Q \Rightarrow \text{\$fail}(k), \sigma'$$

$$\rho, m+1, \sigma \vdash Q \Rightarrow \text{\$error}(0e), \sigma'$$

$$\rho, m, \sigma \vdash \backslash? Q \Rightarrow \text{\$error}(0e), \sigma'$$

10.10. Отсечения

Хвост $\backslash! Q$ 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.

$$\frac{\rho, m, \sigma \vdash Q \Rightarrow 0e, \sigma'}{\rho, m+1, \sigma \vdash \backslash! Q \Rightarrow 0e, \sigma'}$$

$$\frac{\rho, m, \sigma \vdash Q \Rightarrow \$fail(k), \sigma'}{\rho, m+1, \sigma \vdash \backslash! Q \Rightarrow \$fail(k+1), \sigma'}$$

$$\frac{\rho, m, \sigma \vdash Q \Rightarrow \$error(0e), \sigma'}{\rho, m+1, \sigma \vdash \backslash! Q \Rightarrow \$error(0e), \sigma'}$$

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';
}
```

10.11. Тупики

Вычисление хвоста $\$fail$ всегда дает $\$fail(0)$, т.е. неуспех серьезности 0.

$$\rho, m, \sigma \vdash \$fail \Rightarrow \$fail(0), \sigma$$

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.

$$\frac{\rho, 0, \sigma \vdash Q \Rightarrow 0e, \sigma'}{\rho, m, \sigma \vdash = Q \Rightarrow 0e, \sigma'}$$

$$\frac{\rho, 0, \sigma \vdash Q \Rightarrow \$fail(k), \sigma'}{\rho, m, \sigma \vdash = Q \Rightarrow \$fail(m+1), \sigma'}$$

$$\frac{\rho, 0, \sigma \vdash Q \Rightarrow \$error(0e), \sigma'}{\rho, m, \sigma \vdash = Q \Rightarrow \$error(0e), \sigma'}$$

10.13. Аварии

Хвост $\$error Q$ позволяет породить ошибку $\$error(0e)$, где объектное выражение $0e$ является результатом вычисления тропы Q .

$$\frac{\rho, 0, \sigma \vdash Q \Rightarrow 0e, \sigma'}{\rho, m, \sigma \vdash \$error Q \Rightarrow \$error(0e), \sigma'}$$

$$\frac{\rho, 0, \sigma \vdash Q \Rightarrow \$fail(k), \sigma' \quad \text{Fname — имя функции, в которой находится конструкция.}}{\rho, m, \sigma \vdash \$error Q \Rightarrow \$error(\text{Fname "Unexpected fail"}), \sigma'}$$

$$\frac{\rho, 0, \sigma \vdash Q \Rightarrow \$error(0e), \sigma'}{\rho, m, \sigma \vdash \$error Q \Rightarrow \$error(0e), \sigma'}$$

10.14. Перехваты аварий

Хвост $\$strap Q \$with Palt$ означает, что следует попы-

гаться вычислить Q . Если результатом этого вычисления является $\$error(Oe)$, то дальше вычисляется выбор

$$Oe : Palt$$

и то, что получится, считается результатом всей конструкции.

Считается, что тропа Q находится на нулевом уровне.

$$\rho, O, \sigma \vdash Q \Rightarrow Oe, \sigma'$$

$$\rho, m, \sigma \vdash \$strap Q \$with Palt \Rightarrow Oe, \sigma'$$

$$\rho, O, \sigma \vdash Q \Rightarrow \$fail(k), \sigma'$$

$Fname$ — имя функции, в которой находится конструкция.

$$\rho, m, \sigma' \vdash Fname "Unexpected fail" : Palt \Rightarrow X, \sigma''$$

$$\rho, m, \sigma \vdash \$strap Q \$with Palt \Rightarrow X, \sigma''$$

$$\rho, O, \sigma \vdash Q \Rightarrow \$error(Oe), \sigma'$$

$$\rho, m, \sigma' \vdash Oe : Palt \Rightarrow X, \sigma''$$

$$\rho, m, \sigma \vdash \$strap Q \$with Palt \Rightarrow X, \sigma''$$

10.15. Распутья

Источник $\{Q_1; Q_2; \dots; Q_n\}$ означает, что следует вычислять тропы Q_1, Q_2, \dots, 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 Oe , то Oe считается результатом всего распутья. Если результат — $\$error(Oe)$, то $\$error(Oe)$ 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 $\{Q_1; Q_2; \dots; Q_n\}$ эквивалентно распутью

$\backslash\{Q_1; Q_2; \dots Q_n; \$error(Fname \text{ "Unexpected fail"});\}$, где $Fname$ — имя функции, в которой находится конструкция.

$$\{Q_1; Q_2; \dots Q_n;\} \Rightarrow \\ \backslash\{Q_1; Q_2; \dots Q_n; \\ \$error(Fname \text{ "Unexpected fail"}); \}$$

где $Fname$ — имя функции, в которой находится конструкция.

$$\rho, m, \sigma \vdash \{\} \Rightarrow \$fail(0), \sigma$$

$$\rho, m, \sigma \vdash Q_1 \Rightarrow 0e, \sigma'$$

$$\rho, m, \sigma \vdash \{Q_1; Q_2; \dots Q_n;\} \Rightarrow 0e, \sigma'$$

$$\rho, m, \sigma \vdash Q_1 \Rightarrow \$fail(0), \sigma'$$

$$\rho, m, \sigma' \vdash \{Q_2; \dots Q_n;\} \Rightarrow X, \sigma''$$

$$\rho, m, \sigma \vdash \{Q_1; Q_2; \dots Q_n;\} \Rightarrow X, \sigma''$$

$$\rho, m, \sigma \vdash Q_1 \Rightarrow \$fail(k+1), \sigma'$$

$$\rho, m, \sigma \vdash \{Q_1; Q_2; \dots Q_n;\} \Rightarrow \$fail(k+1), \sigma'$$

$$\rho, m, \sigma \vdash Q_1 \Rightarrow \$error(0e), \sigma'$$

$$\rho, m, \sigma \vdash \{Q_1; Q_2; \dots Q_n;\} \Rightarrow \$error(0e), \sigma'$$

10.16. Выборы

Вычисление источника $S : \{Snt_1; \dots Snt_n;\}$ всегда дает тот же результат, что и вычисление тропы $S : Ve, \{Ve : Snt_1; \dots Ve : Snt_n;\}$, при условии, что Ve — некоторая е-переменная, которая больше нигде не используется в программе.

Источник $S : \{Snt_1; \dots Snt_n;\}$ эквивалентен источнику $S : \{Snt_1; \dots Snt_n; e \$error(Fname \text{ "Unexpected fail"});\}$, где $Fname$ — имя функции, в которой находится данная конструкция.

$S : \{Snt_1; \dots Snt_n;\} \Rightarrow$
 $S : \backslash\{ Snt_1; \dots Snt_n;$
 $\quad e \text{ \$error(Fname "Unexpected fail"); } \}$
 где Fname — имя функции, в которой находится
 конструкция.

$$\rho, 0, \sigma \vdash S \Rightarrow 0e, \sigma'$$

$$\rho, m, \sigma' \vdash \backslash\{0e : Snt_1; \dots 0e : Snt_n;\} \Rightarrow X, \sigma''$$

$$\rho, m, \sigma \vdash S : \backslash\{Snt_1; \dots Snt_n;\} \Rightarrow X, \sigma''$$

$$\rho, 0, \sigma \vdash S \Rightarrow \$fail(k), \sigma'$$

$$\rho, m, \sigma \vdash S : \backslash\{Snt_1; \dots Snt_n;\} \Rightarrow \$fail(0), \sigma'$$

$$\rho, 0, \sigma \vdash S \Rightarrow \$error(0e), \sigma'$$

$$\rho, m, \sigma \vdash S : \backslash\{Snt_1; \dots Snt_n;\} \Rightarrow \$error(0e), \sigma'$$

10.17. Результатные выражения как источники

Источник вида Re означает, что следует попытаться
 вычислить результатное выражение Re, и то, что получится,
 считать результатом вычисления источника.

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

$$\rho, m, \sigma \vdash Re \Rightarrow X, \sigma'$$

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 Fname Snt; , оно
 эквивалентно определению Fname \{ Snt; }; .

Пусть определение функции Fname имеет вид

Fname Palt;

Тогда вычисление вызова этой функции, имеющего вид $\langle \text{Fname Re} \rangle$, сводится к вычислению источника $Oe : \text{Palt}$, где Oe — результат вычисления результатного выражения Re . Если при этом получается объектное выражение Oe' или ошибка $\$error(Oe')$, они и считаются результатом вычисления вызова. Если же получается неуспех $\$fail(k)$, то дальнейшие действия зависят от того, была ли функция Fname объявлена как откатная или безоткатная. Для откатной функции в качестве результата выдается $\$fail(0)$, а для безоткатной — $\$error(\text{Fname "Unexpected fail"})$.

$$\{\}, 0, \sigma \vdash Oe : \text{Palt} \Rightarrow Oe', \sigma'$$

$$\sigma \vdash \langle \text{Fname } Oe \rangle \Rightarrow Oe', \sigma'$$

$$\{\}, 0, \sigma \vdash Oe : \text{Palt} \Rightarrow \$fail(k), \sigma'$$

Функция Fname — откатная, т.е.
объявлена как $\$func? \text{Fname } F_{in} = F_{out};$.

$$\sigma \vdash \langle \text{Fname } Oe \rangle \Rightarrow \$fail(0), \sigma'$$

$$\{\}, 0, \sigma \vdash Oe : \text{Palt} \Rightarrow \$fail(k), \sigma'$$

Функция Fname — безоткатная, т.е.
объявлена как $\$func \text{Fname } F_{in} = F_{out};$.

$$\sigma \vdash \langle \text{Fname } Oe \rangle \Rightarrow \$error(\text{Fname "Unexpected fail"}), \sigma'$$

$$\{\}, 0, \sigma \vdash Oe : \text{Palt} \Rightarrow \$error(Oe'), \sigma'$$

$$\sigma \vdash \langle \text{Fname } Oe \rangle \Rightarrow \$error(Oe'), \sigma'$$

Во всех вышеприведенных правилах вывода предполагается, что функция Fname имеет определение Fname Palt.

12. Объявления

12.1. Объявления констант

```

$   ОбъявлениеКонстант =
$       "$const" [ ОбьКонст { ", " ОбьКонст } ] ";".
$   ОбьКонст = ИмяВыражения "=" КонстантноеВыражение.
$   КонстантноеВыражение =
$       { КонстантныйТерм | ИмяВыражения }.
$   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 O_e присвоено имя N , то в дальнейшем конструкция $\&N$ является изображением этого выражения. Например, следующее объявление вводит три имени объектных выражений:

```
$const LF = 10, CR = 13, "****" = A B C;
```

после чего $\&LF$ обозначает 10, $\&CR$ обозначает 13, а $\&"****"$ обозначает A B C.

Объявление константы само может ссылаться на предшествующие объявления констант, ящиков, таблиц, каналов и функций. Например, после объявления

```
$const "CR-LF-****" = &CR &LF &"****";
```

$\&"CR-LF-****"$ начинает обозначать выражение 13 10 A B C.

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 = Слово.`

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;`

12.3. Объявления функций

\$ `ОбъявлениеФункции =`

\$ `"$func" ИмяФункции`

\$ `ВходнойФормат "=" ВыходнойФормат ";" |`

\$ `"$func?" ИмяФункции`

\$ `ВходнойФормат "=" ВыходнойФормат ";"`.

\$ `ИмяФункции = Слово.`

\$ `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.

В дальнейшем форматные выражения обозначаются через *F*.

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 сигнала об ошибке.

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 сигнала о неуспехе, либо выдачей сигнала об ошибке.

Например:

```
$func Interpreter (e.Program) (e.Input) = e.Result;
$func? Attempt t.Arg = s.Result1 t.Result2 (e.Result3);
```

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 "\$tracall" означает, что следует печатать отладочную информацию для всех последующих функций.

13. Контекстные ограничения

13.1. Устранение избыточных конструкций

В этом разделе мы описываем различные контекстные синтаксические ограничения, которым должна удовлетворять любая программа не 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:

S :: He	⇒	S :: He ,
S' \$iter S''	⇒	S' \$iter S'' :: ,
S' \$iter S'' :: He	⇒	S' \$iter S'' :: He ,
S' \$iter S'' R	⇒	S' \$iter S'' :: R
S : P	⇒	S : P ,
# S	⇒	# S ,

Хвосты, опущенные в предложениях, заменены на пустые огражденные тропы следующим образом:

P	⇒	P ,
---	---	-----

"Непрозрачные" фигурные скобки "{" заменены на "прозрачные" фигурные скобки "\{" в распутьях и выборах следующим образом:

{Snt ₁ ; ... Snt _n ;} ⇒	
\{Snt ₁ ; ... Snt _n ;	
\$error(Fname "Unexpected fail");}	

$$S : \{Snt_1; \dots Snt_n;\} \Rightarrow$$

$$S : \backslash\{Snt_1; \dots Snt_n;$$

$$e \$error(Fname "Unexpected fail");\}$$

где $Fname$ — имя функции, в которой находится конструкция.

"Непрозрачные" фигурные скобки "{" заменены на прозрачные фигурные скобки "\{" в определениях функций следующим образом:

$$Fname \{Snt_1; \dots Snt_n;\} \Rightarrow$$

$$Fname \backslash\{Snt_1; \dots Snt_n;$$

$$F_{in} \$error(Fname "Unexpected fail");\}$$

где F_{in} — входной формат функции из объявления функции $Fname$ (в котором опущены индексы переменных).

Тела функций, являющиеся предложениями, заменены на соответствующие образцовые распутья:

$$Fname Snt; \quad \Rightarrow \quad Fname \backslash\{ Snt; \};$$

13.2. Ограничения налагаемые объявлениями функций

Объявления функций имеют вид

$$\$func \quad F_{in} = F_{out};$$

$$\$func? \quad F_{in} = F_{out};$$

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 $F' \geq F''$ будет озна-

чать, что формат F'' является (частным) случаем формата F' .
Отношение \geq определяется следующими правилами.

- $F \geq F$.
- Если $F1' \geq F1$ и $F2' \geq F2$, то $F1'F2' \geq F1 F2$.
- Если $F1 \geq F2$, то $(F1) \geq (F2)$.
- $e \geq F$.
- Если F не имеет вида $e e \dots e$, то $v \geq F$.
- $t \geq 0s$, для любого символа $0s$.
- $t \geq s$.
- $t \geq (F)$.
- $s \geq 0s$, для любого символа $0s$.

Если в программе имеется результирующее 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 $\langle Fname Re' \rangle$, а объявление функции $Fname$ имеет вид $\$func Fname F_{in} = F_{out}$; либо $\$func? Fname F_{in} = F_{out}$; то $\langle Fname Re' \rangle$ 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, 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 $F[Re]$ будет обозначать формат результатного выражения Re , $F[P]$ — формат образца P , а $F[He]$ — 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 $F[Re]$ 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 $Fname$ имеет входной формат F_{in} , выходной формат F_{out} , а в ее определении

$$Fname \text{ Palt}$$

образцовое распутье $Palt$ имеет вид $\{P_1 R_1; \dots P_n R_n\}$.

Тогда должны быть выполнены следующие условия.

Входные образцы функции P_1, \dots, P_n должны удовлетворять следующему ограничению: $F_{in} \supseteq F[P_j]$.

Все вызовы функции $Fname$ во всех определениях функций должны удовлетворять следующему ограничению.

Пусть вызов этой функции имеет вид $\langle Fname Re \rangle$. Тогда должно выполняться условие $F_{in} \supseteq F[Re]$.

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 распутием $Palt$, 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 $Fname$ имеет определение $Fname Palt$ 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.

$$\begin{array}{c}
 \frac{F[] \vdash S \quad F \vdash R}{F \vdash S R} \qquad \frac{F[He] \vdash S \quad F \vdash R}{F \vdash S :: He R} \\
 \\
 \frac{F[He] \vdash S'' \quad F[He] \vdash S' \quad F \vdash R}{F \vdash S'' \text{ \$iter } S' :: He R} \\
 \\
 \frac{F \vdash R}{F \vdash S : P R} \qquad \frac{F \vdash Q}{F \vdash , Q} \qquad \frac{F[] \vdash S \quad F \vdash R}{F \vdash \# S R} \\
 \\
 \frac{F \vdash Q}{F \vdash \backslash ? Q} \qquad \frac{F \vdash Q}{F \vdash \backslash ! Q} \qquad F \vdash \text{ \$fail} \\
 \\
 \frac{F \vdash Q}{F \vdash = Q} \qquad F \vdash \text{ \$error } Q \\
 \\
 \frac{F \vdash Q \quad F \vdash Palt}{F \vdash \text{ \$strap } Q \text{ \$with } Palt} \\
 \\
 \frac{F \vdash Q_j \quad \text{для всех } j=1, \dots, n}{F \vdash \backslash \{ Q ; \dots Q ; \}}
 \end{array}$$

$$\begin{array}{c}
 \begin{array}{c}
 1 \qquad n \\
 \hline
 F \vdash \text{Palt} \qquad F \geq \mathbb{F}[\text{Re}] \\
 \hline
 F \vdash S : \text{Palt} \qquad F \vdash \text{Re}
 \end{array} \\
 \\
 \hline
 F \vdash R_j \quad \text{для всех } j=1, \dots, n \\
 \hline
 F \vdash \{P_1 R_1; \dots P_n R_n\}
 \end{array}$$

13.3. Ограничения на использование ссылок на функции

Если в образцовом или результатном выражении появляется ссылка на функцию $Fname$ вида $\&Fname$, то эта функция должна быть объявлена как $\$func\ Fname\ e = e$ или $\$func? Fname\ e = e$.

13.4. Ограничения на использование переменных

Переменные, появляющиеся в определении функции, должны удовлетворять определенным ограничениям.

А именно, 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 различными.

Сформулируем эти требования более точно. Для этого введем следующие обозначения.

$\forall[X]$ обозначает множество переменных, входящих в конструкцию X , $\{\}$ — пустое множество, $\varphi' \cup \varphi''$ — 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 переменных из φ' . Например, $\{sX, sY\} \nabla \{eY, eZ\}$

$= \{sX, eY, eZ\}$.

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 $\varphi \vdash \text{Palt}$ означает, что все переменные из φ имеют разные индексы, и все переменные, значения которых могут понадобиться для вычисления образцового распутия Palt , входят в φ .

Теперь мы можем описать ограничения, налагаемые на использование переменных в определениях функций, следующим образом.

Пусть функция Fname имеет определение Fname Palt . Тогда должно выполняться $\{\} \vdash \text{Palt}$.

Отношения $\varphi \vdash Q$ и $\varphi \vdash \text{Palt}$ определяются с помощью следующих правил.

$$\begin{array}{c}
 \varphi \vdash S \\
 \varphi \vdash R \\
 \hline
 \varphi \vdash S R
 \end{array}
 \qquad
 \begin{array}{c}
 \varphi \vdash S \\
 \varphi \nabla V[\text{He}] \vdash R \\
 \hline
 \varphi \vdash S :: \text{He } R
 \end{array}$$

$$\begin{array}{c}
 \varphi \vdash S'' \\
 \varphi \nabla V[\text{He}] \vdash S' \\
 \varphi \nabla V[\text{He}] \vdash R \\
 \hline
 \varphi \vdash S'' \text{ \$iter } S' :: \text{He } R
 \end{array}
 \qquad
 \begin{array}{c}
 \varphi \vdash S \\
 \varphi \cup V[P] \vdash R \\
 \hline
 \varphi \vdash S : P R
 \end{array}$$

$$\begin{array}{c}
 \varphi \vdash Q \\
 \hline
 \varphi \vdash , Q
 \end{array}
 \qquad
 \begin{array}{c}
 \varphi \vdash S \\
 \varphi \vdash R \\
 \hline
 \varphi \vdash \# S R
 \end{array}
 \qquad
 \begin{array}{c}
 \varphi \vdash Q \\
 \hline
 \varphi \vdash \backslash ? Q
 \end{array}
 \qquad
 \begin{array}{c}
 \varphi \vdash Q \\
 \hline
 \varphi \vdash \backslash ! Q
 \end{array}$$

$$\begin{array}{c}
 \varphi \vdash \text{\$fail} \\
 \hline
 \varphi \vdash \text{\$error } Q
 \end{array}
 \qquad
 \begin{array}{c}
 \varphi \vdash Q \\
 \hline
 \varphi \vdash = Q
 \end{array}
 \qquad
 \begin{array}{c}
 \varphi \vdash Q \\
 \hline
 \varphi \vdash \text{\$error } Q
 \end{array}$$

$\varphi \vdash Q$	
$\varphi \vdash \text{Palt}$	
$\varphi \vdash \text{\$strap } Q \text{\$with Palt}$	
$\varphi \vdash Q_j$ для всех $j=1, \dots, n$	$\varphi \vdash S$
$\varphi \vdash \{Q_1; \dots Q_n\}$	$\varphi \vdash \text{Palt}$
$\varphi \vdash S : \text{Palt}$	
Все переменные из φ имеют разные индексы.	
$V[\text{Re}] \subseteq \varphi$	
$\varphi \vdash \text{Re}$	
$\varphi \cup V[P_j] \vdash R_j$ для всех $j=1, \dots, n$	
$\varphi \vdash \{P_1 R_1; \dots P_n R_n\}$	

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 " $\backslash?$ " увеличивает текущий уровень на 1, а прохождение через " $\backslash!$ " уменьшает уровень на 1. Таким образом, каждому отсечению " $\backslash!$ " должен однозначно соответствовать некоторый "парный" забор " $\backslash?$ ".

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 $k \vdash \text{Palt}$ 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 F_{name} имеет определение $F_{name} \text{ Palt}$. Тогда должно выполняться $0 \vdash \text{Palt}$.

Отношения $k \vdash Q$ и $k \vdash \text{Palt}$ определяются с помощью следующих правил.

$$\begin{array}{c}
 \frac{0 \vdash S \quad k \vdash R}{k \vdash S R} \qquad \frac{0 \vdash S \quad k \vdash R}{k \vdash S :: \text{He } R} \\
 \\
 \frac{0 \vdash S'' \quad 0 \vdash S' \quad k \vdash R}{k \vdash S'' \text{ \$iter } S' :: \text{He } R} \qquad \frac{0 \vdash S \quad k \vdash R}{k \vdash S : P R} \\
 \\
 \frac{k \vdash Q}{k \vdash , Q} \qquad \frac{0 \vdash S \quad k \vdash R}{k \vdash \# S R} \qquad \frac{k+1 \vdash Q}{k \vdash \setminus ? Q} \\
 \\
 \frac{k \vdash Q}{k+1 \vdash \setminus ! Q} \qquad k \vdash \text{\$fail} \qquad \frac{0 \vdash Q}{k \vdash = Q} \\
 \\
 \frac{0 \vdash Q}{k \vdash \text{\$error } Q} \qquad \frac{0 \vdash Q \quad k \vdash \text{Palt}}{k \vdash \text{\$strap } Q \text{ \$with } \text{Palt}} \\
 \\
 \frac{k \vdash Q_j \quad \text{для всех } j=1, \dots, n}{k \vdash \setminus \{Q_1; \dots Q_n\}} \\
 \\
 \frac{0 \vdash S \quad k \vdash \text{Palt}}{k \vdash S : \text{Palt}} \qquad k \vdash \text{Re} \\
 \\
 \frac{k \vdash R_j \quad \text{для всех } j=1, \dots, n}{k \vdash \setminus \{P_1 R_1; \dots P_n R_n\}}
 \end{array}$$

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 ММММ занимает два файла. А именно, интерфейс модуля хранится в файле ММММ.RFI, а реализация — в файле ММММ.RF.

```

$   ИнтерфейсМодуля =
$       { Объявление }.
$   Объявление =
$       ОбъявлениеКонстант | ОбъявлениеЯщиков |
$       ОбъявлениеВекторов | ОбъявлениеСтрок |
$       ОбъявлениеТаблиц | ОбъявлениеКаналов |
$       ОбъявлениеФункции.

$   РеализацияМодуля =
$       { Импорт } { ДирективаРеализации }.
$   ДирективаРеализации =
$       Объявление |
$       ДирективаОтладки |
$       ОпределениеФункции.

$   Импорт = "$use" { ИмяМодуля } ";".
$   ИмяМодуля = Слово.

```

Если внутри реализации модуля ХХХХ требуется получить доступ к именам, объявленным в интерфейсе модуля УУУУ, следует поместить внутри реализации модуля ХХХХ директиву \$use УУУУ следующим образом:

```
/* Интерфейс модуля XXXX. */
...

```

Файл XXXX.RFI

```
/* Реализация модуля XXXX. */
$use ... YYYU ... ;
/* С этого места доступны имена, */
/* объявленные в YYYU.RFI */
...

```

Файл XXXX.RF

```
/* Интерфейс модуля YYYU. */
...

```

Файл YYYU.RFI

```
/* Реализация модуля YYYU. */
...

```

Файл YYYU.RF

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

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 ММММ, в начале модуля пользователя следует поместить директиву

```
$use ММММ;
```

В результате этого в модуль пользователя будут импортированы объявления всех функций, входящих в библиотечный модуль ММММ.

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.
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.

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(Fname "Invalid argument")`, где `Fname` — имя функции.

`LENGTH` выдает длину выражения `e.Exp`, измеренную в термах. А именно, если объектное выражение `Oe` имеет вид $Ot_1 Ot_2 \dots Ot_n$, 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.Left 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>

s.Left s.Len



<LEFT s.Left s.Len e.Exp>

s.Len s.Right



<RIGHT s.Right s.Len e.Exp>

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.

4.ARITHM: целочисленная арифметика

\$func "+" s.Int1 s.Int2 = s.Int;
 \$func "-" s.Int1 s.Int2 = s.Int;
 \$func "*" s.Int1 s.Int2 = s.Int;
 \$func DIV-REM 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(Fname "Invalid argument"), где Fname — имя функции.

Если оба аргумента — числа, эти функции выдают следующие результаты.

"+" выдает сумму двух аргументов, "-" — разность пер-

вого и второго аргументов, "*" — произведение двух аргументов, DIV — результат деления нацело первого аргумента на второй, REM — остаток от деления нацело первого аргумента на второй, DIV-REM — результат деления нацело и остаток от деления нацело первого аргумента на второй, GCD — наибольший общий делитель двух аргументов.

Если результат одной из функций "+", "-" или "*" больше, чем допускается реализацией, результатом является ошибка \$error(Fname "Size limit exceeded"), где Fname — имя функции.

Например:

<"+" 3 5>	⇒	8
<"+" 3 -5>	⇒	-2
<"-" 3 -5>	⇒	8
<"*" -2 3>	⇒	-6
<DIV 5 2>	⇒	2
<REM 5 2>	⇒	1
<DIV-REM 5 2>	⇒	2 1
<DIV 6 2>	⇒	3
<REM 6 2>	⇒	0
<DIV-REM 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(Fname "Divide by zero")`, где `Fname` — имя функции. Например:

```

<DIV 5 0>       ⇒  $error(DIV "Divide by zero")
<REM 5 0>       ⇒  $error(REM "Divide by zero")
<DIV-REM 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")

```

5. BOX: работа с ящиками

```

$func BOX      e.Exp = s.Box;
$func ?        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.

`"?"` выдает содержимое ящика, на который ссылается `s.Box`.

`STORE` заносит в ящик, на который ссылается `s.Box`, новое содержимое `e.Box`.

6.CLASS: предикаты для распознавания классов символов

\$func? BOX? e.Exp = ;
\$func? CHANNEL? e.Exp = ;
\$func? CHAR? e.Exp = ;
\$func? DIGIT? e.Exp = ;
\$func? FUNC? e.Exp = ;
\$func? INT? e.Exp = ;
\$func? LETTER? e.Exp = ;
\$func? STRING? e.Exp = ;
\$func? TABLE? e.Exp = ;
\$func? VECTOR? e.Exp = ;
\$func? WORD? 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 принадлежит 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:

BOX? — ссылки на ящики.
CHANNEL? — ссылки на каналы.
CHAR? — символы-литеры.
DIGIT? — символы-литеры, которые соответствуют десятичным цифрам.
FUNC? — ссылки на функции.
INT? — целые числа.
LETTER? — символы-литеры, которые соответствуют латинским прописным и строчным буквам.
STRING? — ссылки на строки.
TABLE? — ссылки на таблицы.

VECTOR? — ссылки на векторы.
WORD? — слова.

7.COMPARE: сравнение выражений

\$func? "=" (e.Expr1)(e.Expr2) = ;
\$func? "/=" (e.Expr1)(e.Expr2) = ;
\$func? ">=" (e.Expr1)(e.Expr2) = ;
\$func? ">" (e.Expr1)(e.Expr2) = ;
\$func? "<=" (e.Expr1)(e.Expr2) = ;
\$func? "<" (e.Expr1)(e.Expr2) = ;

Эти функции сравнивают два выражения e.Expr1 и e.Expr2 и проверяют, выполнено ли соответствующее отношение между ними: "=" — равно, "/=" — не равно, ">=" — больше или равно, ">" — больше чем, "<=" — меньше или равно, "<" — меньше чем.

Если условие выполнено, результат работы — пусто, иначе — \$fail(0).

\$func COMPARE (e.Expr1)(e.Expr2) = s.Res;

COMPARE сравнивает два выражения e.Expr1 и e.Expr2. Результатом является '<', если e.Expr1 меньше чем e.Expr2, '>' если e.Expr1 больше чем e.Expr2, и '=' , если e.Expr1 равно e.Expr2.

Объектные выражения сравниваются в соответствии со следующим отношением линейного порядка <.

Для любых двух выражений Oe' и Oe'' всегда либо $Oe' < Oe''$, либо $Oe' = Oe''$, либо $Oe'' < Oe'$.

Два выражения $Oe' = Ot_1' \dots Ot_m'$ и $Oe'' = Ot_1'' \dots Ot_n''$ сравниваются *лексикографически*, т.е. составляющие их термы сравниваются попарно слева направо, пока не будет найдена пара неравных термов Ot_k' и Ot_k'' . Если при этом $Ot_k' < Ot_k''$, то считается, что $Oe' < Oe''$.

Если окажется, что Oe' короче, чем Oe'' и все термы из Oe' равны соответствующим термам из Oe'' , то считается, что $Oe' < Oe''$.

Более формально: для любых объектных выражений Oe , Oe' , Oe'' и любых объектных термов Ot , Ot' , Ot'' имеет место

Если $Oe' < Oe''$, то $Ot Oe' < Ot Oe''$.

Если $Ot' < Ot''$, то $Ot' Oe' < Ot'' Oe''$.

$\square < Ot Oe$.

где через \square обозначено пустое объектное выражение.

Для объектных термов отношение порядка определяется следующим образом.

Все символы предшествуют термам вида (Oe) , т.е. всегда

$Os < (Oe)$

Сравнение термов вида (Oe) сводится к сравнению их содержимого, т.е.

Если $Oe' < Oe''$, то $(Oe') < (Oe'')$.

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 Os' и Os'' принадлежат двум различным

типам \mathcal{J}' и \mathcal{J}'' , и при этом $\mathcal{J}' < \mathcal{J}''$, то считается, что $0s' < 0s''$.

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.

8.CONVERT: преобразования между различными типами данных

```
$func TO-LOWER e.Char = e.Char;
$func TO-UPPER e.Char = e.Char;
$func CHARS-TO-BYTES e.Char = e.Int;
$func BYTES-TO-CHARS e.Int = e.Char;
$func TO-CHARS e.Exp = e.Char;
$func TO-WORD e.Exp = s.Word;
$func? TO-INT e.Exp = s.Int;
```

TO-LOWER преобразует цепочку символов–литер в цепочку литер, в которой все прописные латинские буквы заменены на соответствующие строчные буквы.

TO-UPPER преобразует цепочку символов–литер в цепочку литер, в которой все строчные латинские буквы заменены на соответствующие прописные буквы.

CHARS-TO-BYTES преобразует цепочку символов–литер в последовательность чисел, являющихся их кодами ASCII.

Если аргумент этих функций не является цепочкой символов–литер, результатом является `$error(Fname "Invalid argument")`, где Fname — имя функции.

Например:

```

<TO-LOWER 'AbCd+'> ⇒ 'abcd+'
<TO-LOWER 25>      ⇒
                    $error(TO-LOWER "Invalid argument")
<TO-UPPER 'AbCd+'> ⇒ 'ABCD+'
<TO-UPPER 25>      ⇒
                    $error(TO-UPPER "Invalid argument")

<CHARS-TO-BYTES 'ABC'> ⇒ 65 66 67

```

BYTES-TO-CHARS преобразует цепочку целых чисел, каждое из которых лежит в диапазоне от 0 до 255, в цепочку символов-литер, имеющих соответствующие коды ASCII.

Например:

```
<BYTES-TO-CHARS 65 66 67> ⇒ 'ABC'
```

Функции TO-CHARS, TO-WORD и TO-INT 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(Fname "Argument too large for conversion").

Дальнейшая работа функций TO-CHARS, TO-WORD и TO-INT происходит следующим образом.

TO-CHARS выдает полученную цепочку литер в качестве результата.

```

<TO-CHARS "John"> ⇒ 'John'
<TO-CHARS 'John'> ⇒ 'John'

```

```

<TO-CHARS 326>      ⇒  '326'
<TO-CHARS -326>     ⇒  '-326'
<TO-CHARS (-326) "John"> ⇒  '(-326)John'

```

TO-WORD преобразует полученную цепочку литер в слово с соответствующим печатным именем.

```

<TO-WORD "John">    ⇒  "John"
<TO-WORD 'John'>   ⇒  "John"
<TO-WORD 326>       ⇒  "326"
<TO-WORD -326>      ⇒  "-326"
<TO-WORD (-326) "John"> ⇒  "(-326)John"

```

TO-INT преобразует полученную цепочку литер в целое число s.Int. Если эта цепочка литер не является правильным изображением целого числа, результатом является \$fail(0).

Например:

```

<TO-INT '326'>      ⇒  326
<TO-INT '+326'>     ⇒  326
<TO-INT "-3" '26'>  ⇒  -326
<TO-INT -32 006>    ⇒  -326
<TO-INT 'John'>     ⇒  $fail(0)

```

9.DOS: связь с операционной системой

```

$func ARG      s.Int = e.Arg;
$func GETENV  e.VarName = e.Value;
$func TIME    = e.String;
$func EXIT    s.ReturnCode = ;

```

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(Fname

"Invalid argument"), где FName — имя функции.

ARG выдает аргумент командной строки с номером s.Int. Если он отсутствует, выдается пустая строка.

GETENV извлекает из среды (в смысле MSDOS) значение переменной с именем e.VarName и выдает его.

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(0e), код завершения равен 100.

10.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? OPEN-FILE      s.Channel e.FileName s.Mode = ;
$func  CLOSE-CHANNEL s.Channel = ;
```

OPEN-FILE открывает канал `s.Channel` и связывает его с файлом с именем `e.FileName`. `s.Mode` — это слово, которое 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 удалось, результатом работы OPEN-FILE является неуспех `$fail(0)`.

CLOSE-CHANNEL закрывает канал `s.Channel`.

```
$func? EOF?          s.Channel = ;
```

EOF? проверяет, что текущая позиция у файла, с которым связан канал `s.Channel`, находится в конце файла.

```
$func? READ          = t.Term;
$func? READ-CHAR     = s.Char;
$func? READ-LINE     = 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 термина. Если не осталось ни одного термина — выдает `$fail(0)`.

READ-CHAR читает из канала `&STDIN` очередную литеру. Если не осталось ни одной литеры — выдает `$fail(0)`.

READ-LINE 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 TO-CHARS, и выводит его в канал &STDOUT.

PRINTLN делает то же, что и PRINT, но в конце добавляет литеру конца строки.

```
$func? READ!          s.Channel = t.Term;
$func? READ-CHAR!    s.Channel = s.Char;
$func? READ-LINE!    s.Channel = e.Char;
$func WRITE!         s.Channel e.Exp = ;
$func WRITELN!       s.Channel e.Exp = ;
$func PRINT!         s.Channel e.Exp = ;
$func PRINTLN!       s.Channel e.Exp = ;
```

Эти функции отличаются от соответствующих функций без восклицательного знака тем, что операции выполняются над каналом s.Channel.

II.STRING: работа со строками

```
$func STRING          e.Source = s.String;
$func STRING-INIT    s.String s.Len s.Fill = ;
$func STRING-FILL    s.String s.Fill = ;
$func STRING-LENGTH  s.String = s.Len;
$func STRING-REF     s.String s.Index = s.Char;
$func STRING-SET     s.String s.Index s.Char = ;
$func STRING-REPLACE s.String e.Source = ;
$func SUBSTRING      s.String s.Index s.Len = s.NewString;
$func SUBSTRING-FILL 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(Fname "Invalid argument"),

где `Fname` — 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 Os_0, Os_1, \dots, Os_n , мы будем говорить, что строка имеет *длину* $n+1$, а содержимое строки будем изображать в виде

$$Os_0 \ Os_1 \ \dots \ Os_n$$

Таким образом, компоненты строки Os_0, Os_1, \dots, Os_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: $Os_1 \ Os_2 \ \dots \ Os_m$, где каждый из символов Os_j является либо символом-литерой, либо символом-словом, либо ссылкой на строку. Тогда каждый из Os_j преобразуется следующим образом.

Если Os_j является символом-литерой Os , он остается без изменения.

Если Os_j является символом-словом, он заменяется на цепочку литер, которая является печатным именем этого символа.

Если Os_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.

`STRING-INIT` 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`.

`STRING-FILL` изменяет содержимое строки, на которую указывает `s.String`. При этом длина строки не меняется, а каждая компонента строки принимает значение `s.Fill`.

`STRING-LENGTH` выдает длину строки, на которую указывает

s.String.

STRING-REF извлекает из строки, на который указывает s.String, содержимое ее компоненты с номером s.Index, которое и выдается в качестве результата.

STRING-SET 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.

STRING-REPLACE 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 создаёт 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 $0c_0 0c_1 \dots 0c_n$. 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.

SUBSTRING-FILL 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(Fname "Index out of range"), где Fname — 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(Fname "Size limit exceeded"), где Fname — имя функции, а состояние строки не меняется.

12.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? IN-TABLE?     s.Tab e.Key = ;
$func DOMAIN          s.Tab = e.Domain ;
$func TABLE-COPY    s.Tab = s.TabCopy;
$func REPLACE-TABLE 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)`.

IN-TABLE? 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 $\{0e_1, 0e_2, \dots, 0e_n\}$, `e.Domain` имеет вид

$$(0e_1)(0e_2) \dots (0e_n)$$

где порядок расположения ключей описанием языка не определен.

TABLE-COPY создает новую таблицу, копирует в него содержимое таблицы, на которую ссылается `s.Tab`, и выдает в качестве результата ссылку на новую таблицу.

REPLACE-TABLE уничтожает прежнее содержимое таблицы, на которую ссылается `s.TargetTable`, и заменяет его на копию содержимого таблицы, на которую ссылается `s.Source`

13. VECTOR: работа с векторами

```

$func VECTOR          e.Source = s.Vector;
$func VECTOR-TO-EXP   s.Vector = e.Exp;
$func VECTOR-INIT     s.Vector s.Len e.Fill = ;
$func VECTOR-FILL     s.Vector e.Fill = ;
$func VECTOR-LENGTH   s.Vector = s.Len;
$func VECTOR-REF      s.Vector s.Index = e.Exp;
$func VECTOR-SET      s.Vector s.Index e.Exp = ;
$func VECTOR-REPLACE  s.Vector e.Source = ;
$func SUBVECTOR       s.Vector s.Index s.Len = s.NewVector;
$func SUBVECTOR-FILL  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 $(0e)$.

Если хотя бы одно из этих условий не выполнено, результатом функций является `$error(Fname "Invalid argument")`, где `Fname` — 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 $0e_0, 0e_1, \dots, 0e_n$, мы будем говорить, что вектор имеет *длину* $n+1$, а содержимое вектора будем изображать в виде

$$(0e_0)(0e_1) \dots (0e_n)$$

Таким образом, компоненты вектора $0e_0, 0e_1, \dots, 0e_n$ 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: $0t_1 0t_2 \dots 0t_m$, где каждый из объектных термов $0t_j$ либо имеет вид $(0e)$, либо является ссылкой на вектор. Тогда каждый из $0t_j$ преобразуется следующим образом.

Если $0t_j$ имеет вид $(0e)$, он остается без изменения.

Если $0t_j$ 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.

`VECTOR-TO-EXP` выдает в виде объектного выражения содержимое вектора, на который указывает `s.Vector`.

`VECTOR-INIT` 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`.

`VECTOR-FILL` изменяет содержимое вектора, на который указывает `s.Vector`. При этом длина вектора не меняется, а каждая компонента вектора принимает значение `e.Fill`.

`VECTOR-LENGTH` выдает длину вектора, на который указывает `s.Vector`.

`VECTOR-REF` извлекает из вектора, на который указывает `s.Vector`, содержимое его компоненты с номером `s.Index`, которое и выдается в качестве результата.

`VECTOR-SET` 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`.

`VECTOR-REPLACE` 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 $(0e_0)(0e_1) \dots (0e_n)$.

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.

`SUBVECTOR-FILL` 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.Ehr`.

Если размер исходного вектора недостаточен для выполнения одной из вышеуказанных операций, в качестве результата выдается `$error(Fname "Index out of range")`, где `Fname` — 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(Fname "Size limit exceeded")`, где `Fname` — имя функции, а состояние вектора не меняется.

ЛИТЕРАТУРА

[АБР 88]

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.

[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 79]

А.Ахо, Дж.Хопкрофт, Дж.Ульман. Построение и анализ вычислительных алгоритмов: Пер. с англ. — М.:Мир, 1979. — 536 с.

[БзР 77]

Базисный Рефал и его реализация на вычислительных машинах. — М.:ЦНИПИАСС, 1977 — 258 с.

[БьД 82]

D.Bjorner, C.B.Jones. Formal Specification and Software Development. Prentice-Hall International, London, 1982.

[Вир 77]

Н.Вирт. Систематическое программирование. Введение: Пер. с англ. — М.:Мир, 1977. — 184 с.

[Вир 85]

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 87]

В.Л.Кистлеров. Принципы построения языка алгебраических вычислений FLAC. — Препринт, М.:Институт проблем управления, 1987. — 39 с.

[КлР 86]

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 № 200. — 38 с.

[КлР 87]

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 72]

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 87а]

С.А.Романенко. Реализация Рефала-2. — М.:ИПМ им.М.В.Келдыша АН СССР, 1977. — 191 с.

[Ром 87б]

С.А.Романенко. Рефал-4 — расширение Рефала-2, обеспечивающее выразимость результатов прогонки. — М.:ИПМ им.М.В.Келдыша АН СССР, 1987, препринт № 147. — 27 с.

[Ром 87в]

С.А.Романенко. Прогонка для программ на Рефале-4. — М.:ИПМ им.М.В.Келдыша АН СССР, 1987, препринт N 211. — 19 с.

[Ром 87г]

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 88]

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 88]

S. A. Romanenko. A Compiler Generator Produced by a Self-Applicable Specializer Can Have a Surprisingly Natural and Understandable Structure. In D. Bjerne, A. P. Ershov and N. D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 445-463, North-Holland, 1988.

[Тур 66]

В.Ф.Турчин. Метаязык для формального описания алгоритмических языков. — В сб.: Цифровая вычислительная техника и программирование, М.:Сов. Радио, 1966, с.116-124.

[Тур 71]

В.Ф.Турчин. Программирование на языке Рефал. — М.:ИПМ АН СССР, 1971, препринты N 41, N 43, N 44, N 48, N 49.

[Typ 86]

V.F.Turchin. The concept of a supercompiler. ACM Transactions on Programming Languages and Systems, Vol.8, No.3, July 1986, pp.292-325.

[Typ 89]

V.F.Turchin. Refal-5, *Programming Guide and Reference Manual*. New England Publishing Co., Holyoke, 1989.

[Yop 80]

D.H.D.Warren. Logic Programming and Compiler Writing. Software — Practice and Experience, Vol.10, 97-125 (1980).

[Хен 83]

П.Хендерсон. Функциональное программирование. Применение и реализация: Пер. с англ. — М.:Мир, 1983. — 349 с.

[Шмд 86]

D.A.Schmidt. *Denotational Semantics*. Allyn and Bacon, Boston, 1986.

АЛФАВИТНЫЙ УКАЗАТЕЛЬ ФУНКЦИЙ

\$func	"*"	s.Int1 s.Int2 = s.Int;	ARITHM
\$func	"+"	s.Int1 s.Int2 = s.Int;	ARITHM
\$func	"-"	s.Int1 s.Int2 = s.Int;	ARITHM
\$func?	"!="	(e.Exp1)(e.Exp2) = ;	COMPARE
\$func?	"<"	(e.Exp1)(e.Exp2) = ;	COMPARE
\$func?	"<="	(e.Exp1)(e.Exp2) = ;	COMPARE
\$func?	"="	(e.Exp1)(e.Exp2) = ;	COMPARE
\$func?	">"	(e.Exp1)(e.Exp2) = ;	COMPARE
\$func?	">="	(e.Exp1)(e.Exp2) = ;	COMPARE
\$func	?	s.Box = e.Exp;	BOX
\$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	BOX	e.Exp = s.Box;	BOX
\$func?	BOX?	e.Exp = ;	CLASS
\$func	BYTES-TO-CHARS	e.Char = e.Int;	CONVERT
\$func	CHANNEL	= s.Channel;	STDIO
\$func?	CHANNEL?	e.Exp = ;	CLASS
\$func?	CHAR?	e.Exp = ;	CLASS
\$func	CHARS-TO-BYTES	e.Int = e.Char;	CONVERT
\$func	CLOSE-CHANNEL	s.Channel = ;	STDIO
\$func	COMPARE	(e.Exp1)(e.Exp2) = s.Res;	COMPARE
\$func?	DIGIT?	e.Exp = ;	CLASS
\$func	DIV	s.Int1 s.Int2 = s.Quo;	ARITHM
\$func	DIV-REM	s.Int1 s.Int2 = s.Quo s.Rem;	ARITHM
\$func?	DOMAIN	s.Tab = e.KeyList;	TABLE
\$func?	EOF?	s.Channel = ;	STDIO
\$func	EXIT	s.ReturnCode = ;	DOS
\$func?	FUNC?	e.Exp = ;	CLASS
\$func	GCD	s.Int1 s.Int2 = s.Gcd;	ARITHM

\$func	GETENV	e.VarName = e.Value;	DOS
\$func?	IN-TABLE?	s.Tab e.Key = ;	TABLE
\$func?	INT?	e.Exp = ;	CLASS
\$func?	L	s.Left e.Exp = t.SubTerm;	ACCESS
\$func?	LEFT	s.Left s.Len e.Exp = e.SubExp;	ACCESS
\$func	LENGTH	e.Exp = s.ExpLen;	ACCESS
\$func?	LETTER?	e.Exp = ;	CLASS
\$func?	LOOKUP	s.Tab e.Key = e.Val;	TABLE
\$func?	MIDDLE	s.Left s.Right e.Exp = e.SubExp;	ACCESS
\$func?	OPEN-FILE	s.Channel e.FileName s.Mode = ;	STDIO
\$func	PRINT	e.Exp = ;	STDIO
\$func	PRINT!	s.Channel e.Exp = ;	STDIO
\$func	PRINTLN	e.Exp = ;	STDIO
\$func	PRINTLN!	s.Channel e.Exp = ;	STDIO
\$func?	R	s.Right e.Exp = t.SubTerm;	ACCESS
\$func?	READ	= t.Term;	STDIO
\$func?	READ!	s.Channel = t.Term;	STDIO
\$func?	READ-CHAR	= s.Char;	STDIO
\$func?	READ-CHAR!	s.Channel = s.Char;	STDIO
\$func?	READ-LINE	= e.Char;	STDIO
\$func?	READ-LINE!	s.Channel = e.Char;	STDIO
\$func	REM	s.Int1 s.Int2 = s.Rem;	ARITHM
\$func	REPLACE-TABLE	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	STRING-FILL	s.String s.Fill = ;	STRING
\$func	STRING-INIT	s.String s.Len s.Fill = ;	STRING
\$func	STRING-LENGTH	s.String = s.Len;	STRING
\$func	STRING-REF	s.String s.Index = s.Char;	STRING
\$func	STRING-REPLACE	s.String e.Source = ;	STRING
\$func	STRING-SET	s.String s.Index s.Char = ;	STRING
\$func?	STRING?	e.Exp = ;	CLASS
\$func	SUBSTRING		STRING
		s.String s.Ind s.Len = s.NewString;	
\$func	SUBSTRING-FILL		STRING
		s.String s.Index s.Len s.Fill = ;	

\$func	SUBVECTOR		VECTOR
		s.Vector s.Index s.Len = s.NewVector;	
\$func	SUBVECTOR-FILL		VECTOR
		s.Vector s.Index s.Len e.Fill =;	
\$func	TABLE	= s.Tab;	TABLE
\$func	TABLE-COPY	s.Tab = s.TabCopy ;	TABLE
\$func?	TABLE?	e.Exp = ;	CLASS
\$func	TIME	= e.String;	DOS
\$func	TO-CHARS	e.Exp = e.Char;	CONVERT
\$func?	TO-INT	e.Char = s.Int;	CONVERT
\$func	TO-LOWER	e.Char = e.Char;	CONVERT
\$func	TO-UPPER	e.Char = e.Char;	CONVERT
\$func	TO-WORD	e.Char = s.Word;	CONVERT
\$func	UNBIND	s.Tab e.Key = ;	TABLE
\$func	VECTOR	e.Source = s.Vector;	VECTOR
\$func	VECTOR-FILL	s.Vector e.Fill = ;	VECTOR
\$func	VECTOR-INIT	s.Vector s.Len e.Fill = ;	VECTOR
\$func	VECTOR-LENGTH	s.Vector = s.Len;	VECTOR
\$func	VECTOR-REF	s.Vector s.Index = e.Exp;	VECTOR
\$func	VECTOR-REPLACE	s.Vector e.Source = ;	VECTOR
\$func	VECTOR-SET	s.Vector s.Index e.Exp = ;	VECTOR
\$func	VECTOR-TO-EXP	s.Vector = e.Exp;	VECTOR
\$func?	VECTOR?	e.Exp = ;	CLASS
\$func?	WORD?	e.Exp = ;	CLASS
\$func	WRITE	e.Exp = ;	STDIO
\$func	WRITE!	s.Channel e.Exp = ;	STDIO
\$func	WRITELN	e.Exp = ;	STDIO
\$func	WRITELN!	s.Channel e.Exp = ;	STDIO

МАЛОЕ ПРЕДПРИЯТИЕ П Е Р С Е Й

для тех, кто работает хотя бы в одной из областей:

- аналитические (символьные) вычисления в математике и физике;
- построение трансляторов языков программирования;
- искусственный интеллект (базы знаний, языки представления знаний, логический вывод, синтез программ);
- обработка (преобразование) текстов;
- человеко-машинный интерфейс (анализ и понимание текстов на естественном языке, синтез текстов)

предлагает

СИСТЕМУ ПРОГРАММИРОВАНИЯ Р Е Ф А Л – 6

Рефал (алгоритмический язык рекурсивных функций) – это функциональный язык программирования продукционного типа. Его отличает концептуальная простота, сочетающаяся с богатством выразительных возможностей программирования.

Язык Рефал (его автор – бывший советский физик и математик В.Ф.Турчин) существует уже более 20 лет, но широта его использования до недавнего времени сдерживалась низкой эффективностью его реализаций: все они, до появления систем Рефал Плюс и Рефал-6, были основаны не на компиляции в прямой машинный код, а на интерпретации некоторого промежуточного языка.

Рефал-6 – это первая и пока единственная в мире система программирования на языке Рефал, включающая возможности глобальной оптимизации (суперкомпиляции) рефал-программы с отображением ее в язык программирования С (т.е., в конечном счете, в непосредственно исполняемый машинный код).

Использование суперкомпиляции и отображения в С увеличивает эффективность Ваших программ от 5 до 15 раз!

Входной язык системы практически совпадает с языком Рефал-5, разработанным В.Турчиным во 2-й половине 80-х годов. Он также включает (или включит в недалеком будущем) многое из расширений, вошедших в язык Рефал Плюс (предложенный С.А.Романенко).

Система Рефал-6 допускает динамическую загрузку и выгрузку программных модулей, работающих в режиме интерпретации, что также отличает ее от всех других реализаций Рефала.

Система Рефал-6 может быть установлена на любой ЭВМ (при наличии компилятора стандартного языка С) с памятью не менее 500 Кбайт.

152140, г.Переславль-Залесский.
ул.Советская,д.2.
МП "ПЕРСЕЙ"

Телефоны:
(08535) 2-07-15 Сергей Абрамов
(095) 387-71-44 Аркадий Климов