# Встраиваемый язык обработки текстов *Рефал-0* и разработка его транслятора на *Компонентном Паскале*

Ермаков И.Е.

Февраль-июнь-сентябрь 2008 г.

## Оглавление

# 1 Введение

Предложенный нами язык *Рефал-0* представляет собой компактное подмножество языка *Рефал-5* [1] и предназначен для встраивания в императивные языки в целях использования для обработки текстов. Подмножество средств, включенных в *Рефал-0*, выбрано таким образом, чтобы сделать возможной лёгкую трансляцию в эффективный код на распространённых императивных языках, не требующий библиотек поддержки выполнения и динамического управления памятью.

Транслятор с языка *Рефал-0* реализован нами на *Компонентном Паскале* - языке Оберон-семейства. Он поддерживает трансляцию в код на любом императивном языке (при написании соответствующего оконечного модуля).

## 1.1 Исходные соображения и постановка задачи

Мотивом для настоящей разработки стали известные трудности, связанные с написанием алгоритмов обработки текстовых данных на императивных языках программирования. В общем, можно выделить две категории алгоритмов такого рода. К первой категории относятся *простые преобразования строк*, которые часто встречаются в самых разных программах (простейшим примером может служить преобразование файловых путей между форматами, принятыми на разных платформах). Вторая категория задач – *организация синтаксического разбора* текстов формальной структуры.

Существуют стандартные средства для решения названных задач. Простая обработка строк может основываться на функциях специальной библиотеки. Для построения синтаксических анализаторов могут применяться инструментальные пакеты, автоматически строящие анализаторы по формально описанным грамматикам. Однако и одно, и другое решение не являются универсально применимыми. Если алгоритм преобразования строк интенсивно используется и является «узким местом» в работе программы, то использование строковых библиотек становится неприемлемым (речь идёт не о «накладных расходах», а об алгоритмической неэффективности обработки строк типовыми операциями этих библиотек). Что касается несложных задач разбора, которые часто встречаются в самых разных разработках, то применение к ним специальных инструментальных пакетов неоправданно из-за громоздкости последних.

Таким образом, существуют и достаточно распространены случаи, в которых названные общеизвестные средства плохо применимы. Это особенно характерно для задач системного программирования, где может быть недопустимой зависимость от какой-либо прикладной библиотеки либо громоздкого инструментального пакета и, кроме того, требуется максимальное быстродействие. Однако даже в прикладном программировании использование исключительно библиотечных средств часто даёт вполне ощутимое замедление программ: потери, возникающие на всей «площади» исходного кода, складываются в интеграле в десятки и

даже сотни процентов (это особенно заметно по многим приложениям, написанным под платформу *.NET*, которая известна богатством стандартных библиотек).

Итак, нередки случаи, когда разработчик вынужден создавать алгоритмы обработки строк вручную на императивном языке программирования. Такие алгоритмы, как правило, нетривиальны, то есть, для уверенного их составления требуется какое-либо формальное обоснование. В то же время классический формальный вывод на основе преобразования предикатов, «по Дейкстре», здесь, как показывает практика, труден для применения (фигурирует большое количество условий и элементов).

Для ручного написания синтаксических анализаторов имеется общеизвестный метод, который заключается в применении формализма конечных автоматов для написания распознавателя лексем и в последующем разборе методом рекурсивного спуска, который строится на основе КС-грамматик подходящего вида. В целом, такой подход даёт анализаторы с простой и прозрачной структурой. Однако на уровне отдельных операторов исходный текст является достаточно громоздким и трудным для модификации (а это особенно важно при одновременной разработке языка и его транслятора, когда в синтаксис могут вноситься постоянные уточнения). Это делает разработку синтаксического анализатора делом, требующим большого внимания и долгой отладки.

Отметим, что в настоящее время имеется тенденция считать проблемы, подобные рассматриваемой, имманентными для императивного программирования, и решать их либо путём полной миграции на функциональные языки, либо включением в императивный язык средств, заимствованных из них (чаще всего так называемого «синтаксического сахара», примером могут служить *C#* и *Python*). Мы не можем согласиться с таким подходом и далее продемонстрируем на примере разработки транслятора, что современные императивные языки *Оберон*-семейства за счёт мощной типизации с динамической поддержкой и автоматического управления памятью дают новые степени свободы для построения сложных алгоритмов, позволяя решать ряд задач не менее красиво и выразительно, чем с применением функциональных языков.

На самом деле, рассматриваемая проблема имеет чёткие границы, и описать её можно следующим образом: требуется **метод лёгкого составления алгоритмов обработки неструктурированных потоков данных, применимый как для непосредственных трансформаций этих данных, так и для формального разбора с преобразованием в структурированный вид (например, в синтаксическое дерево)**.

## 1.2 Подход к решению задачи: встраиваемый язык *Рефал-0*

Рассмотрим требования, которым должно удовлетворять решение. Задачи первой категории (непосредственная трансформация текста) требуют алгоритмически полного языка (т.е. недостаточно описательного формализма, например, основанного на грамматиках). Задачи формального разбора требуют преимущественно наличия мощных средств анализа входного потока и возможности лёгкого сопряжения с императивным

языком для передачи результата разбора. **Очевидным вариантом кажется использование для обработки текстов специального языка программирования.**

Заметим, что для рассматриваемых задач имеется решение, пришедшее из *Unix*-экосистемы: специализированные скриптовые языки, наиболее известным из которых является *Perl*. Недостатками этих средств является то, что скриптовые языки, как правило, плохо и хаотично спроектированы, развивались бессистемно, являются громоздкими. Кроме того, для использования даже во встраиваемом режиме они требуют наличия интерпретирующей библиотеки.

Однако мы поставили своей целью исключить необходимость ручного написания алгоритмов обработки текста, в том числе в тех случаях, когда это обычно делается из-за требований к быстродействию и ограничений на использование сторонних библиотек. **Поэтому обязательным требованием к языку обработки текстов поставим возможность лёгкой и прозрачной трансляции в эффективный код на распространённых императивных языках, не требующий динамической поддержки (нежелательно даже динамическое управление памятью).**

Обратимся к так называемым *марковским языкам программирования*, близким к абстрактной модели вычислений «нормальный алгорифм Маркова». Основными средствами марковских языков являются: *сопоставление с образцом*; замена в соответствии со *схемой подстановок*; рекурсивное применение функций (что роднит марковские языки с функциональными). Названные примитивы кажутся подходящими для наших задач, при этом из обоих категорий. При написании анализаторов сопоставление с образцом применимо для выделения лексем, а непосредственно синтаксический разбор может выполняться методом рекурсивного спуска.

Мы взяли за основу язык *Рефал* В.Ф. Турчина [1], в версии *Рефал-5* [2], которая является классическим авторским вариантом (умеренным и строгим). Язык был значительно урезан нами до простого подмножества, которое названо *Рефал-0* и предназначено для использования в качестве встраиваемого языка обработки текстов, транслируемого в императивный код. Кроме того, в подмножестве выдержана совместимость с версией *Рефал-Плюс* [3, 4], которая используется в настоящее время в отечественных суперкомпьютерных системах (проект СКИФ).

Сокращение языка заключается, в основном, в следующем: оставлен единственный вид символов – литеры (символы-числа и символы-имена не включены); образцы могут содержать не более двух *e*-переменных, порядок которых в части подстановки должен быть сохранён; отсутствуют структурированные (древовидные) объектные выражения. Различные виды символов становятся ненужными, исходя из назначения встраиваемого языка. Ограничение на вид образцов делает возможным безоткатное сопоставление, ограничение на вид подстановок облегчает построение императивного кода и делает возможным распределение памяти на этапе трансляции. Древовидные выражения и их сопоставление с образцом представляют собой наиболее мощную часть базового *Рефала*, но для нашей задачи они не нужны. Однако, как будет показано далее, при

реализации транслятора на *Компонентном Паскале* нами использованы Рефал-подобные универсальные выражения и их преобразования, описанные средствами Оберон-языка.

Как показала практика использования *Рефала-0* (в том числе при написании его собственного транслятора), в задачах синтаксического разбора он эффективно применим на переднем крае анализатора для лексического разбора и анализа рекурсивным спуском; результат анализа передаётся окружающей императивной программе через вызовы её процедур. Несложные трансформации строк могут быть описаны полностью на встроенном языке, сложные подстановки могут требовать описания базовых операций на окружающем языке и обращения к ним из *Рефала-0*. Это не является проблематичным, т.к. построение таких операций на императивном языке тривиально, а общий алгоритм трансформации по-прежнему скрывается под Рефал-описанием. Автоматически генерируемый код по эффективности аналогичен тому, что может быть написано вручную. В особых случаях максимальная эффективность может быть достигнута уже упомянутым описанием базовых операций в виде процедур окружающей программы.

## 1.3 Трансляция в код на императивном языке

После формирования подмножества средств, включаемых в *Рефал-0*, оказалось достаточно очевидным, каким образом Рефал-программа будет отображаться на конструкции императивного языка. Однако для написания транслятора было необходимо формализовать это понимание. Для этого **было решено ввести промежуточное представление – код для простой абстрактной машины обработки строк**. Основным требованием к такой машине является лёгкость отображения программ на *Рефале-0* в её код и лёгкость последующей генерации императивного кода из промежуточного. Такая машина была разработана методом постепенного уточнения, на каждом витке которого проводилась пробная ручная трансляция исходного текста на *Рефале-0* (в качестве которого был использован его собственный синтаксический анализатор) в текущий вариант промежуточного кода и далее – в императивный код на *Компонентном Паскале*. Главной целью являлось достижение максимальной простоты промежуточного представления и алгоритмов трансляции.

Важным следствием применения промежуточного кода абстрактной строковой машины является простота организации трансляции на любой из императивных языков. Для этого необходимо написать соответствующий генератор из промежуточного представления, что является шаблонной задачей.

## 1.4 Написание транслятора на *Компонентном Паскале* с частичной раскруткой

Написание транслятора для такого небольшого языка, как *Рефал-0*, не является сложной задачей. Однако, приступая к его созданию, мы

руководствовались несколькими посылками, которые привели к нетипичному решению и оригинальному подходу к разработке трансляторов. Во-первых, было решено применить *Рефал-0* на переднем крае его собственной реализации – т.е. использовать метод частичной раскрутки. Во-вторых, присутствовало понимание того, что классические архитектуры трансляторов не ориентированы на использование преимуществ современного языка, каким является *Компонентный Паскаль* (в частности, его динамических средств типизации и автоматического управления памятью), более того, эти архитектуры иногда содержат очевидные рудименты. Поэтому хотелось выполнить создание транслятора продуманно и качественно, сформировав эффективный метод и набор средств для дальнейших подобных разработок на *Компонентном Паскале* в среде *BlackBox Component Builder*.

Отметим важные принципы, которые даёт нам знакомство с Оберон-системами, воплотившими в себе опыт научной школы Никлауса Вирта [5-11]. Качество программного обеспечения является следствием простоты, которая достигается тщательным дизайном, основанным на опыте решения практических задач. При этом необходима шлифовка каждой мелкой детали, поскольку непродуманные мелочи могут давать неожиданные сквозные эффекты во всей системе. Применительно к разработке трансляторов: качественный дизайн языка, всех его конструкций и средств является основой для построения качественного и простого инструментария, что даёт долговременные, интегральные преимущества тем разработчикам, которые его будут использовать.

Ещё один ценный принцип, наиболее последовательно представленный в разработке инструментария языков Паскаль-семейства (*Паскаль*, *Модула-2*, *Оберон*...) - использование раскрутки, при этом в самом широком смысле этого термина. Речь идёт не только об использовании самого языка для разработки его компилятора, но и о целых программно-аппаратных системах, создаваемых методом постепенного, совместного развития (например, параллельная разработка коллективом Н. Вирта ПК *Lilith*, ОС *Medos* и языка *Модула-2*, а затем ПК *Ceres*, ОС *Oberon* и языка *Оберон* и родственная новосибирская разработка ПК *Кронос* и ОС *Эксельсиор* на базе Модулы-2/Оберона). Для такого интегрированного подхода характерно самоприменение, прямое и косвенное, различных частей разрабатываемых систем, когда каждый виток работ как бы индуцирует дальнейшее развитие. Это даёт высокое качество технических решений и отсекает избыточность.

Следствием разработки транслятора *Рефала-0* явилось создание небольшого комплекта средств, которые дают основу для инструментария разработки компиляторов в среде *BlackBox*. Будучи простыми сами по себе, эти средства, объединённые единым подходом и возможностями раскрутки, кажутся нам перспективными и имеющими значение для дальнейшего развития Оберон-инструментария.

# 2 Описание языка *Рефал-0*

## 2.1 Формальное описание синтаксиса

Приведём описание синтаксиса *Рефал-0* в расширенной нотации Бэкуса-Наура (*РБНФ*). Вертикальная черта | разделяет альтернативы. Квадратные скобки [ и ] обозначают необязательность заключённого в них выражения, фигурные скобки { и } - возможное повторение 0 и более раз, круглые скобки ( и ) используются для группировки. Каждое РБНФ-правило завершается точкой.

**Программа** = {Функция [ *";"* ]}.
**Функция** = ИмяФункции *"{"* Предложение { *";"* Предложение } [ *";"* ] *"}"*.
**Предложение** = Образец {*","* Условие} *"="* Выражение.
**Образец** = "" **|** Шаблон **|** [Шаблон] e-Переменная [Шаблон] **|**
[Шаблон] e-Переменная Шаблон e-Переменная [Шаблон].
**Шаблон** = (Строка **|** s-Переменная) {Строка **|** s-Переменная}.
**Условие** = *"<"*ИмяФункции (s-Переменная **|** e-Переменная) *">" ":"* (*"'T'"* **|** *"'F'"*).
**Выражение** = {Строка **|** s-Переменная **|** e-Переменная **|** ВызовФункции}.
**ВызовФункции** = *"<"*ИмяФункции Выражение *">"*.

**ИмяФункции** = латинСимвол{латинСимвол**|**цифра**|***"_"***|***"-"*}.
**s-Переменная** = *"s"*Идентификатор.
**e-Переменная** = *"e"*Идентификатор.
**Идентификатор** = (латинСимвол **|** цифра) {латинСимвол **|** цифра}.
**Строка** = *"'"* {обычныйСимвол **|** *"\"*особыйСимвол **|** знак **|** кодСимвола **|** переводСтроки | *"\"*переводСтроки} *"'"*.

**обычныйСимвол** = НЕ особыйСимвол.
**особыйСимвол** = *"\"* **|** *"""* **|** *"'"*.
**знак** = *"\n"* **|***"\t"*.
**кодСимвола** = *"\"*цифра{цифра}.
**латинСимвол** = *"A"* .. *"Z"*.
**цифра** = *"0"* | *"1"* | *"2"* | *"3"* | *"4"* | *"5"* | *"6"* | *"7"* | *"8"* | *"9"*.

**Примечания:**

1) Между последовательностями символов, соответствующими термам, разделённым в РБНФ-выражении пробелом, может находиться произвольное число пробелов, символов табуляции или переводов строки.
2) При отсутствии пробела между термами в РБНФ-выражении цепочки символов, соответствующие этим термам, должны следовать непосредственно друг за другом (в нетерминалах *Условие* и *ВызовФункции* имя функции следует непосредственно за открывающей угловой скобкой; в нетерминалах *s-Переменная* и *e-Переменная* идентификатор следует непосредственнно за префиксом и т.д.).
3) Строки в программах на *Рефале-0* ограничиваются одинарными кавычками. Чтобы поместить внутри строки одинарную кавычку, нужно использовать сочетание \'. Непосредственно в строке недопустимы символы " и \. Для того, чтобы их поместить в строку, нужно использовать соответственно сочетания \" и \\. Для того, чтобы поместить в строку символы перевода строки и табуляции, нужно использовать сочетания \n и \t. Кроме того, между открывающей и закрывающей кавычкой допустим непосредственный перевод строки - он подставляется в строку. Чтобы перевод строки был проигнорирован, следует поставить перед ним символ \.
4) **Комментарии** многострочные – обрамляются парами символов /* и */. Вложенные комментарии не допускаются.

## 2.2 Семантика языка

Выполнение программы на *Рефале-0* представляет собой применение функции к конечной последовательности символов (которую будем называть *текстом*). Функция представляет собой упорядоченный набор предложений. Каждое предложение состоит из образца и выражения, описывающего подстановку. При выполнении функции выполняется последовательное сопоставление входного текста с образцами предложений, до тех пор, пока не будет получено соответствие. Если соответствие найдено, то выполняется замена входного текста в соответствии с подстановочным выражением предложения. Если соответствие не обнаружено ни с одним образцом предложений функции, это считается ошибочной ситуацией и выполнение Рефал-0-программы прекращается (или вызывается внешняя процедура, выполняющая восстановление разбора, после чего Рефал-функция нормально завершается – см. п. 4.4).

**Пример 2.2.1**

```
 Expect {
        sC sC e1      =     e1;
        sC ' ' e1     =     <Expect sC e1>
 }
```

Функция *Expect* принимает текст, в голове которого находится символ-параметр, и проверяет (требует, в смысле безусловного утверждения), что следующий символ идентичен символу-параметру, либо отделён от него произвольным количеством пробелов. В противном случае порождается ошибочная ситуация. Первое предложение выполняется в случае, если два первых символа текста равны. Тогда оба символа удаляются, результатом применения функции *Expect* становится оставшаяся часть текста. Второе предложение выполняется в случае, если за сиволом-параметром следует пробел. Тогда пробел удаляется, а функция *Expect* рекурсивно применяется к изменённому тексту (рекурсивные применения мы выделяем жирным шрифтом).

**Образцы** строятся из двух компонент – *строк символов* и *свободных переменных*. Строка в образце требует, чтобы в соответствующем месте текста находились идентичные символы. Свободные переменные бывают двух видов: *s*-переменные и *e*-переменные; *s*-переменной в тексте должен быть сопоставлен в точности один символ; *e*-переменной может быть сопоставлена произвольная последовательность символов (в том числе пустая). Каждая свободная переменная в образце имеет *идентификатор*. Идентификаторы *s*- и *e*-переменных должны быть различными. Идентификаторы *s*-переменных могут совпадать – в таком случае *s*-переменным с одинаковыми идентификаторами должны соответствовать одинаковые символы в тексте. Как видно из описания синтаксиса языка, в образце не может быть больше двух e-переменных, которые обязательно должны быть разделены символами или s-переменными (последовательность символов и s-переменных в РБНФ-грамматике названа *шаблоном*).

Таким образом, можно выделить следующие виды образцов:

пустой образец*:* может быть записан как пустая строка или вообще опущен перед знаком "=" в предложении – он может быть сопоставлен только пустому тексту;

свободный образец: *e-переменная* – может быть сопоставлен

произвольному тексту (предложение с таким видом образца гарантировано будет применено; все последующие предложения функции никогда не будут применены – не имеют смысла);

образец фиксированной длины: *шаблон* – может быть сопоставлен тексту соответствующей длины, при условии совпадения с символами и удовлетворения равенству одноимённых *s*-переменных (пример: *s1 '+' s2 '=' s1 '*' s2*);

образец с концевыми сравнениями: *шаблон e-переменная* или *e-переменная шаблон* или *шаблон e-переменная шаблон* - может быть сопоставлен любому тексту, начало и конец которого соответствуют шаблонам (пример: *s1' ' e2' 's1*);

образец с плавающей частью:

*[шаблон] e-переменная1 шаблон e-переменная2 [шаблон]*

- такой образец сопоставляется тексту при удовлетворении концевых сравнений и в том случае, если в середине текста (между фрагментами, уже сопоставленными концевым шаблонам) имеется последовательность символов, удовлетворяющая среднему шаблону. Поиск плавающей части выполняется слева направо (т.е. среднему шаблону будет сопоставлена самая левая удовлетворяющая ему последовательность символов).

Примеры:

*eColumn '\t' e2* - переменной *eColumn* будет сопоставлена последовательность символов до первого символа табуляции в тексте; если табулятор в тексте не обнаружен, то сопоставление с этим образцом потерпит неудачу;

*'//' eComment '\n' e2* - ищет конец однострочного комментария в исходном тексте на языке C++.

Упрощение образцов по сравнению с оригинальным Рефалом позволяет выполнять их сопоставление без откатов (концевыми сравнениями и линейным поиском плавающей части).

**Образцы с условиями.** В предложениях Рефал-0-функций после образца могут следовать *условия*. Условие – это логический предикат, применённый к *s*-переменной образца, и его требуемое истинностное значение (*'T'* или *'F'*). Предложение может содержать несколько условий. Для успешного сопоставления необходимо соответствие истинностных значений, возвращённых предикатами всех условий, требуемым. Предполагается, что предикаты описаны в окружающей программе как функции императивного языка. Их назначение – задание классов символов.

**Пример 2.2.2**

```
Identifier {
        s1 eIdent s2 e3, <IsFirstIdentChar s1>: 'T', <IsIdentChar s2>: 'F' =
                <EmitIdent s1 eIdent> s2 e3
}
```

Функция *Identifier* ожидает в начале переданного текста идентификатор переменной и выделяет его - ищет первый символ, который не является допустимым для идентификатора. Используются внешние предикаты, определяющие, является ли символ допустимым первым символом и допустимым последующим символом идентификатора, соответственно. Выделенный идентификатор передаётся внешней процедуре *EmitIdentifier* и возвращается последующая часть текста (предполагается, что *EmitIdent* возвращает пустой текст). Внешним процедурам, предназначенным для

передачи извлечённой информации в окружающую программу, мы назначаем префикс *Emit* и выделяем их синим цветом (примечание: среда *BlackBox* допускает произвольное шрифтовое и стилевое оформление исходных текстов программ).

**Выражения.** В правой части предложения (после знака "=") находится *выражение*, описывающее *подстановку*, которой должен быть заменён исходный текст, сообщённый функции. Подстановка формируется из строк символов, *s*-переменных, *e*-переменных и применений функций. Взаимный порядок *e*-переменных во всём подстановочном выражении должен соответствовать их порядку в образце, либо одна или обе переменных могут быть опущены.

Функции применяются к *подвыражениям*, которые имеют аналогичный вид. Таким образом, выражения имеют древовидную рекурсивную структуру произвольной вложенности. Допускается прямая и косвенная рекурсия в вызовах функций (очевидно, что хвостовая рекурсия в Рефале-0 может быть легко оптимизирована до цикла – разработанный транслятор выполняет такую оптимизацию как для прямой, так и для косвенной хвостовой рекурсии).

Для сопряжения с императивным окружением важен порядок применения функций в подстановочных выражениях. Функции применяются строго по порядку вхождения в выражение (слева направо), до применения функции применяются в том же порядке все функции в её подвыражении (то есть в порядке от внутренних применений к внешним). Функции в Рефал-0-программе могут объявляться в любом порядке. Если в подстановочном выражении используется функция, отсутствующая среди объявлений, то она считается внешней – объявленной в окружающей императивной программе.

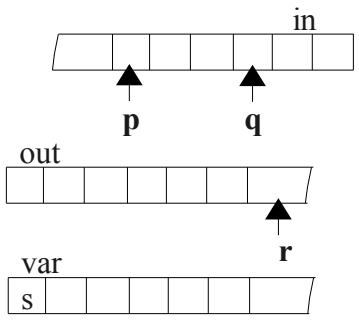
**Пример 2.2.3**

Функция *SolvePath* упрощает файловый путь *Unix*, в котором присутствуют откаты на уровень вверх (*/..*).

```
SolvePath {
        '/../' e1 = <SolvePath '/' e1>; /* абсолютный путь */
        '../' e1 = '../' <SolvePath e1>; /* относительный путь */
        e1 '/../' e2 = <SolvePath <LevelUp e1> e2>;
        e1 '/..' = <LevelUp e1>;
        e1 = e1
}

LevelUp {
        e1 '/' e2       =       e1 '/' <LevelUp e2>;
        e1      =       ''
}
```

# 3 Описание абстрактной строковой машины (АСМ)




АСМ имеет два рабочих стека – *in* и *out*, в ячейках которых содержатся символы. Стек *in* содержит текст, который предстоит обработать, на стеке *out* формируется результат. Регистр *r* указывает на вершину стека *out* (первую свободную ячейку). Интервал *[p, q)* на стеке *in* представляет собой *поле зрения* машины (фрагмент текста, подлежащий, в зависимости от контекста, либо сопоставлению с образцом, либо применению функции, либо выводу в стек *out*). Кроме того, машина имеет набор непосредственно адресуемых ячеек *vars*, используемых функциями в качестве временной памяти. Регистры *p* и *r* являются частью глобального состояния машины. Регистр *q* имеет смысл в контексте текущего вызова функции. Регистр *r* явно не указывается ни в каких операциях, будучи используем только неявно.

Опишем язык машины, используя РБНФ-нотацию. Предполагается, что код АСМ имеет не текстовый, а структурированный древовидный формат (при разработке транслятора для этого были использованы Рефал-подобные древовидные выражения, реализованные на Компонентном Паскале. Можно считать такой формат изоморфным с XML и S-выражениями языка *Scheme*). На верхнем уровне структура программы для АСМ повторяет структуру Рефал-программы:

**Program** = {Function}.
**Function** = Sentence {Sentence}.

Выражение может содержать до четырёх необязательных частей:

**Sentence** = [Match] [Store] [Process] [Loop **|** Call].

На императивном языке интерфейс АСМ-функции может быть описан следующим образом:

PROCEDURE SomeFunction (VAR in: ARRAY OF CHAR; VAR p: INTEGER; q: INTEGER; VAR out: ARRAY OF CHAR; VAR r: INTEGER).

## 3.1 Анализ поля зрения

Часть *Match* представляет собой набор условий, которым должны соответствовать символы в поле зрения *[p, q)* машины, чтобы было применено данное предложение.

**Match** = Metric (Look **|** Compare **|** Predicate) {Look **|** Compare **|** Predicate}.
**Metric** = RECORD head, med, tail: INTEGER; strict: BOOLEAN END;
**Look** = RECORD reg, offs: INTEGER; char: CHAR END;
**Compare** = RECORD s1, s2: RECORD reg, offs: INTEGER END END;
**Predicate** = RECORD reg, offs: INTEGER; func: Name; expect: BOOLEAN END;

Метрика *Metric* задаёт длины начала, конца и плавающей средней части текста, которые предполагаются данным предложением. Метрика накладывает следующее ограничение на поле зрения:

strict & (q – p = head)
OR ~strict & (q – p >= head + med + tail).

Каждое условие применяется к одному символу (*Compare* – к двум) из

поля зрения. Символ задаётся регистром и смещением относительно регистра. Кроме регистров *p* и *q* может быть указан регистр плавающей части *i*. Если среди условий *Match* присутствуют условия относительно регистра *i*, то АСМ пытается выполнить поиск такого *i*: *(p + head) <= i < (q – tail – med)*, при котором будут удовлетворены все эти условия. Смещения в условиях должны удовлетворять условиям:

```
(reg = p) & (0 <= offs < metric.head)
OR (reg = q) & (metric.tail <= offs < 0)
OR (reg = i) & (0 <= offs < metric.med).
```

Условия имеют следующую семантику (угловыми скобками <> будем обозначать разыменование указателя регистра или имени функции):

**Look**: in[<reg> + offs] = char,
**Compare**: in[<s1.reg> + s1.offs] = in[<s2.reg> + s2.offs],
**Predicate**: <func>(in[<reg> + offs]) = expect.

В случае, если текст в поле зрения удовлетворяет всем условиям секции *Match* текущего предложения, АСМ выполняет остальные его секции и опускает последующие предложения функции. В случае, если в текущем предложении отсутствует секция *Match*, оно выполняется безусловно, а последующие предложения игнорируются (не имеют смысла). После выполнения секции *Match* плавающий регистр *i* (если он используется в данном предложении) получает фиксированное значение.

Очевидно, что образцы и условия предложений *Рефала-0* целиком отображаются в секции *Match* соответствующих предложений АСМ, при этом их трансляция тривиальна. Концевые шаблоны образуют условия относительно регистров *p* и *q*, плавающий шаблон (в случае его наличия) – условия относительно регистра *i*. Метрика образуется из длин соответствующих шаблонов.

## 3.2 Временная память

Перед преобразованием текста в поле зрения АСМ может потребоваться сохранение отдельных его символов во временной памяти (это необходимо для преобразований, перестанавливающих или копирующих символы текста, т.е. *s*-переменные *Рефала-0*). Сохранение выполняет секция *Store* предложения.

**Store** = Var {Var}.
**Var** = RECORD reg, offs: INTEGER END.

Символы *in[<reg> + offs]* помещаются в ячейки временной памяти *vars*, в том порядке, в каком расположены инструкции *Var*.

## 3.3 Преобразование текста

Секция *Process* предложения выполняет преобразование текста. Преобразования включают в себя операции над вершинами стеков *in* и *out*, с изменением глобальных регистров *p* и *r*. Предложение может сформировать левую часть результирующего текста на вершине *out*, а правую – в поле зрения на вершине *in*, по своему усмотрению. Это позволяет выполнять преобразования эффективно, с минимизацией лишних перемещений символов. Все операции секции *Process* имеют своим параметром не позицию, а длину интервала. Поэтому вместо регистров *p*, *q*

и *i* используются новые регистры *d*, *dl* и *dr*, связанные с первыми следующими равенствами (*dl* и *dr* определены, только если определён *i*):

d = q – p, dl = i – p, dr = q – i.

Также может быть указан регистр *null*, если интервал определяется только константой. Локальный регистр *q* секцией *Process* неявно используется для выбора фрагмента текста для обработки. Секция *Process* может содержать следующие инструкции:

**Go** = RECORD reg, add: INTEGER END;
**Select** = RECORD reg, add: INTEGER END;
**Out** = RECORD END;
**OutS** = RECORD END;
**OutC** = RECORD char: CHAR END;
**OutV** = RECORD var: INTEGER END;
**PutC** = RECORD char: CHAR END;
**PutV** = RECORD var: INTEGER END;
**Call** = RECORD func: Name END;
**BeginSeq** = RECORD END;
**UseSeq** = RECORD END.

Инструкции имеют следующую семантику:

**Go**: p := p + <reg> + add (пропускает - «выбрасывает» - символы со стека *in*);

**Select**: q := q + <reg> + add (выбирает для последующей операции символы на стеке *in*);

**Out**: out[r .. r + q – p-1] := in[p .. q-1]; r := r + q – p; p := q (перемещает выбранные символы со стека *in* на стек *out*);

**OutS**: out[r] := in[p]; r := r + 1; p := p + 1 (перемещает один символ с *in* на *out*);

**OutC**: out[r] := char; r := r + 1 (помещает на стек *out* символ *char*);

**OutV**: out[r] := vars[var]; r := r + 1 (помещает на стек *out* запомненное значение из ячейки *var*);

**PutC**: p := p – 1; in[p] := char (помещает на стек *in* символ *char*);

**PutV**: p := p – 1; in[p] := vars[var] (помещает на стек *out* запомненное значение из ячейки *var*);

**Call**: <func>(in, p, q, out, r) (вызывает функцию *func* с полем зрения [p, q)).

**BeginSeq**: запоминает текущее значение *r*.

**UseSeq**: in[p – (r – r0) .. p-1] := out[r0 .. r-1]; p := p – (r-r0); r := r0 (перемещает результат инструкций между скобками *BeginSeq* и *UseSeq* полностью на стек *in*, для обработки следующими инструкциями).

Скобки *BeginSeq* и *UseSeq* могут располагаться вложенно друг в друга.

Данный набор инструкций позволяет достаточно просто и эффективно представлять преобразования текста в поле зрения. Стратегия трансляции подстановочного выражения *Рефала-0* в инструкции АСМ заключается в том, чтобы, насколько это возможно, использовать «близость» исходного текста к требуемому результату (в идеале, если текст не требуется преобразовывать, копирование не должно выполняться вообще). Транслятор последовательно сравнивает образец с подстановкой и вычисляет длину очередной совпадающей части. Если совпадение наблюдается до конца и образца, и выражения, то никакие инструкции не генерируются (перемещение на стек *out* может выполнять функция верхнего уровня – но только в случае необходимости). При первом несовпадении генерируются инструкции перемещения совпадающей части на стек *out* - в зависимости от длины совпадения, либо *OutS*, либо *Select-Out*. Для несовпадающих элементов генерируются инструкции вывода *OutC* и *OutV*; для бесполезных (не встречающихся в подстановке) частей

образца – инструкция пропуска *Go*. Когда несовпадающий элемент оказывается на вершине текста, могут генерироваться инструкции *PutC* и *PutV* (правило не отрывать отдельные элементы от остального текста даёт выигрыш в подстановках с применениями функций).

Для применения функции в подстановке текст-аргумент должен быть сформирован на стеке *in*, затем выбран инструкцией *Select*, после которой следует инструкция *Call*. Выражение-аргумент у функции может быть совпадающим с образцом – в таком случае дополнительной подготовки не требуется. Для более сложного аргумента требуется генерация инструкций по вышеописанной схеме. Эти инструкции ограничиваются аргументными скобками *BeginSeq* и *UseSeq*. Инструкция *UseSeq* выполняет полный перенос текста, сформированного вложенными инструкциями, на стек *in*, для дальнейшего применения функции.

## 3.4 Оптимизация концевой рекурсии

В случае, если рекурсивный *Call* является последней инструкцией в секции *Process*, то такая рекурсия оказывается концевой и может быть легко оптимизирована до цикла. В случае прямой концевой рекурсии вместо последнего *Call* в секции *Process* добавляется инструкция *Loop* в конце конструкции *Sentence*. Косвенная концевая рекурсия может быть выявлена в транзитивном замыкании графа концевых вызовов. Косвенная концевая рекурсия фактически представляет собой детерминированный конечный автомат, состояниями которого являются функции, находящиеся в петле рекурсии. Таким образом, косвенная концевая рекурсия может быть преобразована из вложенных вызовов на стеке к вызовам в цикле. Косвенный концевой *Call* выносится из секции *Process* в конец конструкции *Sentence*.

# 4 Трансляция кода АСМ в императивный код

Выше мы уже привели описание интерфейса АСМ-функции. Для практического использования его следует дополнить ещё одним выходным параметром – кодом ошибки:

```
PROCEDURE SomeFunction (VAR in: ARRAY OF CHAR; VAR p: INTEGER; q: INTEGER; VAR out: ARRAY OF CHAR; VAR r: INTEGER; OUT error: INTEGER)
```

В случае, если поле зрения не удалось сопоставить условиям ни одного из предложений, функция должна вернуть код ошибки (её собственный уникальный номер). Активации функций, вызвавших данную, должны прекратить обработку текста и передать вверх полученный код ошибки. Внешние процедуры, описанные в императивной программе, имеют такой же интерфейс и могут возвращать собственные коды ошибок (например, сообщать о нарушении каких-либо семантических правил). Также возможна организация обработки и восстановления после ошибки в разборе (см. п. 4.4).

В интерфейс функции может быть включён дополнительный параметр – указатель на разделяемые данные, используемые программой обработки текста. Рефал-функции игнорируют этот параметр, он используется внешними процедурами, которые вызываются Рефал-программой.

Перед вызовом главной функции Рефал-программы должен быть подготовлен стек с исходным текстом и выходной стек достаточного размера. Следует предусмотреть дополнительное место в обоих стеках для промежуточных операций в функциях. После окончания выполнения Рефал-программы результирующий текст формируется конкатенацией двух стеков: `out[0 .. r-1] + in[p .. q-1]` (*q* не изменяется Рефал-программой, обычно равно длине массива *in*).

Мы описали реализацию на основе массивов, однако без значительных изменений может быть сделана реализация на основе буферизованных потоков ввода-вывода, если из-за размеров обрабатываемых текстов требуется пространственная, а не временная оптимизация.

## 4.1 Трансляция предложений и условий

Последовательность предложений, очевидно, транслируется в последовательность условий IF. Вводится переменная *match* – признак успешного сопоставления с образцом.

```
match := FALSE; d := q – p;
IF ~ match & (метрика) & ( простые_условия_на_поле_зрения ) THEN
    дополнительные_проверки_на_поле_зрения;
    IF успешно_сопоставлено THEN
        match := TRUE;
        преобразования
    END
END;
IF ~match ...
    ...
END;
IF ~match THEN error := уникальный_код_функции END;
```

Условия относительно регистров *p* и *q* транслируются в термы основного IF:

```
IF ~match & (d >= 5) & (in[p] = 'a') & (in[q-1] = 'b') & ~IsDigit(in[p+1]) THEN ...
```

При наличии условий относительно плавающего регистра i организуется линейный поиск в средней части текста (с учётом метрики):

```
IF ~match & ... THEN
    i := p + 2; WHILE (i < q – 1) & ~IsLetter(in[i]) DO INC(i) END;
    IF i < q – 1 THEN
        match := TRUE;
        ...
```

## 4.2 Трансляция преобразований и рекурсии

Отображение секций *Store* и *Process* в императивный код тривиально (непосредственно следует из приведённого в п. 3.3 описания семантики операций). Операции перемещения символов кодируются непосредственным циклом. После каждого вызова функции помещается проверка кода ошибки `IF error = 0 THEN продолжение END`.

Для прямой концевой рекурсии используется цикл с постусловием `REPEAT error := 0; ... UNTIL error # спец_значение`, в который помещаются все предложения функции. В таком случае инструкция АСМ *Loop* транслируется в `error := спец_значение`.

В интерфейс косвенно-рекурсивных функций вводится дополнительный параметр *tailCall*:

PROCEDURE SomeFunction (VAR in: ARRAY OF CHAR; VAR p: INTEGER; q: INTEGER; VAR out: ARRAY OF CHAR; VAR r: INTEGER; OUT tailCall: Refal0TailCall; OUT error: INTEGER),

где Refal0TailCall = (VAR in: ARRAY OF CHAR; VAR p: INTEGER; q: INTEGER; VAR out: ARRAY OF CHAR; VAR r: INTEGER; OUT tailCall: Refal0TailCall; OUT error: INTEGER).

Цикл для косвенной рекурсии организуется в точке вызова функции:

SomeFunction(in, p, q, out, r, tail, error); WHILE (error = 0) & (tail # NIL) DO tail(in, p, q, out, r, tail, error) END,

а концевой *Call* отображается в *tailCall := OtherFunction*.

## 4.3 Пример трансляции Рефал-0-программы

Рассмотрим промежуточное представление и императивный код, генерируемые для функций из примера 2.2.3. Промежуточное представление запишем в древовидном виде, указывая тип узла как *ИмяТипа* =, а его элементы далее в скобках, разделяя их запятыми.

**Пример 4.3.1 – Промежуточное представление для примера 2.2.3**

```
Program = (
Function = ('SolvePath',
   Sentence = (
      Match = ( Look=(p, 0, '/'), Look=(p, 1, '.'), Look=(p, 2, '.'), Look=(p, 3, '/') ),
      Process = ( BeginSeq=, OutS=, Go=(null, 3), Select=(d, -4), UseSeq= ),
      Loop
   ),
   Sentence = (
      Match = ( Look=(p, 0, '.'), Look=(p, 1, '.'), Look=(p, 2, '/') ),
      Process = ( Select=(null, 3), Out=, Select=(d, -3) ),
      Loop
   ),
   Sentence = (
```

```
            Match = ( Look=(i, 0, '/'), Look=(i, 1, '.'), Look=(i, 2, '.'), Look=(i, 3, '/') ),
            Process = ( BeginSeq=, Select=(dl, 0), Call=('LevelUp'), Out=, Go=(null, 4),
Select=(dr, -4), UseSeq= ),
            Loop
         ),
         Sentence = (
            Match = ( Look=(q, -3, '/'), Look=(q, -2, '.'), Look=(q, -1, '.') ),
            Process = ( Select=(d, -3), Call=('LevelUp'), Out=, Go=(null, 3) ),
            Loop
         )
      ),
      Function = ('LevelUp',
         Sentence = (
            Match = ( Look=(i, 0, '/') ),
            Process = ( Select=(dl, 1), Out=, Select=(dr, -1) ),
            Loop
         ),
         Sentence = (
            Process = ( Go=(d, 0) )
         ))
```

**Пример 4.3.2 – Императивный код для примера 2.2.3**

```
      PROCEDURE ^ ParseLevelUp (VAR par: Parser; VAR in: ARRAY OF CHAR; VAR p:
INTEGER; q: INTEGER; VAR out: ARRAY OF CHAR; VAR r: INTEGER; OUT error: INTEGER);

      PROCEDURE ParseSolvePath (VAR par: Parser; VAR in: ARRAY OF CHAR; VAR p:
INTEGER; q: INTEGER; VAR out: ARRAY OF CHAR; VAR r: INTEGER; OUT error: INTEGER);
         VAR d: INTEGER; match: BOOLEAN; j0, j1, i, dl, dr: INTEGER; seq: ARRAY 1 OF
INTEGER;
      BEGIN
         REPEAT error := 0;
            d := q - p; match := FALSE;
            IF ~match  & (d >= 4) & (in[p] = '/') & (in[p+1] = '.') & (in[p+2] = '.') & (in[p+3]
= '/') THEN
               match := TRUE;
               q := p; seq[0] := r; out[r] := out[p]; INC(r); INC(p); q := p; INC(p, 3); q := p;
INC(q, d-4); IF r # seq[0] THEN j0 := r-1; j1 := p-1; WHILE j0 >= seq[0] DO in[j1] :=
out[j0]; DEC(j0); DEC(j1); END; r := j0 + 1; p := j1 + 1 END;
               error := MIN(INTEGER);
            END;
            IF ~match  & (d >= 3) & (in[p] = '.') & (in[p+1] = '.') & (in[p+2] = '/') THEN
               match := TRUE;
               q := p; INC(q, 3); j0 := p; j1 := r; WHILE j0 < q DO out[j1] := in[j0]; INC(j0);
INC(j1) END; p := j0; r := j1; q := p; INC(q, d-3);
               error := MIN(INTEGER);
            END;
            IF ~match  & (d >= 4) THEN
               i := p + 0;
               WHILE (i <= q - 4) & ~( (in[i] = '/') & (in[i+1] = '.') & (in[i+2] = '.') & (in[i+3]
= '/') ) DO INC(i) END;
               IF i <= q - 4 THEN
                  dl := i - p; dr := q - i; match := TRUE;
                  q := p; seq[0] := r; INC(q, dl); ParseLevelUp(par, in, p, q, out, r, error);
                  IF error = 0 THEN
                       j0 := p; j1 := r; WHILE j0 < q DO out[j1] := in[j0]; INC(j0); INC(j1)
END; p := j0; r := j1; q := p; INC(p, 4); q := p; INC(q, dr-4); IF r # seq[0] THEN j0 := r-1;
j1 := p-1; WHILE j0 >= seq[0] DO in[j1] := out[j0]; DEC(j0); DEC(j1); END; r := j0 + 1;
p := j1 + 1 END;
                     error := MIN(INTEGER);
```

```
            END
         END;
      END;
      IF ~match  & (d >= 3) & (in[q-3] = '/') & (in[q-2] = '.') & (in[q-1] = '.') THEN
         match := TRUE;
         q := p; INC(q, d-3); ParseLevelUp(par, in, p, q, out, r, error);
         IF error = 0 THEN
            INC(p, 3);
         END
      END;
   UNTIL error # MIN(INTEGER)
END ParseSolvePath;

   PROCEDURE ParseLevelUp (VAR par: Parser; VAR in: ARRAY OF CHAR; VAR p: INTEGER;
q: INTEGER; VAR out: ARRAY OF CHAR; VAR r: INTEGER; OUT error: INTEGER);
      VAR d: INTEGER; match: BOOLEAN; i, dl, dr, j0, j1: INTEGER;
   BEGIN
      REPEAT error := 0;
         d := q - p; match := FALSE;
         IF ~match  & (d >= 1) THEN
            i := p + 0;
            WHILE (i <= q - 1) & ~( (in[i] = '/') ) DO INC(i) END;
            IF i <= q - 1 THEN
               dl := i - p; dr := q - i; match := TRUE;
               q := p; INC(q, dl+1); j0 := p; j1 := r; WHILE j0 < q DO out[j1] := in[j0];
INC(j0); INC(j1) END; p := j0; r := j1; q := p; INC(q, dr-1);
               error := MIN(INTEGER);
            END;
         END;
         IF ~match  THEN
            match := TRUE;
            q := p; INC(p, d);
         END;
      UNTIL error # MIN(INTEGER)
   END ParseLevelUp;
```

## 4.4 Восстановление после ошибок разбора

В случае, если сопоставление с образцом терпит неудачу, возвращается код ошибки, информирующий о том, в какой функции это случилось.

**Пример 4.4.1 – Передача информации о неудачном сопоставлении**

```
FuncName {
        s1 eName s2 e3, <IsFirstFuncChar s1>: 'T', <IsFuncChar s2>: 'F' =
                        <EmitFuncName s1 eName> s2 e3
 }
(* Declarations:
   CONST
      noMatchInFuncName = -1;
*)

   PROCEDURE FuncName (VAR in: ARRAY OF CHAR; VAR p: INTEGER; q: INTEGER; VAR
out: ARRAY OF CHAR; VAR r: INTEGER; OUT error: INTEGER);
      VAR d, q0, r0: INTEGER; match: BOOLEAN; i, dl, dr: INTEGER;
   BEGIN
      q0 := q;  r0 := r;
```

```
    d := q - p; match := FALSE;
    IF ~match  & (d >= 2) & IsFirstFuncChar(in[p]) THEN
      i := p + 1;
      WHILE (i <= q - 1) & ~( ~IsFuncChar(in[i]) ) DO INC(i) END;
      IF i <= q - 1 THEN
        dl := i - p; dr := q - i; match := TRUE;
        q := p; INC(q, dl); EmitFuncName(in, p, q, out, r, error);
      END;
    END;
    IF ~match THEN error := -1 END;
  END FuncName;
```

В этом примере может возвращаться ошибка *noMatchInFuncName*, где *FuncName* – имя функции, в которой произошла ошибка.

Также может быть назначена внешняя процедура – обработчик ошибок. Она должна иметь сигнатуру:

```
PROCEDURE (VAR par: ANYREC; level: INTEGER; VAR in: ARRAY OF CHAR; VAR p: INTEGER; q: INTEGER; VAR out: ARRAY OF CHAR; VAR r: INTEGER; VAR error: INTEGER)
```

В конце сгенерированной процедуры в таком случае добавляется вызов:

```
IF error # 0 THEN SyntaxError (par, -1, in, p, q0, out, r, error); r := r0 END
```

где первым параметром передаётся общий контекст, вторым – идентификатор процедуры, из которой вызывается обработчик. Остальные параметры – контекст разбора, который может быть изменён обработчиком. Обработка может быть произведена на более высоком уровне от места возникновения ошибки. Обработчик может сохранить информацию об ошибке, изменить контекст разбора так, чтобы работа анализатора могла быть продолжена, и «погасить» ошибку (error := 0).

**Пример 4.4.2 – Восстановление после ошибки в синтаксическом анализаторе Рефала-0**

```
PROCEDURE SyntaxError (VAR par: ANYREC; level: INTEGER; VAR in: ARRAY OF CHAR; VAR p: INTEGER; q: INTEGER; VAR out: ARRAY OF CHAR; VAR r: INTEGER; VAR error: INTEGER);
    VAR e: Structs.ErrorInfo;
    PROCEDURE Complete;
    BEGIN
      error := 0; par(Parser).errorMarked := FALSE
    END Complete;
  BEGIN
    IF ~par(Parser).errorMarked THEN
      e := ErrorInfo(error, in, p); INC(e.posInSource, par(Parser).errorCorr);
      par(Parser).errors.Insert(e, par(Parser).errors.last);
      par(Parser).errorMarked := TRUE
    END;
    CASE level OF
    | r0Expect:
      INC(p)
    | r0Pattern, r0Condition:
      WHILE (p < q) & ~( (in[p] = ',') OR (in[p] = '=') ) DO INC(p) END;
      Complete
    | r0Expression:
      WHILE (p < q) & ~( (in[p] = '>') OR (in[p] = ';') OR (in[p] = '}' ) ) DO INC(p)
```

```
END;
            Complete
          | r0Sentence:
            WHILE (p < q) & ~( (in[p] = ';') OR (in[p] = '}' ) ) DO INC(p) END;
            Complete
          | r0Function:
            WHILE (p < q) & ~( (in[p] = '}' ) ) DO INC(p) END; INC(p);
            Complete
          | r0Main:
            Complete
          ELSE
          END
       END SyntaxError;
```

# 5 Архитектура транслятора и пакет *Rocot*

На первом уровне детализации транслятор языка Рефал-0 оказывается состоящим из трёх частей:

- синтаксический анализатор (исходный текст => синтаксическое дерево);
- транслятор в промежуточное представление (синтаксическое дерево => код абстрактной строковой машины);
- генератор императивного кода (код АСМ => исходный текст на одном из императивных языков программирования).

Для модулей трансляторов характерен обмен сложными динамическими структурами данных, введёнными для представления таблиц символов, синтаксических деревьев, промежуточного кода и т.д. Имеются традиционные подходы к проектированию этих структур данных. Эти подходы во многом были ориентированы на старые языки программирования и на максимальную оптимизацию. С одной стороны, Оберон-языки предоставляют новые степени свободы за счёт мощной типизации и сборки мусора, с другой стороны, оптимизация в ущерб прозрачности транслятора в настоящее время не имеет смысла (и не только из-за роста быстродействия техники, но и из-за того, что хорошо спроектированные языки программирования дают огромный выигрыш в скорости их компиляции). Таким образом, имеет смысл упростить и унифицировать работу с внутренними и внешними структурами данных трансляторов.

## 5.1 Универсальное древовидное представление

Решение проблемы представления данных было заимствовано из оригинального Рефала В.Ф. Турчина. В нём для этих целей используются типизированные древовидные выражения — фактически, списки термов, каждый из которых может быть тоже выражением. Выражения могут иметь первым элементом метки типа (так называемые символы-имена), которые динамически распознаются при сопоставлении с образцом. Такое представление является изоморфным к модели данных XML и к S-выражениям языка Scheme[1].

Мы реализовали универсальные древовидные выражения как библиотечное средство (модуль *RocotStructs*). Они были использованы для представления всех структур данных транслятора (синтаксического дерева, таблиц имён, промежуточного кода). Работа с типизированными данными в Компонентном Паскале проста и удобна, выразительность кода не проигрывает функциональным языкам. Главную роль здесь играет оператор динамической селекции типа WITH, применяемый к расширяемым записям. Он выполняет роль простейшего сопоставления с образцом. Ниже приведён сокращённый интерфейс модуля *RocotStructs*.

1 Существует спецификация представления XML-данных в форме S-выражений – SXML, предложенная разработчиками XML СУБД Sedna (ИСП РАН, http://modis.ispras.ru/).

```
DEFINITION RocotStructs;

      TYPE
         NotedRecord = POINTER TO ABSTRACT RECORD
            (* ... поддержка пометок для узлов дерева ... *)
         END;

(* Узел дерева *)
         Term = POINTER TO ABSTRACT RECORD (NotedRecord)
            next-, prev-: Term;
            pos-: INTEGER;
            (t: Term) Expr (): Expr, NEW, EXTENSIBLE;
            (t: Term) HandleMsg (VAR msg: Message), NEW, EMPTY
         END;

(* Узел дерева, содержащий подузлы – собственно, «древовидное выражение» *)
         Expr = POINTER TO EXTENSIBLE RECORD (Term)
            first-, last-: Term;
            (e: Expr) Insert (t, after: Term), NEW;
            (e: Expr) Paste (source: Expr; from, to, after: Term), NEW;
            (e: Expr) Delete (from, to: Term), NEW;
         END;

(* По дереву могут передаваться типизированные сообщения через Term.HandleMsg *)
         Message = ABSTRACT RECORD  END;

         ErrorInfo = POINTER TO EXTENSIBLE RECORD (Term)
            posInSource, code: INTEGER;
            params: ARRAY 3 OF ARRAY 24 OF CHAR;
         END;

         Name = POINTER TO ARRAY OF CHAR;

(* Абстрактный интерфейс таблицы символов *)
         SymTab = POINTER TO ABSTRACT RECORD (Term)
            (st: SymTab) AddUnicum (IN name: ARRAY OF CHAR; rec: ANYPTR; OUT ok:
BOOLEAN), NEW, ABSTRACT;
            (st: SymTab) Add (IN name: ARRAY OF CHAR; rec: ANYPTR), NEW, ABSTRACT;
            (st: SymTab) Look (IN name: ARRAY OF CHAR): ANYPTR, NEW,

            (st: SymTab) First (IN name: ARRAY OF CHAR): ANYPTR, NEW, ABSTRACT;
            (st: SymTab) Next (): ANYPTR, NEW, ABSTRACT;
            (st: SymTab) End, NEW, ABSTRACT;

            (st: SymTab) FirstName (OUT name: ARRAY OF CHAR), NEW, ABSTRACT;
 ABSTRACT;
            (st: SymTab) NextName (OUT name: ARRAY OF CHAR), NEW, ABSTRACT;
            (st: SymTab) EndName, NEW, ABSTRACT;

            (st: SymTab) Delete (IN name: ARRAY OF CHAR; rec: ANYPTR), NEW, ABSTRACT;
            (st: SymTab) DeleteName (IN name: ARRAY OF CHAR), NEW, ABSTRACT;
            (st: SymTab) Clear, NEW, ABSTRACT;
            (st: SymTab) Copy (): SymTab, NEW, ABSTRACT;
         END;

(* Абстрактный интерфейс множества произвольных объектов. Множества используются в
алгоритме транзитивного замыкания графа вызовов функций. *)
         Set = POINTER TO ABSTRACT RECORD (Term) ... END;
```

```
        Directory = POINTER TO ABSTRACT RECORD
           (d: Directory) NewSet (avgCount: INTEGER): Set, NEW, ABSTRACT;
           (d: Directory) NewSymTab (avgCount: INTEGER): SymTab, NEW, ABSTRACT
        (* Инсталлируемая фабрика, создающая таблицы символов и множества. Базовая
реализация таблиц очень проста, т.к. основана на предположении, что в современных
языках программирования единое глобальное пространство имён отсутствует, а
локальные строго структурированы и не содержат много переменных – для каждого
локального пространства заводится своя таблица символов. Эту особенность также
отмечал Н. Вирт в своих последних работах.  *)
        END;

    VAR
       dir-: Directory;
       stdDir-: Directory;

    PROCEDURE LookupTerm (e: Expr; IN type: ARRAY OF CHAR; after: Term): Term;
    PROCEDURE SetDir (d: Directory);

END RocotStructs.
```

## 5.2 Передний план транслятора. Активное синтаксическое дерево.

Важнейшей особенностью описанных выше древовидных структур является то, что каждый узел дерева может иметь процедуру обработки типизированных сообщений – расширяемых записей Оберона [11]. Таким образом, синтаксическое дерево становится активным. Этот механизм является основной переднего плана компилятора (модуль *RocotRefal0Front*, см. *Приложение А*).

Анализатор, написанный на Рефале-0, разбирает входной поток (нет специального разделения между лексическим и синтаксическим уровнем – весь анализ реализован единообразно, как набор рекурсивных Рефал-функций) и передаёт результат через вызовы нескольких внешних Оберон-процедур. Эти процедуры преобразуют информацию в типизированные сообщения, которые передаются корню синтаксического дерева. Далее каждый узел дерева либо обрабатывает сообщение непосредственно, трансформируя себя, либо передаёт его на нижний уровень. Таким образом, анализатор на Рефал-0 генерирует на основе исходного текста «события», которые преобразуются в сообщения и передаются активному синтаксическому дереву, которое в соответствии с «событиями» изменяет себя.

Такая архитектура позволила получить чрезвычайно прозрачную структуру исходного текста транслятора, легко доступную для изменения и развития. Разработанный транслятор подтвердил наше убеждение в том, что алгоритмы обработки хорошо структурированных данных на Обероне (как хорошо типизированном языке со сборкой мусора) по простоте и выразительности не уступают функциональным языкам программирования.

Идея активных структур данных может найти самое широкое применение в сложных задачах программирования.

Интересно было обнаружить тезис о родстве активных структур данных и сентенциального (марковского) стиля программирования в книге Н.Н. Непейводы и И.Н. Скопина «Основания программирования»:

*«Конечно, реализовывать аналитические вычисления можно самыми разнообразными способами. Можно упрятать реализацию всех нужных преобразований в каком-либо прикладном пакете (как это сделано, например, в Maple) и даже не заметить, что мы имеем дело с сентенциальными по сути своей задачами. Однако этот путь ведёт лишь к таким решениям, которые нужны "здесь и сейчас": программист, к примеру, не сможет подсказать программе символического дифференцирования, что она имеет дело с неким частным случаем, для которого можно резко сократить перебор вариантов. Когда требуется проводить подобные преобразования систематически, удобнее снабдить перерабатываемые данные дополнительными атрибутами, направляющими вычисления в нужное русло. Иными словами, полезно делать структурные единицы данных активными* (выделение авторов - И.Е.). *А это - привнесение в программу, обрабатывающую такие данные, элементов сентенциального стиля, пусть даже без использования подходящей языковой поддержки. Преимущества, которые при этом появляются, непосредственно следуют из того, что данный вид обработки является сентенциальным по своей сути, требующим адекватных средств реализации.»* [13, п. 13.4.1].

**Пример 5.2.1 – Устройство переднего плана транслятора**

**Анализатор, написанный на Рефале-0:**

```
Program {
        /* Program = { Function [";"] } . */
        ''      =       '';
        ' ' e1  =       <Program e1>;
        e1      =       <Program <Optional ';' <Function e1>>>
}

Function {
        /* Function = FuncName "{" Sentence { ";" Sentence } [";"] "}" . */
        e1      =       <EmitNewFunc> <Expect '}' <Block <Sentence <Expect '{'
<FuncName e1>>>>>
}

FuncName {
        s1 eName s2 e3, <IsFirstFuncChar s1>: 'T', <IsFuncChar s2>: 'F' =
                <EmitFuncName s1 eName> s2 e3
}

Block {
        ' ' e1  =       <Block e1>;
        ';' e1  =       <Block <EmitNewSent><Sentence e1>>;
        e1      =       e1
}
```

**Связующие процедуры:**

```
PROCEDURE EmitNewFunc (VAR par: ANYREC; VAR in: ARRAY OF CHAR; VAR p:
INTEGER; q: INTEGER; VAR out: ARRAY OF CHAR; VAR r: INTEGER; OUT error: INTEGER);
   VAR msg: NewFuncMsg;
BEGIN
   par(Parser).program.HandleMsg(msg);
```

```
    error := msg.error
END EmitNewFunc;

PROCEDURE EmitFuncName (VAR par: ANYREC; VAR in: ARRAY OF CHAR; VAR p:
INTEGER; q: INTEGER; VAR out: ARRAY OF CHAR; VAR r: INTEGER; OUT error: INTEGER);
    VAR msg: FuncNameMsg;
BEGIN
    Strings.Extract(in, p, q - p, msg.name);
    par(Parser).program.HandleMsg(msg);
    IF msg.error = 0 THEN p := q ELSE error := msg.error END
END EmitFuncName;

PROCEDURE EmitNewSent (VAR par: ANYREC; VAR in: ARRAY OF CHAR; VAR p:
INTEGER; q: INTEGER; VAR out: ARRAY OF CHAR; VAR r: INTEGER; OUT error: INTEGER);
    VAR msg: NewSentMsg;
BEGIN
    par(Parser).program.HandleMsg(msg);
    error := msg.error
END EmitNewSent;
```

**Обработчики узлов активного дерева:**

```
PROCEDURE (f: Function) HandleMsg* (VAR msg: Structs.Message);
    VAR s: Sentence;
        nsm: NewSentMsg;
        symTab: Structs.SymTab;
        pat: Pattern;
        ok: BOOLEAN;
BEGIN
    WITH msg: NewSentMsg DO
      NEW(s); symTab := Structs.dir.NewSymTab(7); s.Insert(symTab, s.last);
      NEW(pat); s.Insert(pat, s.last);
      f.Insert(s, f.last)
    | msg: FuncNameMsg DO
      IF f.first = NIL THEN
        f.name := msg.name$;
        f.HandleMsg(nsm);
        f.Expr()(Program).first(Structs.SymTab).AddUnicum(f.name, f, ok);
        IF ~ok THEN
          msg.error := duplicatedFuncName
        END
      ELSE
        f.last.HandleMsg(msg)
      END
    | msg: Message DO
        f.last.HandleMsg(msg)
    ELSE END
END HandleMsg;

PROCEDURE (p: Program) HandleMsg* (VAR msg: Structs.Message);
    VAR f: Function;
BEGIN
    WITH msg: NewFuncMsg DO
      msg.error := 0;
      NEW(f);
      p.Insert(f, p.last)
    | msg: Message DO
      msg.error := 0;
      p.last.HandleMsg(msg)
```

```
    ELSE END
END HandleMsg;
```

## 5.3 Задний план транслятора

Задний план компилятора распадается на два уровня: транслятор из синтаксического дерева в дерево кода абстрактной строковой машины и генератор текста на требуемом императивном языке из кода АСМ. В модуле *RocotRefal0Back* (см. *Приложение Б*) реализован транслятор АСМ и генератор текста на Компонентном Паскале (подмножество, в основном совместимое с *Oberon*, *Oberon-2* и *Active Oberon*).

В модуле описаны узлы для кода АСМ (в соотв. с п. 3.3) и реализованы связанные процедуры *Build* для их построения из соответствующих структур синтаксиса *Рефала-0*. В процессе обхода древовидных структур выполняется их анализ с помощью селектора типа – оператора WITH – и параллельная генерация структур АСМ.

Наиболее сложен алгоритм построения секции *Process* (т.е. кода для выражений подстановки; процедура *Process.Build*). Общая стратегия уже была описана в п. 3.3. Выполняется последовательное сравнение образца с выражением подстановки и накопление длины совпадающей части (как <регистр АСМ> + const), при очередном несовпадении выполняется генерация инструкций *OutS*, *Out* для переноса неизменной части на выходной стек (выполняет процедура *Flush*). Код для подвыражений в вызовах функций обрамляется инструкциями *beginSeq* и *useSeq*.

Транслятор выполняет оптимизацию прямой и косвенной концевой рекурсии, генерируя циклы, как описано в п. 3.4. Анализ рекурсии производится на основе транзитивного замыкания графа вызовов функций. Транзитивное замыкание (процедура *RocotAlgo.TransitiveClosure*) выполняется алгоритмом «Workpile»[2].

Генерация исходного текста на императивном языке, как описано в разделе 4, является шаблонной задачей. Создание нового генератора, практически, может быть выполнено путём аккуратной замены строковых значений на новые, соответствующие целевому языку.

## 5.4 Инструментальная подсистема *Rocot*

Разработанная подсистема *Rocot* может использоваться и развиваться как инфраструктура для написания компиляторов в среде *BlackBox Component Builder*.

Центральной частью подсистемы *Rocot* является транслятор языка *Рефал-0*, реализованный методом частичной раскрутки. Он состоит из переднего плана (анализ исходного текста и преобразование в синтаксическое дерево - модуль *RocotRefal0Front*), заднего плана (трансляция в промежуточный код абстрактной строковой двухстековой машины, кодогенерация в исходный текст на Компонентном Паскале — модуль *RocotRefal0Back*) и объединяющего интерфейса (модуль *RocotRefal0*). Могут разрабатываться дополнительные генераторы для

2 Который, в отличие от классического алгоритма Воршалла, имеет квадратичную сложность на разрежённых графах, при этом не являясь более сложным в реализации.

различных языков; в настоящее время поддерживается генерация в исходный текст на C — модуль *RocotRefal0toC*.

Модуль *RocotLangCells* поддерживает удобную работу со вставками, написанными на других языках, в модулях Компонентного Паскаля. Вставки оформляются как складки *BlackBox* (*StdFolds*), которые с одной стороны содержат исходный текст, например, на Рефале-0, а с другой – сгенерированный текст на Компонентном Паскале.

**Пример 5.4.1 – Оформление вставки на Рефале-0**

```
  PROCEDURE EmitNewCond (VAR par: ANYREC; VAR in: ARRAY OF CHAR; VAR p: INTEGER; q:
INTEGER; VAR out: ARRAY OF CHAR; VAR r: INTEGER; OUT error: INTEGER);
    VAR msg: NewCondMsg;
  BEGIN
    par(Parser).program.HandleMsg(msg);
    error := msg.error
  END EmitNewCond;

  PROCEDURE ^ SyntaxError (VAR par: ANYREC; level: INTEGER; VAR in: ARRAY OF CHAR; VAR
p: INTEGER; q: INTEGER; VAR out: ARRAY OF CHAR; VAR r: INTEGER; VAR error: INTEGER);

⇨RocotRefal0 (parserParam = TRUE; funcPrefix = 'Parse'; exceptProc = 'SyntaxError');
    Preprocess {
        '\n' e1       =  ' ' <Preprocess e1>;
        '\t' e1       =  ' ' <Preprocess e1>;
        '\'' e1       =  '\'' <PreString e1>;
        '/*' e1       =  ' ' <PreComment e1>;
        s1 e2         =  s1 <Preprocess e2>;
        e1            =  e1
    }

    PreComment {
        '*/' e1       =  ' ' <Preprocess  e1>;
```

Заголовок вставки указывает имя модуля, ответственного за компиляцию данного языка, и специальные параметры, поддерживаемые этим модулем.

**Другая сторона вкладки:**

```
  PROCEDURE EmitNewCond (VAR par: ANYREC; VAR in: ARRAY OF CHAR; VAR p: INTEGER; q:
INTEGER; VAR out: ARRAY OF CHAR; VAR r: INTEGER; OUT error: INTEGER);
    VAR msg: NewCondMsg;
  BEGIN
    par(Parser).program.HandleMsg(msg);
    error := msg.error
  END EmitNewCond;

  PROCEDURE ^ SyntaxError (VAR par: ANYREC; level: INTEGER; VAR in: ARRAY OF CHAR; VAR
p: INTEGER; q: INTEGER; VAR out: ARRAY OF CHAR; VAR r: INTEGER; VAR error: INTEGER);

➔(* Begin of code that have been automatically generated by RocotRefal0 *)

(* Declarations:
   CONST
        r0Preprocess = -1;    r0PreComment = -2;    r0PreString = -3;    r0Expect = -4;    r0Optional = -5;    r0Main = -6;    r0Program = -7;
r0Function = -8;    r0FuncName = -9;    r0Block = -10;    r0Sentence = -11;    r0Pattern = -12;    r0FreePart = -13;    r0Template = -14;    r0String
= -15;    r0CharCode = -16;    r0CharVar = -17;    r0FreeVar = -18;    r0CondSection = -19;    r0Condition = -20;    r0CondConst = -21;
r0Expression = -22;
        noMatchInPreComment = -2;    noMatchInPreString = -3;    noMatchInExpect = -4;    noMatchInOptional = -5;    noMatchInFuncName = -9;
noMatchInString = -15;    noMatchInCharCode = -16;    noMatchInCharVar = -17;    noMatchInFreeVar = -18;    noMatchInCondition = -20;
noMatchInCondConst = -21;
```

Модуль *RocotLangCells* поддерживает трансляцию вставок как для

отдельного открытого документа, так и пакетную для группы модулей.

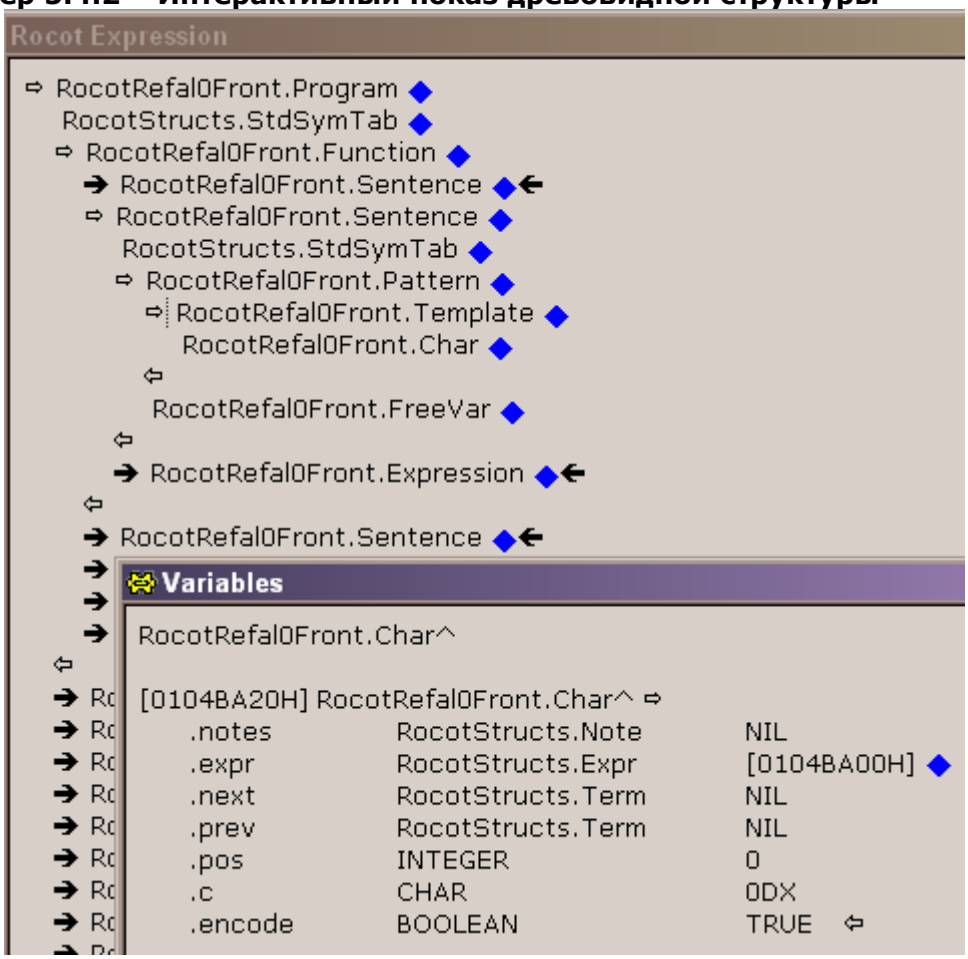
Определены обобщённые (абстрактные) интерфейсы потокового чтения-записи текстов, их стандартная реализация сопряжена с текстовыми документами *BlackBox*.

Реализован консольный (командный) интерфейс – модуль *RocotConsole*. На его основе собрана консольная версия транслятора Рефала-0 – модуль *RocotRefal0Con*.

Основой для различных структур данных являются универсальные древовидные выражения, реализованные в модуле *RocotStructs*. Кроме того, в этом модуле реализованы таблицы символов и множества объектов.

Модуль *RocotDebug* поддерживает показ древовидных структур в виде интерактивных документов, с возможностью навигации по указателям и т.п. Это средство основывается на стандартных механизмах метапрограммирования среды и реализовано крайне компактно – порядка 50 строк исходного текста.

**Пример 5.4.2 – Интерактивный показ древовидной структуры**




## 5.5 Кодировки

Среда *BlackBox* и язык *Компонентный Паскаль* используют *Unicode* (*UCS2*) в качестве основной кодировки. Это же касается и пакета *Rocot*.

В случае генерации Рефал-0-функций в текст на другом языке, нежели *Компонентный Паскаль*, используемая кодировка полностью будет зависеть от типов данных целевого языка.

## 6 Выводы

Кроме того, что задача, поставленная во введении этой работы, была успешно решена, нам представляется ценным некий опыт, полученный в процессе разработки транслятора и подсистемы *Rocot*.

Мы стремились достичь простоты и изящества во всех аспектах системы, равняясь на лучшие традиции Оберон-систем и швейцарской школы программирования. Результатом стало создание некоторого единого простого метода трансляции. Область методов трансляции в настоящее время кажется нам переусложнённой рудиментами (связанными с опытом неудачных и плохо спроектированных языков, подобных *C++*). Мы считаем, что любой язык, для которого нельзя создать транслятор с простой и ясной структурой, не может считаться удачным – и вина за избыточную сложность, которая будет наслаиваться на весь инструментарий для этого языка, лежит на его авторе. Н. Вирт опытом своих работ наглядно продемонстрировал противоположный образ действий, изложенный им в книге «Compiler Construction»[8]. Кроме того, нам запомнилась мысль из книги А. Шеня [13] в заключительном разделе «Общие замечания о разных методах разбора»: «*Дилетантский совет: если Вы сами проектируете входной язык, то не следует выпендриваться и употреблять одни и те же символы для разных целей – и тогда обычно несложно написать LL(1)-грамматику или рекурсивный анализатор*». Возможно, слишком часто разработчиками движет не стремление к поиску наилучшего решения, а боязнь последовать «дилетантским советам».

Широкое применение в разных проектах могут найти древовидные выражения (см. п. 5.1), которые делались по образу объектных выражений Рефала и изоморфны модели данных *XML* и *S*-выражениям языка *Scheme*. Подсистема *Rocot* может лечь в основу *XML*-инструментов для *BlackBox* и интерфейсов к *XML*-базам данных (в частности, к XML-СУБД «Седна»). Важно, что в хорошо типизированном языке со сборкой мусора нет необходимости заворачивать сложные структуры данных в объектные обёртки, подобные интерфейсу *DOM* для *XML*, которые вызывают большой перерасход памяти и неудобны в использовании. Это же замечание справедливо для служб доступа к семантической информации об исходных текстах, подобных *ASIS* (*Ada Semantic Interface Specification*) – вместо объектных интерфейсов служба может предоставлять клиенту непосредственно древовидную структуру; на основе *Rocot* планируется разработка аналогичных средств для Оберон-языков. В целом, представляется, что в современном программировании ещё недооценено влияние автоматического управления памятью на сложные алгоритмы и структуры данных.

Очень ценной представляется концепция (см. п. 5.2) активных структур данных, способных локально перестраиваться в соответствии с событиями-сообщениями, проходящими по ним. Обратим внимание, что такая концепция не реализуется эффективно в рамках массового ООП, основанного на вызове методов, для этого требуется эффективный механизм работы с расширяемыми типизированными сообщениями, который из массовых языков пока присутствует лишь в Оберонах[11].

В целом, немногочисленные, но мощные средства Оберон-языков

делают программирование задач, подобных разработке трансляторов, простым и выразительным. Это заставляет усомниться в часто звучащих заявлениях о принципиальном отставании алгоритмических языков по сравнению с функциональными. Проблемы скорее связаны с недоработками важнейших аспектов массовых языков С-семейства (при переизбытке в них третьестепенных средств).

**Ссылки**


1. Центральный сайт Рефал-сообщества. Refal.ru(.net).
2. Турчин В.Ф. РЕФАЛ-5. Руководство по программированию и справочник. http://refal.ru/rf5_frm.htm
3. Сайт разработчиков Рефал-Плюс. http://wiki.botik.ru/Refaldevel/WebHome
4. Гурин Р.Ф., Романенко С.А. Язык программирования Рефал-Плюс. http://wiki.botik.ru/Refaldevel/RefalPlusBook
5. Oberon page of the ETH Zurich. http://www.oberon.ethz.ch/
6. Niklaus Wirth, Jürg Gutknecht. Project Oberon: The Design of an Operating System and Compiler. - Zurich, 2005. http://oberoncore.ru/library/wirth_gutknecht_project_oberon_the_design_of_an_operating_system_and_compiler
7. Никлаус Вирт. Проектирование системы с нуля. // Перев. с англ. Р. Богатырёв. http://oberon2005.oberoncore.ru/paper/scratch.pdf Оригинал: Niklaus Wirth. Designing a System from Scratch // Structured Programming, № 10, 1989.
8. Niklaus Wirth. Compiler Construction. - Zurich, 2005. http://oberoncore.ru/library/compiler_construction
9. Оберон-технологии в России http://OberonCore.ru
10. Ермаков И.Е. Оберон-технологии: что это такое? - 2006. http://store.oberoncore.ru/lib/article/oberontech.pdf
11. Ермаков И.Е. Некоторые идеи архитектуры Оберон-систем. - 2007. http://store.oberoncore.ru/lib/article/DesignIdeas.pdf
12. Непейвода Н.Н., Скопин И.Н. Основания программирования. - 2002. http://ulm.udsu.ru/~nnn/fp.zip
13. Шень А. Программирование: теоремы и задачи. - М.: МЦНМО, 2004.
14. Ермаков И.Е. Встраиваемый язык обработки текстов РЕФАЛ-0 и разработка его транслятора на Компонентном Паскале. //Тезисы докладов XVI межд. школы-семинара «Новые информационные технологии-2008». - М.: МИЭМ, 2008.