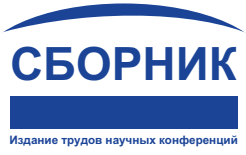
РОССИЙСКАЯ АКАДЕМИЯ НАУК
ИНСТИТУТ ПРОГРАММНЫХ СИСТЕМ
ЛАБОРАТОРИЯ АВТОМАТИЗАЦИИ ПРОГРАММИРОВАНИЯ

СБОРНИК ТРУДОВ
по функциональному языку программирования Рефал

II



ПЕРЕСЛАВЛЬ-ЗАЛЕССКИЙ
2015

УДК 004.42(063)
ББК 22.18

В874

Сборник трудов по функциональному языку программирования Рефал, том II // *Под редакцией А. П. Немытых.* — Переславль-Залесский: Издательство «СБОРНИК», 2015, **156** с. — ISBN 978-5-9905410-2-3

Книга является сборником статей и лекций, посвященных различным сторонам функционального языка программирования Рефал (автор В. Ф. Турчин), его реализаций и методам компиляции и оптимизации программ, написанных на Рефале.

Сборник может рассматриваться как учебное пособие по функциональному программированию и принципам реализации функциональных языков программирования.

Для программистов и студентов, специализирующихся в области функционального программирования и символьных вычислений.

ISBN 978-5-9905410-2-3

Предисловие редактора

Второй том сборника открывается лекциями по языку Рефал, прочитанными Александром В. Корлюковым в Гродненском государственном университете. Любую задачу, за которую брался Александр, он превращал в источник вдохновения. Увлекался ей сам и своим энтузиазмом притягивал людей, которым посчастливилось знать Александра.

Рефал излагается на примерах символьного программирования простых алгебраических задач. Предполагается, что читатель слушал введение в алгебру на первом курсе университета, либо способен самостоятельно разобраться в элементарной логике соответствующих алгебраических понятий.

Лекции были прочитаны до последней редакции синтаксиса Рефала, сделанной В. Ф. Турчиным. Я не стал исправлять текст Александра – подгонять его под этот новый синтаксис, – а дал лишь краткие поясняющие сноски. Читатель легко разберется в конкретных примерах программ.

Годовая экспедиция редактора Сборника в Поднебесную Империю заронила семена Рефала на берегах Длинной реки. Одним из китайских студентов был реализован новый компилятор `crefal` из языка Рефал-5 в язык сборки. Наша совместная статья является кратким отчетом об этой работе. К сожалению, я не смог найти деньги для продолжения поддержки интереса к Рефалу в Китае. Некоторые планы были осуществлены лишь частично. Например, не закончен перевод на китайский язык книги Валентина Фёдоровича Турчина «REFAL-5 Programming Guide and Reference Manual».

При реализации компилятора `crefal` на языке Рефал-5 возникла необходимость в расширении библиотеки встроенных функций Рефала-5. В данном сборнике описание новых функций дано в краткой заметке.

В этом томе мы начинаем публикацию заметок, основанных на опыте разработки суперкомпилятора SCP4 языка Рефал-5, об основных понятиях и алгоритмах метода суперкомпиляции.

В статье Марата Исламова представлен интересный результат размышлений о возможных направлениях развития Рефала. Автор предлагает свой диалект языка – Refal-D, в частности, содержащий конструкции, ориентированные на поддержку преобразований электронных интернет-документов, описанных на языке XML. Им же была реализована экспериментальная версия интерпретатора диалекта Refal-D.

Несколько лет назад метод оптимизации программ, написанных на функциональных языках программирования, названный В. Ф. Турчиным суперкомпиляцией, был применён для решения задач автоматической верификации глобальных свойств надежности программных моделей недетерминированных вычислительных систем. Например, для проверки указанных свойств некоторого семейства протоколов когерентности кэша.

Эти задачи, возникшие за рамками внутренней «рефальской кухни», явились толчком к стимулированию изучения и развития основных алгоритмов суперкомпиляции.

Одна из статей данного тома Сборника подробно излагает интересные результаты развития идей алгоритма Турчина, который используется для адекватного обобщения двух параметризованных стеков функций, возникающих в

процессе развертки суперкомпилируемой программы. Алгоритм основан на анализе истории возможных вычислений этой программы от первого стека до второго. Автор статьи показывает, как с помощью алгоритма Турчина можно верифицировать свойство секретности общения клиентов, использующих нетривиальные криптографические протоколы передачи сообщений, через открытый канал связи.

Статьи в сборнике расположены в порядке времени проведения соответствующих исследований.

Редактор искренне благодарит Антонину Непейвода за помощь в подготовке данного тома Сборника.

Андрей П. Немытых

Содержание

А. В. Корлюков

Введение в программирование на языке Рефал (с приложениями в алгебре)

Лекции были прочитаны автором в Гродненском государственном университете.

Библ. 11 наим.

Содержание

Лекции подготовлены к печати в 2001 г.

Введение

> Коммерческие программисты рассматривают рекурсию в целом как забавную безделушку, в то время как специалисты по искусственному интеллекту, если можно так выразиться, живут и дышат рекурсивно.
>
> Б. Шнейдерман «Психология программирования», 1984.

Машинно-независимый алгоритмический язык Рефал – РЕкурсивных Функций АЛгоритмический язык [5, 2] – ориентирован на работу с символьными данными. Возможности языка Рефал позволяют использовать его в таких областях, в которых использование традиционных языков программирования (Basic, Pascal, ФОРТРАН, АЛГОЛ, ПЛ-I) затруднительно, а именно:

- для произвольной обработки текстов, в частности для создания трансляторов с одних языков на другие,
- для вычислений с произвольной точностью,
- для обработки предложений на естественном языке (создание диалектов).

Простота и естественность внутренней структуры языка Рефал позволяют студентам изучить этот язык и основные приемы программирования буквально за несколько занятий. Поскольку на Рефале легко программируются[1] такие задачи, как дифференцирование произвольных функций, вычисления в конечных полях и в линейных группах, то представляется заманчивым использование языка Рефал при написании курсовых и дипломных работ. Примером может служить создание студентами системы КОЛИБРИ, осуществляющей вычисления в линейных группах небольших степеней.

Язык программирования Рефал реализован к настоящему времени практически на всех отечественных вычислительных машинах[2]. Он широко применяется при решении научных задач в различных областях математики, механики, физики. К сожалению, описание языка Рефал содержится лишь в трудно доступных для студентов печатных работах и в инструктивных материалах. Поскольку имеющаяся литература не годится для первоначального знакомства с языком Рефал, автор настоящего пособия в течении ряда лет читал лекции

[1] Время ещё раз показало, что всё коммерческое мимолётно, включая «коммерческие» мысли. Функциональные языки программирования уже не рассматриваются как «забавные безделушки». Производительность современных вычислительных машин позволяет при программировании размышлять в содержательных терминах задач, которые решает человек. – *Прим. ред.*

[2] В настоящее время понятия «отечественная вычислительная машина» и «отечественная операционная система» как таковые исчезли из обихода. И это прискорбно. Т. е. мы вынуждены сказать, что к настоящему времени имеются в свободном доступе исполняемые модули интерпретаторов и (полу)компиляторов Рефала для операционной системы Windows-*** (включая Windows Mobile), как и их исходные тексты, которые можно собрать под операционные системы семейства Unix и, возможно, можно собрать для других операционных систем (см. [10, 7]). – *Прим. ред.*

и проводил занятия по элементам программирования на Рефале. Отсутствие соответствующей литературы создавало свои трудности в обучении студентов.

Целью данного пособия является восполнение существующего пробела в учебной литературе, а также помощь в организации самостоятельной работы студентов по изучению основных элементов программирования на языке Рефал.

Мы не стремились изложить язык Рефал в полном объеме описания языка, которое поставляется вместе с реализацией. Дается описание того подмножества языка Рефал, которое называется Рефал-5 [6].

При отборе материала мы исходили из соображений наглядности и доступности. Большое количество примеров помогает студентам быстро перейти к самостоятельному программированию.

Язык Рефал является машинно-независимым языком программирования, поэтому лишь в некоторых случаях упоминается реализация языка. Все примеры и упражнения даются на входном языке трансляторов Рефала, в том виде, в котором они вводятся в компьютер.

§ 1. Описание языка Рефал

> Преподавание программирования – дело почти безнадежное, а его изучение – непосильный труд.
>
> Ч. Уэзерелл «Этюды для программистов», 1982

Имеется несколько способов изложения теоретических основ языка Рефал. В настоящих лекциях определения пп. 1, 3, 4, 6, 7 соответствуют работе [1]. Часть задач и примеров взяты из работ [2, 1], другая часть относится к рефальскому фольклору, и установить авторство многих задач или их элегантных решений не представляется возможным. Поэтому мы приводим все задачи без каких-либо ссылок и не претендуем на оригинальность в тех случаях, когда нам кажется, что задача сформулирована впервые.

1.1. Знаки, символы, переменные. Язык Рефал ориентирован на работу с символьными данными. Основными объектами, с которыми работает Рефал, являются символы.

Символы бывают простые и составные.

Простой символ – это обычный символ. При записи программы простой символ обрамляется апострофами, за исключением самого апострофа – он просто удваивается[3].

Составной символ – это либо составной символ-метка (детерминатив, имя функции), либо составной символ-число – макроцифра.

[3] Автор описывает старый синтаксис Рефала-5. Описание актуального синтаксиса Рефала-5 – в редакции Турчина – см. в [11]. Далее мы иногда не будем указывать на соответствующие незначительные отличия в этих двух синтаксисах. – *Прим. ред.*

Составной символ-метка – это последовательность букв, цифр и знаков `'-'`, `'_'`, которая начинается с большой буквы. Длина составного символа-метки не ограничивается.

Составной символ-число (макроцифра) – это последовательность цифр, число. Число записывается в обычной десятичной системе счисления. Максимальное значение числа равно $2^{32} - 1$[4].

Заметим, что хотя для записи составного символа `End` требуется три знака, для записи макроцифры `1999` – четыре знака, а для записи простого символа `'A'` – три знака, но с точки зрения языка Рефал во всех случаях мы имеем дело лишь с одним символом. Другими словами, конструкции `End`, `1999`, `'A'` составляют одно целое и не могут быть расчленены.

1.2. Последовательности символов. Последовательность символов – это несколько символов, идущих подряд. При записи последовательности простых символов в программе они разделяются пробелами:

- `'A' 'B' 'C'` – последовательность из трех символов – «A», «B», «C»;
- `1 0` – последовательность из двух макроцифр 1, 0.

Ясно, что такой способ записи последовательности простых символов является неэкономным.

Последовательность простых символов – строка символов – может быть записана короче:

- `'A+B'` – строка из трех символов – «A», «+», «B»;
- `'A''B'` – строка из трех символов – «A», апостроф, «B»;
- `''''` – строка из двух символов – апостроф, апостроф.[5]

Итак, в последовательности знаков, изображающей строку символов, знак апостроф встречается четное количество раз. Каждая пара рядом стоящих знаков апостроф, при просмотре слева направо, изображает один символ апостроф.

1.3. Объектное выражение. Объектное выражение – это выражение, состоящее из символов, а также из левых и правых структурных скобок «(» и «)». Скобки должны образовывать правильную скобочную структуру (в обычном «школьном» понимании). Выражение может быть, в частности, и пустым. Знаки «пробел», не входящие в последовательность простых символов, служат для внешнего оформления программы.

1.4. Переменные символа, выражения, терма. Переменные символа, выражения, терма служат для описания различных множеств объектных выражений.

Переменная символа записывается так: знак «`s`» и сразу за ним индекс переменной – буква, цифра, или символ «точка», за которым следует индекс

[4]Точнее, $2^N - 1$, где N равно длине *машинного слова*. – *Прим. ред.*

[5]В актуальной версии языка рефал-5 [11] эти примеры строк, включающие апострофы, будут записываться так: `'A\'B'` и `'\'\''`. – *Прим. ред.*

переменной (идентификатор или макроцифра).[6] Переменная выражения записывается так: знак «e» и сразу за ним – индекс переменной. Переменная терма записывается так: знак «t» и сразу за ним – индекс переменной.

Например, `sa`, `e1`, `e.1`, `e.String`, `s.N1`.

Правила использования индексов будут описаны ниже в п. 3.

Опишем теперь, какие значения могут принимать переменные. Переменная символа может принимать значение одного символа (любого). Переменная выражения может принимать значение любого объектного выражения. Переменная терма может принимать значение любого *терма*, т. е. либо одного символа, либо одного объектного выражения, заключенного в структурные скобки.

§ 2. Пример программы на языке Рефал

Символьные преобразования, производимые в Рефале, во многом похожи на те алгебраические преобразования, к которым мы все хорошо привыкли.

Например, вычисление выражения (в традиционной, нерефальской записи)

$$cos(cos(cos(cos(1))))$$

распадается на последовательность шагов

$$cos(cos(cos(cos(1)))) = cos(cos(cos(0.540302))) =$$
$$= cos(cos(0.857530)) = cos(0.654290) = 0.793480$$

Усложним последний пример так: пусть имеется выражение

$$cos(cos(cos(tg(1))))$$

и определение функции tg

$$tg(x) = sin(x)/cos(x)$$

Тогда вычисление выражения распадается на шаги

$$cos(cos(cos(tg(1)))) = cos(cos(cos(sin(1)/cos(1)))) =$$
$$= cos(cos(cos(0.841471/cos(1)))) = cos(cos(cos(0.841471/0.540302))) =$$
$$= cos(cos(cos(1.55741))) = cos(cos(0.0133861)) = cos(0.99991) = 0.540378$$

Последний пример во многом показателен для Рефала. Имеется выражение, которое надо вычислить, имеется определение функции tg, зависящей от переменной, обозначенной через x. При вычислении переменная x принимает значение 1, выражение $tg(1)$ заменяется внутри всего выражения на $sin(1)/cos(1)$.

Рассмотрим теперь пример программы на языке Рефал, которая делает следующее:

- программа вводит одну строку символов,
- разбивает ее на слова,
- переставляет в обратном порядке буквы в каждом слове,
- полученное печатает.

Текст этой программы изображен на рис. 1.

Сделаем некоторые пояснения. Строка

```
$EXTRN CARD, PROUT;
```

описывает внешние (стандартные) функции.

Строки, начинающиеся с символа звездочка, являются комментариями и на работу программы не влияют.

[6] В актуальной версии синтаксис языка рефал-5 требует обязательного знака точки после типа переменной (см. [11]). Т. е. `sa`, `e1` суть идентификаторы, а не переменные. – *Прим. ред.*

```
*   Программа вводит одну строку символов,
*   разбивает ее на слова,
*   переставляет в обратном порядке буквы в каждом слове,
*   полученное печатает

$EXTRN CARD , PROUT ;
$ENTRY Go {
            = <Prout <Word <Card >>>;
          }

Word {
      e.1 ' ' e.2 = <Inverse e.1> ' ' <Word e.2>;
      e.1         = <Inverse e.1>;
     }

Inverse {
         s.a e.1 = <Inverse e.1> s.a;
                 = ;
        }
```

Рис. 1. Пример программы на Рефале.

Строка

```
$ENTRY Go
```

показывает, что входной точкой данной программы является функция `Go`.

Программа состоит из трех функций: `Go`, `Word`, `Inverse`.

Описание функции заключается в фигурные скобки. Описание функции состоит из нескольких предложений, каждое из которых заканчивается символом «;» (точка с запятой).

Описание функции `Go` состоит из одного предложения, описания функций `Word`, `Inverse` – из двух предложений.

Каждое предложение имеет левую и правую части, которые отделяются друг от друга знаком «=». Левая часть предложения описывает аргумент функции, правая – значение функции.

Разберем подробно, как работает эта программа.

Работа любой Рефал-программы состоит из последовательности шагов. На каждом шаге происходит некоторое преобразование выражения, находящегося в поле зрения. Преобразование осуществляется в соответствии с описаниями функций, приведенными в программе.

Функции в поле зрения и в программе записываются при помощи знаков «<» и «>». Например, выражение

$$cos(cos(cos(cos(1))))$$

на Рефале можно записать так

```
<COS <COS <COS <COS '1'>>>>
```

Каждому знаку «<» соответствует свой знак «>», который ограничивает область действия функции. Имена функций задаются составными символами-метками, следующими сразу за знаком «<».

Шаг 1. На первом шаге поле зрения всегда имеет вид

```
<D >
```

где `D` – имя некоторой входной точки. В данном примере начальное поле зрения имеет вид

```
<Go >
```

Другими словами, требуется применить функцию `Go` к пустому выражению.

В левой части предложения для функции `Go` (перед знаком «=») стоит как раз пустое выражение. Поэтому на первом шаге работы программы выражение

```
<Go >
```

заменяется на правую часть описания функции `Go`, т. е. на выражение

```
<Prout <Word <Card >>>
```

Вспомним аналогичную замену $tg(1)$ на $sin(1)/cos(1)$.

На этом первый шаг заканчивается, Рефал-машина переходит ко второму шагу.

Шаг 2. Поле зрения имеет вид

```
<Prout <Word <Card >>>
```

Ясно, что на втором шаге возможно вычисление лишь функции `Card`. Эта функция описана в программе как внешняя и является стандартной функцией Рефала. В результате работы функции `Card` выражение

```
<Card >
```

заменяется на символы очередной введенной строки.

Предположим, что мы ввели шесть символов `'123 45'`. Тогда, после выполнения замены на втором шаге, в поле зрения получится выражение

```
<Prout <Word '123 45'>>
```

Шаг 3. На этом шаге возможно вычисление функции `Word`. Левая часть первого предложения описания функции `Word` имеет вид

```
e.1 ' ' e.2
```

Рефал-машина пытается придать переменным `e.1` и `e.2` такие значения, чтобы выражение

```
e.1 ' ' e.2
```

совпало с аргументом функции `Word`:

```
'123 45'
```

Будут использованы следующие значения переменных

`e.1` $\mapsto$ `'123'` `e.2` $\mapsto$ `'45'`

(более подробно алгоритм присваивания значений переменным изложен в п. 3).

Правая часть описания функции `Word` имеет вид

```
<Inverse e.1> ' ' <Word e.2>
```

Итак, после выполнения третьего шага поле зрения будет иметь вид

```
<Prout <Inverse '123'> ' ' <Word '45' >>
```

Шаг 4. На этом шаге происходит вычисление функции `Inverse`. Описание функции `Inverse` состоит из двух предложений. На данном шаге можно применить первое предложение при таких значениях переменных:

`s.a` $\mapsto$ `'1'` `e.1` $\mapsto$ `'23'`

В результате выражение

```
<Inverse '123'>
```

заменится на выражение

```
<Inverse '23'> '1'
```

После шага 4 поле зрения имеет вид

```
<Prout <Inverse '23'> '1 ' <Word '45'>>
```

Шаг 5. Вычисление функции `Inverse`:

```
s.a ↦ '2'     e.1 ↦ '3'
```

Поле зрения после замены имеет вид:

```
<Prout <Inverse '3'> '21 ' <Word '45'>>
```

Шаг 6. Вычисление функции `<Inverse '3'>`

```
s.a ↦ '3'     e.1 ↦ Empty
```

[7]

Поле зрения после замены имеет вид:

```
<Prout <Inverse > '321 ' <Word '45'>>
```

Шаг 7. На этом шаге первое предложение функции `Inverse` применить нельзя, поэтому Рефал-машина пытается применить второе предложение, которое описывает как раз случай пустого аргумента функции `Inverse`. Результатом применения функции `Inverse` к пустому выражению является пустое выражение.

Значит, после шага 7, поле зрения имеет вид

```
<Prout '321 ' <Word '45'>>
```

Шаг 8. Вычисление функции `Word`: первое предложение применить нельзя, так как в аргументе нет пробела, поэтому Рефал-машина пытается применить второе предложение функции `Word`:

```
e.1 ↦ '45'
```

Поле зрения после замены имеет вид:

```
<Prout '321 ' <Inverse '45'>>
```

Шаг 9. Вычисление функции `Inverse`:

```
s.a ↦ '4'     e.1 ↦ '5'
```

Поле зрения после замены имеет вид:

```
<Prout '321 ' <Inverse '5'> '4'>
```

Шаг 10. Вычисление функции `Inverse`:

```
s.a ↦ '5'     e.1 ↦ Empty
```

Поле зрения после замены имеет вид:

```
<Prout '321 ' <Inverse > '54'>
```

Шаг 11. Вычисление функции `Inverse` как на шаге 7: применяется второе предложение.

Поле зрения после замены имеет вид:

```
<Prout '321 54'>
```

Шаг 12. Вычисляется функция `Prout`, которая описана как внешняя функция. Функция `Prout` печатает выражение, к которому она применяется, результатом замены является пустое выражение.

Итак, программа напечатает

```
321 54
```

[7] Пустое выражение.

после чего поле зрения будет иметь вид пустого выражения.

Итак, поле зрения не содержит функций, которые надо вычислять, поэтому Рефал-машина останавливается в нормальном состоянии.

Задача 1. Что напечатает программа, если были введены нижеследующие символы?

- `'123 45'`
- `'123 45   '`
- `'   '`

Задача 2. Какие правильные арифметические выражения переходят в правильные при работе программы, данной на рис. 1?

§ 3. Выражения, функции, отождествление

В этом и последующих параграфах мы опишем строго те понятия, с которыми уже познакомились на примере из п. 2. Все определения будут иллюстрироваться программой, данной в этом примере (см. рис. 1).

3.1. Выражения. Выражение может быть *объектным* выражением, *типовым* выражением, *рабочим* выражением, *общим* выражением.

С понятием *объектного выражения* мы уже познакомились в п. 1. В отличие от объектного выражения, типовое выражение задает, вообще говоря, не одно конкретное выражение, а некоторое множество выражений.

Типовое выражение – это выражение, состоящее из символов, левых и правых структурных скобок и переменных одного из трех типов – символа, выражения и терма.

В частности, любое объектное выражение является также и типовым (не содержащим переменных).

Определения переменных даны в п. 2. В одном типовом выражении переменные разного типа не могут иметь одинаковых индексов, но переменные одного типа с одинаковыми индексами могут несколько раз встречаться в одном типовом выражении.

Примерами типовых выражений являются левые части предложений функций (до знака =).

Придавая переменным в типовом выражении различные значения, которые они могут принимать, мы получаем множество объектных выражений, которое задает данное типовое выражение.

Про объектное выражение, которое входит в множество, задаваемое некоторым типовым выражением, будем говорить, что оно отождествляется с этим типовым выражением.

Примеры типовых выражений:

- `s.X s.X` – типовое выражение, которое задает множество всех объектных выражений, состоящих из двух одинаковых символов;
- `'M' s.2 s.3 'A'` – типовое выражение, которое задает множество всех объектных выражений, состоящих из четырех символов, первым из которых является символ «`M`», а последним – символ «`A`». Например, `'MAMA'`, `'МАША'`, `'МИША'`;

- `(e.1) e.2` – типовое выражение, которое задает множество всех объектных выражений, состоящих из двух частей, первая из которых заключена в структурные скобки;
- `'if' ( e.1 ) 'go to ' e.2` – типовое выражение, которое задает некоторое множество условных операторов.

Рабочее выражение – это выражение, содержащее обращения к функциям – так называемые функциональные термы. Рабочее выражение может быть вычислено.

Рабочее выражение состоит из символов, левых и правых структурных скобок, а также из левых и правых функциональных скобок – знаков «<» и «>». Естественно, что структурные и функциональные скобки в совокупности должны образовывать правильную скобочную структуру.

Примером рабочего выражения является поле зрения на каждом шаге работы Рефал-программы.

Способ вычисления рабочего выражения описан ниже в п. 4. Пример вычисления рабочего выражения `<Go >` приведен в п. 2.

Общее выражение – это выражение самого общего вида. Оно может состоять из символов, структурных скобок, переменных и функциональных скобок.

Примерами общих выражений являются правые части (после знака «=») предложений функций.

3.2. Функции. Функция – это набор правил, которые задают способ вычисления функциональных термов. У каждой функции есть свое имя (детерминатив). Функция задает правила вычислений тех функциональных термов, внутренность которых начинается с составного символа-метки с именем данной функции.

Описание функции заключается в фигурные скобки и состоит из нескольких предложений.

Каждое предложение имеет вид:

- левая часть предложения – некоторое типовое выражение,
- знак «=», разделяющий левую и правую части предложения,
- правая часть предложения – некоторое общее выражение.
- знак «;» (точка с запятой).

Все переменные, входящие в правую часть предложения, должны входить и в левую часть предложения.

Теперь определим точно способ вычисления функционального терма, который начинается с составного символа-метки с именем функции, заданной некоторыми предложениями. Вычисление происходит следующим образом:

1. Начинаем с первого предложения.
2. Проверяем, отождествляется ли внутренность функционального терма (не включая составного символа-метки с именем функции, а также без знаков «<» и «>») с левой частью очередного предложения.
3. Если отождествляется, то в этом случае все переменные левой части предложения принимают некоторые значения. Эти значения подставляются в правую часть предложения, после чего получается некоторое

рабочее выражение. Это рабочее выражение заменяет исходный функциональный терм (включая знаки «<» и «>»). На этом вычисление функционального терма заканчивается.

4. Если не отождествляется, то берется следующее предложение и повторяется проверка.
5. Если предложений больше нет, то вычисление данного функционального терма невозможно. Такая ситуация называется *«отождествление невозможно»*.

В вышеприведенном описании процесса вычислений может возникнуть неоднозначность присвоения значений переменным из левой части предложения. Интуитивно, при отождествлении выбирается «первый подходящий вариант отождествления при просмотре слева направо».

Для придания точного смысла интуитивному понятию отождествления, упорядочим все переменные в том порядке, в котором они встречаются в типовом выражении при просмотре слева направо. Теперь из всех вариантов отождествления выберем те, у которых значение первой переменной самое короткое. Если получилось несколько вариантов, то оставим те, у которых значение второй переменной самое короткое, и так далее. Подобная процедура всегда приводит к однозначности.

Выбор подходящего варианта отождествления происходит не просто перебором. Программа предварительно обрабатывается компилятором, основная работа которого и состоит в том, чтобы определить наиболее быстрый для каждой левой части предложения способ отождествления с ней объектных выражений.

§ 4. Выполнение Рефал-программы

Рефал-программа состоит из набора функций. Работа Рефал-программы заключается в вычислении некоторого рабочего выражения (т. е. набора вызовов функций). Это рабочее выражение называется полем зрения. Начальное поле зрения имеет специальный вид и описано в п. 7.

Вычисление рабочего выражения в поле зрения состоит из последовательности шагов. На каждом шаге выполняются следующие действия:

- если в поле зрения нет ни одного функционального терма (знаков «<» и «>»), то происходит нормальная остановка программы;
- если в поле зрения есть функциональные термы, то из них выбирается самый левый терм, который не содержит внутри себя других функциональных термов (он называется ведущим термом для этого шага), после чего выбранный терм вычисляется (способ вычисления описан в п. 3);
- на этом выполнение одного шага заканчивается.

При вычислении одного функционального терма возможны три исхода.

1. Остановка «отождествление невозможно» (см. п. 3).
2. Произошло отождествление с одной из левых частей предложений, описывающих функцию, но для формирования результата замены не хватает свободной памяти компьютера. В этом случае происходит остановка «свободная память исчерпана».

3. Произошло отождествление и удалось сформировать результат замены. Результат замены замещает ведущий терм, и работа продолжается.

Задача 3. Привести пример программы, которая зацикливается, т. е. продолжает вычисления до бесконечности и не останавливается.

Рассмотрим несколько несложных примеров функций.

Пример 1. Кузнечик прыгает по прямой линии то вправо, то влево на 1 метр (или по несколько раз влево или вправо). Прыжки записаны в виде последовательности букв `'L'` (влево) или `'R'` (вправо). Определить, где будет кузнечик в конце концов.

Можно просто сосчитать количество прыжков влево и количество прыжков вправо и вычесть одно из другого. Мы еще не умеем пользоваться арифметикой, поэтому попробуем воспользоваться тем, что два различных прыжка сокращаются (т. е. прыгнуть сначала вправо, а затем влево – это оставаться на месте).

При программировании на Рефале во многих случаях оказывается удобным следовать словесному описанию алгоритма работы. Например, можно действовать так: «если в последовательности прыжков есть рядом стоящие символы `'LR'` или `'RL'`, то убрать эту пару символов и повторить все сначала, в противном случае закончить работу».

Буквальное повторение этой инструкции на Рефале выглядит так (назовем нашу функцию `Step`):

```
Step {
     e.1 'LR' e.2 = <Step e.1 e.2>;
     e.1 'RL' e.2 = <Step e.1 e.2>;
     e.1          = e.1;
     }
```

Первое предложение соответствует случаю «если есть символы 'LR'», второе – случаю «если есть символы `'RL'`» , третье – «если нет ни того, ни другого».

Отсутствие в правой части первого предложения выражения `'LR'` означает «убрать символы `'LR'`». Аналогично во втором предложении.

Обращение в правой части к функции `Step` означает «и повторить все сначала».

Отсутствие в правой части третьего предложения обращения к функции `Step` означает конец вычислений.

Заметим, что порядок предложений существенен. Например, если третье предложение поставить в начало, то первое и второе предложения не будут применяться никогда.

В данном примере мы видим, что при замене левой части на правую функция `Step` вызывает сама себя. Таким образом, описание функции `Step` оказалось рекурсивным.

Перейдем теперь к недостаткам этого решения задачи. Рассмотрение недостатков поможет нам понять тонкости выполнения Рефал-программы.

Заметим сначала, что человек вряд ли бы стал следовать вышеприведенному алгоритму. Действительно, после того, как символы `'LR'` уже найдены,

то остальные пары 'LR', если они есть, могут находиться только правее, и просматривать повторно уже просмотренную часть выражения бессмысленно.

Итак, недостатком алгоритма являются многократные просмотры последовательности символов.

Как можно избавиться от этого недостатка? Поскольку просмотренная часть выражения может еще понадобиться, то мы не можем ничего выносить за функциональные скобки (как мы делали для функции Go в п. 2.) Поэтому надо как-то отделять просмотренную часть выражения от еще не просмотренной.

Сделаем пояснения насчет структурных скобок. При представлении в памяти компьютера поля зрения структурные скобки представляются так, что по левой скобке сразу можно найти правую и наоборот, не просматривая выражения, заключенного в скобки. Например, если типовое выражение имеет вид

(e.1) e.2

то присвоение значений переменным e.1 и e.2 будет происходить очень быстро.

Таким образом, целесообразно для данного примера поместить уже просмотренную часть выражения в структурные скобки. Подобный прием очень часто используется при программировании на Рефале, а вспомогательные скобки называются *«сумкой»*.

Решение с использованием «сумки» приведено ниже .

```
Step  { e.1 = <Step1 ( ) e.1>; }
Step1 {
        (       ) s.a e.2 = <Step1 (s.a        ) e.2>;
        (e.1 s.a) s.a e.2 = <Step1 (e.1 s.a s.a) e.2>;
        (e.1 s.b) s.a e.2 = <Step1 (e.1        ) e.2>;
        (e.1    )         = e.1 ;
      }
```

Первое предложение функции Step1 описывает случаи начала работы, а также те моменты времени, когда кузнечик оказывается на месте старта. Второе предложение рассматривает случай двух одинаковых прыжков. В третьем предложении прыжки разные, поэтому их можно сократить. Последнее предложение заканчивает работу.

Замечание 1. В описании рекурсивной функции всегда должно быть предложение, которое осуществляет «выход из рекурсии», иначе функция либо зациклится, либо произойдет «отождествление невозможно».

Пример 2. Написать функцию Symm, которая проверяет выражение, состоящее из символов, на симметричность.[8]

Прежде чем писать предложения функции Symm, надо договориться, что является результатом работы этой функции.

Возможны лишь два результата – «ДА» (симметричное выражение) или «НЕТ» (несимметричное), т. е. функция Symm является предикатом. В математической логике принято обозначать истину символом «1», ложь – символом «0». Поэтому опишем функцию Symm так, чтобы результатом замены являлся

[8]Такое выражение называется палиндро́мом (от др.-греч. $\pi\alpha\lambda\iota\nu$ — «назад, снова» и $\delta\rho\circ\mu\circ\varsigma$ — «бег, движение»). – *Прим. ред.*

символ «1», если выражение симметричное, и символ «0», если несимметричное.

Можно предложить такое решение примера:

```
Symm {
     s.a         = '1';
     s.a s.a     = '1';
     s.a e.1 s.a = <Symm e.1>;
     e.1         = '0';
     }
```

Другими словами, выражение из одного символа является симметричным (предложение 1), из двух одинаковых символов – также симметричным (предложение 2). Если выражение начинается и кончается одинаковыми символами, то для решения вопроса о симметричности надо проверить на симметричность выражение без этих символов (предложение 3). Во всех остальных случаях выражение не является симметричным (предложение 4).

Задача 4. В предыдущем примере мы предполагали, что последовательность символов не может быть пустой. Будем считать, что пустое выражение является симметричным. Изменить описание функции `Symm`, чтобы обрабатывалось пустое выражение. Как решить задачу, не увеличивая число предложений?

Задача 5. Мы предполагали, что функция `Symm` обрабатывает строки символов. Изменить описание функции `Symm` так, чтобы она обрабатывала произвольное объектное выражение (т. е. выражение со скобками).

Пример 3. Прибавление единицы к числу, записанному в двоичной системе счисления.

Назовем эту функцию `Plus1`[9]

```
Plus1 {
     e.1 '0' = e.1 '1';
     e.1 '1' = <Plus1 e.1> '0';
             = '1';
      }
```

Приведенное решение моделирует «сложение столбиком»: если число оканчивается на 0, то заменяем этот 0 на 1, если же число оканчивается на 1, то «0 пишем, 1 в уме», т. е. нам надо прибавить единицу к более короткому числу. Последнее предложение понадобится в том случае, когда исходное число состоит только из единиц.

Задача 6. Написать функцию, прибавляющую единицу к числу, записанному в обычной десятичной системе счисления.

[9]Программа `Plus1` описывает частично рекурсивную функцию, область определения которой строго включает в себя множество двоичных строк $\mathbb{B}$. Т. е. функция прибавления единицы к числу, записанному в двоичной системе счисления, есть ограничение `Plus1` на $\mathbb{B}$, а не сама `Plus1`.

Аналогичное замечание относится и к программе `Alfa` из примера 4. – *Прим. ред.*

Задача 7. Сложить два числа в двоичной системе счисления. Обозначим функцию через Add2. Предположим, что формат обращения к функции Add2 следующий

```
<Add2 (e.N1) e.N2>
```

где e.N1, e.N2 – слагаемые. Другими словами, первое слагаемое заключено в скобки, второе – вне скобок.

Пример 4. Пусть имеются три символа, каждый из которых либо 0, либо 1. Написать функцию, результатом работы которой является один символ: 0, если в исходной последовательности нулей больше чем единиц, или 1 – в противном случае. Другими словами, функция определяет большинство.

Существует много способов написания такой функции. Самый простой способ состоит в реализации наблюдения, что искомый символ должен встречаться по крайней мере два раза:

```
Alfa {
    e.1 s.a e.2 s.a e.3 = s.a ;
    }
```

Заметим, что функция Alfa, описанная в таком виде, будет успешно работать и в том случае, если вместо нулей и единиц подавать на вход, например, символы «A» и «B».

Задача 8. Пусть имеются три символа, каждый из которых либо 0, либо 1. Написать функцию, результатом работы которой являются два символа, означающие количество единиц в исходной последовательности (в двоичной системе счисления). Например, для последовательности «111» должно получиться «11», для «100» – «01», для «000» – «00».

§ 5. Структурные скобки. Дифференцирование

В этом параграфе мы рассмотрим несколько более сложных примеров. Первый пример связан с превращением символов «(» и «)» в структурные скобки, так называемое «спаривание скобок». Во втором примере мы рассматриваем вопрос о дифференцировании на языке Рефал произвольных функций математического анализа.

5.1. Структурные скобки. Рассмотрим следующие объектные выражения:

```
'(a+b)*(c+d)'
('a+b') '*' ('c+d')
```

Они соответствуют одной и той же формуле из алгебры – $(a+b)(c+d)$, но способы записи этой формулы совершенно различные. В первом случае мы записали эту формулу как последовательность из 11 простых символов: «(», «a», «+»,..., «d», «)». Во втором случае мы использовали две пары структурных скобок. В первой скобке записаны три символа «a», «+», «b», во второй – три символа «c», «+», «d», между скобками стоит один символ «*».

Ниже мы будем вместо мета-обозначений «, » для символов использовать синтаксис рефала. Т. е., например, писать 'd', а не «d».

Второе объектное выражение отождествляется с типовым выражением

```
( e.1 ) s.a ( e.2 )
```

но первое объектное выражение с этим типовым выражением не отождествится.

Функция `Card` (см. пп. 2, 6) помещает в поле зрения символы очередной введенной строки. Если мы наберем первое выражение, то после вычисления функции `<Card >` в поле зрения будем иметь объектное выражение, состоящее из 11 символов. Второе выражение вообще невозможно подготовить для функции `Card`.

При обработке же алгебраических формул на языке Рефал удобно использовать структурные скобки.

Таким образом, мы приходим к задаче превращения символов `'('`, `')'` в структурные скобки.

Отметим, что структурные скобки всегда присутствуют парами – каждой левой структурной скобке соответствует правая, и наоборот. Символ `'('` равноправен среди остальных простых символов, поэтому в объектных выражениях он может присутствовать и без символа `')'`.

Если мы будем просматривать последовательность символов слева направо, то нам будут по отдельности попадаться то символы `'('`, то символы `')'`. Для того, чтобы вставить пару структурных скобок, мы должны для каждого символа «левая скобка» найти соответствующий ему символ «правая скобка» и одновременно заменить их на структурные скобки. Такой процесс описывается предложением

```
e.1 '(' e.2 ')' e.3 = e.1 ( e.2 ) e.3
```

Обратимся к примеру. Пусть скобочная структура имеет вид:

```
(  (  )  (  )  )  (  )
```

Ясно, что первая правая скобка соответствует последней левой скобке, лежащей левее этой правой скобки. После замены найденных скобок-символов на структурные скобки опять ищем первый символ `')'` и т. д.

Поэтому можно было бы предложить такой алгоритм: двигаемся вправо до символа `')'`, затем влево до символа `'('`, превращаем найденные символы `'('` и `')'` в структурные скобки. Далее повторяем эту процедуру, пока не кончатся символы `'('`, `')'`.

Приведенный алгоритм обладает тем недостатком, что требуются многократные просмотры всего выражения. Оказывается, что задачу спаривания скобок можно решить за один просмотр последовательности символов!

Ведь двигаясь вправо до первого символа `')'`, мы уже находим все символы `'('`, лежащие на пути. Надо их только не потерять, т. е. как-то отметить, чтобы сразу можно было найти, когда они понадобятся.

Как можно отделить одну часть последовательности от другой? Опять на помощь приходит понятие «сумки» из п. 4. Будем каждый символ `'('` отмечать сумкой. Поскольку левых скобок может встретиться несколько, то и сумок потребуется несколько.

Обозначим функцию спаривания скобок через `Spar`. Тогда функция `Spar` записывается следующими предложениями (использование функции можно посмотреть в программе п. 14.2).

```
Spar  {  e.1 = <Spar1 ('*') e.1 >; }

Spar1  {
      (e.1)       '(' e.3 = <Spar1 ((e.1))     e.3>;
      ((e.1) e.2) ')' e.3 = <Spar1 (e.1 (e.2)) e.3>;
      ('*'e.1)    ')' e.3 = 'error' e.1 ')'   e.3;
      (e.1)       s.A e.3 = <Spar1 (e.1 s.A)   e.3>;
      ('*'e.1)            = e.1;
      ((e.1) e.2)         = 'error' e.1 '(' e.2;
       }
```

Вопросы к программе Spar:

1. Почему при обращении к функции Spar1 в сумку поставлен символ '*'?
2. Какое предложение обнаруживает случай непарной левой скобки?
3. Какое предложение обнаруживает случай непарной правой скобки?
4. Какое предложение заканчивает вычисление функции, если все скобки сбалансированы?
5. Написать работу функции Spar по шагам для первого объектного выражения, данного в начале параграфа.

Задача 9. Если нам надо в последовательности символов лишь убедиться в правильности расстановки символов '(' и ')', то существует более простой алгоритм. Скобки расставлены правильно, если (а) количество левых скобок равно количеству правых, и (б) при просмотре слева направо в каждый момент количество правых скобок не больше количества левых скобок.

Написать функцию, которая применяется к последовательности символов, и результат работы которой равен «1», если скобки расставлены правильно, и равен «0» в противном случае.

Задача 10. Написать функцию расспаривания скобок, т. е. превращения структурных скобок в символы '(' и ')'.[10]

5.2. Дифференцирование. Задача дифференцирования функций математического анализа относится к числу символьных преобразований формул, поэтому язык Рефал очень хорошо справляется с этой задачей. Предположим, что формат обращения к функции дифференцирования следующий:

```
<Diff (e.t) e.Function>
```

где e.t – имя переменной, по которой происходит дифференцирование, e.Function – дифференцируемое выражение.

Сформулируем ограничения, которым должна удовлетворять формула e.Function.

1. Формула записывается по правилам записи арифметических выражений алгоритмических языков, т. е. операции сложение, вычитание, умножение, деление, возведение в степень обозначаются через '+', '-', '*', '/', '^'.
2. Показатель степени не должен зависеть от переменной e.t.

[10] Ответ можно посмотреть в программе п. 14.2.

3. Ограничимся использованием лишь следующих функций: *sin*, *cos*, *exp*; введение остальных функций достигается добавлением по одному предложению на каждую новую функцию.
4. Функция `Diff` не будет делать никаких алгебраических упрощений полученной формулы (приведение подобных и т. д.), потому что упрощение формулы – задача гораздо более сложная, чем дифференцирование.
5. Все скобки являются структурными, если это не так, то можно воспользоваться функцией `Spar`.

```
Diff  {
     (e.t) e.1 '+' e.2 = <Diff (e.t) e.1> '+' <Diff (e.t) e.2>;
     (e.t) e.1 '-' e.2 = <Diff (e.t) e.1> '-' <Diff (e.t) e.2>;
     (e.t) e.1 '*' e.2 = (e.1 '*' <Diff (e.t) e.2>
                          '+' <Diff (e.t) e.1> '*' e.2);
     (e.t) e.1 '/' e.2 = (( <Diff (e.t) e.1> '*' e.2 '-'
                              e.1 '*' <Diff (e.t) e.2>) '/'
                              (e.2 '*' e.2) );
     (e.t) e.1 '^' e.2 = (e.1 '^' (e.2 '-1' )) '*' <Diff (e.t) e.1>;
     (e.t) 'sin'(e.1) = 'cos'(e.1) '*' (<Diff (e.t) e.1>);
     (e.t) 'cos'(e.1) = ('-sin'(e.1)) '*' (<Diff (e.t) e.1>);
     (e.t) 'exp'(e.1) = 'exp'(e.1) '*' (<Diff (e.t) e.1>);
     (e.t) (e.1)      = ( <Diff (e.t) e.1> );
     (e.t) e.t        = '1';
     (e.t) e.1        = '0';
      }
```

Отметим, что почти все предложения являются перезаписью известных правил дифференцирования (суммы, произведения, частного, сложной функции и т. д.).

Поскольку лишние скобки не влияют на процесс вычисления по формуле, то мы не следим за появлением «лишних» скобок.

Почему именно такой порядок предложений?

Задача 11. Как правило, не удается написать сразу программу без ошибок. При написании функции `Diff` была допущена ошибка. Мы ее и оставили. Цель данного упражнения состоит в нахождении оставленной ошибки. Какие формулы дифференцируются правильно функцией `Diff`? Для каких формул производная будет вычисляться неправильно?

Задача 12. Расширьте возможности функции `Diff` так, чтобы она обрабатывала все функции библиотеки подпрограмм, а также возведение в степень, являющуюся функцией от переменной.

Задача 13. Проведите анализ алгоритма `Diff` на эффективность. В каких местах происходит многократный просмотр выражения?

Задача 14. Два последних предложения функции `Diff` ясно показывают, что в получающейся формуле будет много лишних нулей и единиц. Напишите программу упрощения алгебраических выражений (в разумных пределах).

§ 6. Стандартные функции

Стандартными функциями называются такие функции, которыми можно пользоваться, не определяя их в Рефал-программах.

С двумя стандартными функциями `Card`, `Prout` мы уже познакомились в п. 2.

Имеются следующие стандартные функции:

- ввод и печать,
- арифметические функции,
- операции с копилкой,
- функции лексического анализа,
- файловый ввод/вывод.

В настоящих лекциях мы познакомимся лишь с первыми тремя типами стандартных функций. Полный список стандартных функций рефала-5 приведен в [6].

Будем использовать следующее соглашение: объектные выражения в форматах обращений к функциям обозначаем через `e.Expr`, `e.N1`, `e.N2`, `e.File-name`, `s.D` и т. д.

Если аргумент функции не соответствует формату обращения к функции, то происходит остановка «отождествление невозможно».

6.1. Ввод. Формат обращения к функции:

```
<Card >
```

Результатом вычисления является введенная строка.

Еще раз подчеркнем, что ни структурных скобок, ни составных символов в поле зрения с помощью функции `Card`[11] получить нельзя.

6.2. Вывод результата. Печать результатов работы Рефал-программы можно осуществлять при помощи функции `Print`.

Формат обращения к функции:

```
<Print e.Expr>
```

где `e.Expr` – объектное выражение, которое нужно напечатать.

В результате выполнения функции `Print` выражение `e.Expr` печатается с новой строки. Если выражение `e.Expr` не помещается на одной строке, то оно продолжается на следующих строчках.

Результат замены[12] – выражение `e.Expr`.

Простые символы `'('`, `')'` и структурные скобки при выводе не различаются.

Формат обращения к функции `Prout`:

```
<Prout e.Expr>
```

Результат замены – пустое выражение. Выражение `e.Expr` выводится в том же виде, как для функции `Print`.

---

[11] Название функции происходит от «ввод *карт*». Современные программисты не знакомы с таким механизмом ввода данных. – *Прим. ред.*

[12] Т. е. возвращаемое значение – выражение `e.Expr`. Здесь и ниже по тексту автор использует исторически сложившуюся терминологию языка Рефал. – *Прим. ред.*

6.3. Операции с копилкой. Кроме поля зрения, в Рефал-машине имеется еще одно выражение, которое называется копилкой.

Копилкой удобно пользоваться, когда некоторое выражение требуется в различных частях программы, а передавать это выражение через поле зрения неудобно.

Каждое запомненное в копилке выражение имеет свое имя. Запись выражений в копилку и чтение их оттуда осуществляется по именам выражений. Более того, под одним именем можно записать несколько значений по очереди, а затем по очереди их читать.

Копилка имеет вид:

```
(e.a '=' e.1) (e.b '=' e.2) ...
```

где `e.a`, `e.b`,... – имена закопанных выражений, `e.1`, `e.2`,... – значения, которые закопаны под именами `e.a`, `e.b`,... соответственно.

Опишем 4 стандартные функции работы с копилкой:

- `Br` – закопать (Bury),
- `Dg` – выкопать (Dig),
- `Cp` – скопировать (Copy),
- `Rp` – заменить (Replace).

6.3.1. *`Br` – закопать.* Формат обращения:

```
<Br e.N '=' e.0>
```

где `e.N` – произвольное выражение, не содержащее символа `'='` на внешнем уровне скобочной структуры, `e.0` – произвольное выражение. Копилка `e.K` преобразуется так:

$$\texttt{e.K} \mapsto \texttt{(e.N' =' e.0) e.K}$$

т. е. терм `( e.N '=' e.0 )` добавляется к копилке слева.

Результатом вычисления будет пустое выражение.

6.3.2. *`Dg` – выкопать.* Формат обращения:

```
<Dg e.N>
```

где `e.N` – произвольное выражение.

Функция `Dg` просматривает копилку слева направо в поисках терма

```
( e.N ' =' e.0 )
```

и, если его находит, удаляет его из копилки и выдает `e.0` в качестве результата замены.

При этом копилка меняется так:

$$\texttt{e.1 (e.N '=' e.0) e.2} \mapsto \texttt{e.1 e.2}$$

Если же терм не найден, то результатом замены будет пустое выражение, копилка не изменяется.

Отметим, что если в копилке несколько выражений закопаны под одним именем, то при первом обращении к функции `Dg` выкапывается выражение, закопанное последним, при втором – предпоследним и т. д.

6.3.3. *`Cp` – скопировать.* Формат обращения:

```
<Cp e.N>
```

где `e.N` – произвольное выражение.

Функция Cp работает аналогично функции Dg: отличие заключается в том, что найденный терм (e.N '=' e.0) из копилки не удаляется, а в поле зрения формируется копия выражения e.0.

Отметим, что функция Cp работает медленнее функции Dg, так как нужно формировать копии выражений. Функции Br и Dg работают очень быстро, поскольку выражение e.0 фактически не переписывается из поля зрения в копилку или обратно, изменяются лишь адреса, которые связывают выражения друг с другом.

6.3.4. *Rp – заменить.* Формат обращения:

```
<Rp e.N '=' e.0>
```

где e.N, e.0 – те же, что и для функции Br.

Функция Rp добавляет в копилку новое выражение и выбрасывает выражение, закопанное под именем e.N в последний раз (если такое выражение есть).

Результат замены – пустое выражение.

Копилка изменяется так:

```
e.1 (e.N '=' e.P) e.2 ↦ e.1 (e.N '=' e.0) e.2
```

Задача 15. Можно ли функцию Rp описать на Рефале следующими предложениями?

```
Rp { e.1 '=' e.2 = <Del <Dg e.1>> <Br e.1 '=' e.2> ; }

Del { e.1 = ; }
```

6.4. Арифметические функции. Стандартные функции Add, Sub, Mul, Div, Divmod, Mod предназначены для работы с целыми числами, записанными с помощью макроцифр (см. п. 1).

- Add – сложение,
- Sub – вычитание,
- Mul – умножение,
- Div – деление, Divmod – деление с остатком, Mod – получение остатка.

Целое число – это непустая последовательность макроцифр, перед которой может стоять знак. Знак – это простой символ '+' или '-'. Если знак отсутствует, то подразумевается '+'.

Такое представление числа является записью в системе счисления по основанию 2^N, где N – количество двоичных разрядов, отводимых под одну макроцифру. Для Рефала-5 $N = 32$.[13]

Например, четыре символа '-' 1 12 3 изображают число

$$-(1 \times 2^{64} + 12 \times 2^{32} + 3)$$

Обращение к функциям Add, Sub, Mul, Div, Divmod, Mod имеет вид

```
<D (e.M) e.N>
```

где D – имя одной из функций, e.M, e.N – целые числа.[14]

Результат вычисления функций Add, Sub, Mul, Div, Divmod, Mod – целое число, являющееся суммой, разностью, произведением, частным или остатком от деления целых чисел e.M и e.N.

[13] Точнее, N равно длине *машинного слова*. – *Прим. ред.*

[14] Структурные скобки над первым аргументом могут быть опущены, если e.M принимает значение макроцифры, возможно со знаком. – *Прим. ред.*

Если результат положителен, то знак '+' не ставится, левые нули опускаются. Нулевой результат выдается в виде одной макроцифры 0.

Пример 5. Функция, вычисляющая факториал неотрицательного числа:

```
Fact {
      0  = 1;
      e.1 = <Mul (e.1) <Fact <Sub (e.1) 1>>> ;
     }
```

Напишите «по шагам» вычисление рабочего выражения `<Fact 4>`. Обратите внимание на порядок вычислений умножений.

Функция `Div` выдает результат в виде

`(e.Q) e.R`

где `e.Q` – частное, `e.R` – остаток.

Частное и остаток выдаются без левых нулей и без знака '+' перед положительными числами. Деление на нуль приводит к остановке «отождествление невозможно».

Деление с остатком производится так: делятся два числа, без учета знаков, а затем частному и остатку приписываются знаки так, чтобы выполнялось соотношение:

$$\texttt{e.M} = \texttt{e.Q} \times \texttt{e.N} + \texttt{e.R}$$

Например,

```
<Divmod  (     5 )     3>  ↦ (     1 )     2
<Divmod  (     5 ) '-' 3>  ↦ ( '-' 1 )     2
<Divmod  ( '-' 5 )     3>  ↦ ( '-' 1 ) '-' 2
<Divmod  ( '-' 5 ) '-' 3>  ↦ (     1 ) '-' 2
```

Пример 6. Приведем описание функции `Nod`, вычисляющей наибольший общий делитель двух чисел по алгоритму Евклида. Формат обращения: `<Nod (e.1) e.2>`.

```
Nod  {
     (e.1) 0   = e.1;
     (e.1) e.2 = <Nod (e.2) <Mod (e.1) e.2>>;
     }
```

Задача 16. Написать работу «по шагам» функции `Nod` для каких-либо небольших целых чисел.

Задача 17. Известная теорема из алгебры (см., например, [3; п. 1.8.3]) утверждает, что наибольший общий делитель d двух чисел a и b всегда выражается в виде

$$d = u \times a + v \times b$$

для некоторых целых чисел u, v.

Числа u, v можно определить, используя соотношения, которые возникают при работе алгоритма Евклида.

Написать функцию, которая для данных целых чисел a, b вычисляет числа u, v.

6.5. Преобразование цифр в макроцифру – `Numb`. Все исходные данные поступают в поле зрения в виде цепочек простых символов. Поэтому числовые данные – последовательности цифр – надо преобразовать в последовательность макроцифр. Для этого служит стандартная функция `Numb`.

Формат обращения:

```
<Numb e.D>
```

где `e.D` – последовательность цифр, перед которой может быть символ `'+'` или `'-'`. Число представляется в десятичной системе счисления и должно быть меньше 2^{32}.[15] Результатом вычисления является одна макроцифра, перед которой стоит знак `'-'`, если исходное число было отрицательным.

Для перевода произвольной последовательности цифр без знака в последовательность макроцифр можно воспользоваться функцией `CVB`, которая приведена в программе п. 14.1.

6.6. Преобразование макроцифр в цифры – `Symb`. Результаты надо печатать в виде последовательности цифр, поэтому существует стандартная функция `Symb`, которая превращает одну макроцифру в последовательность десятичных цифр.

Формат обращения:

```
<Symb e.N>
```

где `e.N` – одна макроцифра.[16]

Результатом вычисления будет число `e.N`, записанное в десятичной системе счисления в виде последовательности простых символов.

Для перевода произвольной последовательности макроцифр без знака в последовательность десятичных цифр можно воспользоваться функцией `CVD`, которая приведена в программе п. 14.1.

Пример 7. Основным преимуществом арифметики языка Рефал является неограниченность обрабатываемых чисел. Рассмотрим пример использования арифметики Рефала.

Вычислить число $e = 2.71828\ldots$ с точностью до 2000 знаков после запятой.

Для вычисления числа используем ряд

$$e = 1 + 1 + 1/2! + 1/3! + 1/4! + \ldots$$

Для решения задачи достаточно взять первые 1000 слагаемых ряда. Будем вычислять частичные суммы ряда в виде дробей.

Обозначим требуемую функцию через `E2000`. Решение приведено в программе п. 14.1.

Обращение к функции: `<E2000 >`.

Результат замены – число с точностью до 2000 знаков.

Сделаем пояснения.

---

[15] В языке Рефал-5 [11] такого ограничения на аргумент функции `Numb` не накладывается. Эта функция переводит число из десятичного символьного представления в систему счисления по основанию 2^N, где N – длина машинного слова; т. е. в последовательность макроцифр, перед которой может быть знак. Макроцифры суть цифры в новой системе счисления. Функция `Symb` является обратной к `Numb`. – *Прим. ред.*

[16] См. предыдущую сноску. – *Прим. ред.*

Функция E2000 делает вспомогательный шаг и подготавливает структуру поля зрения для вычислений: первый символ – макроцифра обозначает номер шага $n+1$ при вычислении частичной суммы ряда Σ_n.

Функция E2001 осуществляет вычисление $n! \times \Sigma_n$, 100^n при $n = 1, 2, \ldots, 1000$.

Первое предложение проверяет окончание цикла.

Функция CVD переводит полученные макроцифры в последовательность из 2000 десятичных цифр, а функция E2003 ставит после первого символа десятичную точку (в числе точка стоит после первой цифры).

Задача 18. Написать функцию E вычисления числа e с заданным (произвольным) количеством знаков.

Формат обращения:

```
<E e.N>
```

где e.N – макроцифра, указывающая точность вычислений.

Задача 19. Написать функцию вычисления числа $\pi = 3.14\ldots$.

§ 7. Модули Рефал-программ

Программа на Рефале может состоять из одного или нескольких модулей. Функции, которые будут многократно использоваться, удобно оформлять отдельными модулями. Все стандартные функции оформлены как отдельные модули.

Для описания входных точек модуля используется служебное объявление $ENTRY, для описания имен внешних функций – служебное объявление $EXTRN.

Имена, описанные посредством служебных объявлений $ENTRY, $EXTRN, становятся глобальными, т. е. ими можно пользоваться в других модулях. Поэтому на имена входных и внешних функций накладываются ограничения, которые требует операционная система.

Все имена функций, которые используются в модуле и которые не описаны при помощи объявления $EXTRN, должны быть именами функций, описанных в этом модуле.

Если имя D описано в каком-то модуле при помощи объявления $ENTRY, то вычисление терма

```
<D e.1>
```

в любом другом модуле, в котором D описана как внешняя функция, вызовет обращение к первому модулю.

Таким образом, объявления $ENTRY, $EXTRN организуют межмодульные связи.

Любая Рефал-программа начинает свою работу с вычисления поля зрения следующего вида

```
<D >
```

где D должно быть описано в служебном объявлении $ENTRY некоторого модуля.

Способ задания имени D зависит от реализации. Имя D задается в качестве параметра задания на выполнение Рефал-программы. Во всех примерах у нас используется имя Go.

Одним из способов передачи информации от модуля к модулю является использование копилки. Копилка для всех модулей одна! Поэтому все, что закапывалось в одном из модулей программы, может быть выкопано в другом.

§ 8. Пример порождения программы

Исходный текст любой программы на любом языке программирования является последовательностью символов. Язык Рефал предназначен для обработки символьных данных произвольной структуры. Поэтому на Рефале удобно обрабатывать тексты программ на любых языках, в том числе тексты Рефал-программ.

В частности, можно строить, порождать тексты программ. Вид порождаемых программ зависит от каких-то исходных данных. Часто на этом пути можно добиться большей эффективности выполнения программ.

В этом параграфе мы рассмотрим пример задачи, которая легко решается методом порождения программ и которую было бы затруднительно решить обычным программированием на известных языках.

Пример 8. Исходными данными является формула в символьном виде F, зависящая, скажем, от переменной t. Требуется вычислить в некоторых точках $t_1, t_2, \ldots, t_n$ значения формулы F, а также значения ее производной.

Ясно, что на языках программирования типа Basic, Pascal, Fortran очень трудно решить эту задачу, поскольку затруднительно вычислять производную.

На Рефале очень легко продифференцировать функцию, но зато в Рефале нет средств для вычислений с плавающей точкой. Даже при наличии таких средств пришлось бы анализировать формулу, организовывать вычисления в режиме интерпретации, что привело бы к снижению эффективности.

Поэтому предлагается следующий способ решения задачи.

Программа на Рефале, назовем ее `gener.ref` (см. п. 14.2), вводит формулу F, затем дифференцирует ее и, наконец, порождает текст программы `GN.BAS` на языке Basic, в которую вставлены формула F и формула производной. Basic-программа, которая получилась, транслируется и выполняется. Ниже приведен вид программы `GN` для функции $F(t) = sin(t) \times cos(t)$.

```
10 t = 0.0
20 FOR I=1 TO 100
30 t = t + 0.01
40 F = sin(t)*cos(t)
50 FDt = (cos(t)*(1))*cos(t) + sin(t)*(-sin(t))*(1)
60 PRINT "t="; t, "F="; F, "FD="; FDt
70 NEXT I
80 END
```

Мы предполагаем, что функция и ее производная будут вычисляться в точках $0.01, 0.02, \ldots, 0.99, 1.00$. От этого предположения можно легко избавиться за счет непринципиального удлинения текста программы.

Предположим, что формула F записывается в нескольких строчках, признаком конца формулы пусть будет символ «&», стоящий в конце формулы. Все пробелы должны игнорироваться.

§ 9. Эквивалентные преобразования алгоритмов на Рефале

Синтаксис языка Рефал очень просто устроен. Поэтому можно преобразовывать описания функций языка Рефал, так же как мы производим алгебраические преобразования.

В этом параграфе мы решим две задачи, которые были сформулированы ранее. С помощью довольно очевидных преобразований мы получим такие описания функций, которые можно назвать красивыми, и до которых так просто додуматься было бы нелегко.

Сразу же сделаем пояснение. Цель данного параграфа состоит не в том, чтобы научить кого-либо преобразованиям алгоритмов, а в том, чтобы показать на ярких примерах небольшой кусочек той работы, которая ведется в этой области. Более того, те преобразования, которые мы рассмотрим, должен делать компилятор с языка Рефал, а не программист. Компилятор должен не только транслировать программу на машинный язык, но и получать при этом эффективную программу.

Программист должен без ошибок описать на Рефале алгоритм, а забота об эффективности выполнения ложится на компилятор. Многое уже сделано в этой области, многое предстоит сделать.

Пример 9. Написать функцию, которая по трем символам из множества «0», «1» выдает двузначное число, обозначающее количество единиц в исходной последовательности.[17]

Решение 1. Обозначим искомую функцию через `Alfa` и опишем ее 8-мью предложениями, которые соответствуют всевозможным наборам «0» и «1».

```
Alfa {
     '111' = '11';
     '110' = '10';
     '101' = '10';
     '100' = '01';
     '011' = '10';
     '010' = '01';
     '001' = '01';
     '000' = '00';
     }
```

Первое и последнее предложения можно объединить, записав их так:

```
          s.a s.a s.a = s.a s.a;
```

Предложения 2, 3, 5 тоже поддаются объединению, ведь левые части их содержат по две единицы, а правые совпадают:

```
          e.1 '1' e.2 '1' e.3 = '10';
```

Произведя то же преобразование с предложениями 4, 6, 7 получаем следующее описание функции:

[17] Эта задача уже была поставлена выше: п. 4, задача 8.

```
Alfa {
     s.a s.a s.a         = s.a s.a;
     e.1 '1' e.2 '1' e.3 = '10';
     e.1 '0' e.2 '0' e.3 = '01';
     }
```

Во втором предложении две из трех переменных e.1, e.2, e.3 принимают значение пусто, а третья – «0». В третьем предложении также две переменные принимают значение пусто, но третья – «1». Поэтому функцию Alfa можно переписать так:

```
Alfa {
     s.a s.a s.a         = s.a s.a;
     e.1 s.a e.2 s.a e.3 = s.a e.1 e.2 e.3;
     }
```

Наконец, первое предложение можно вообще убрать (убедитесь всё-таки!):

```
Alfa {
     e.1 s.a e.2 s.a e.3 = s.a e.1 e.2 e.3;
     }
```

Итак, количество предложений в функции Alfa уменьшилось с 8 до минимума (до одного предложения).

Пример 10. Сложение двух чисел, записанных в двоичной системе счисления. Формат обращения: <Add2 (e.M) e.N>, где e.M, e.N – слагаемые.

Решение 2. Воспользуемся функцией Add1 из п. 4 и опишем всевозможные случаи, которые возникают при «сложении столбиком». Перенос мы не будем держать в «уме», а будем прибавлять его к первому слагаемому.

```
Add2 {
     (e.1 '0') e.2 '0' = <Add2 (e.1) e.2> '0';
     (e.1 '0') e.2 '1' = <Add2 (e.1) e.2> '1';
     (e.1 '1') e.2 '0' = <Add2 (e.1) e.2> '1';
     (e.1 '1') e.2 '1' = <Add2 (<Add1 e.1>) e.2> '0';
     ( )        e.2    = e.2;
     (e.1)             = e.1;
     (  )              = ;
       }

Add1  {
     e.1 '0' = e.1 '1';
     e.1 '1' = <Add1 e.1> '0';
             = '1';
      }
```

Три последние предложения описывают случаи окончания вычислений.

Заметим, что если мы правильно описали функцию Add2, то результат вычисления рабочего выражения <Add1 e.N> совпадает с результатом вычисления выражения <Add2 (e.N) '1'>. Поэтому не будем пользоваться функцией Add1, а четвертое предложение перепишем так:

```
     (e.1 '1' ) e.2 '1' = <Add2 (<Add2 (e.1) '1'> ) e.2> '0';
```

Последние три предложения объединяются в одно:

```
(e.1) e.2 = e.1 e.2;
```

Поскольку первые четыре предложения описывают взаимноисключающие случаи, то их можно переставлять. Итак,

```
Add2 {
     (e.1 '0') e.2 '0' = <Add2 (e.1) e.2> '0';
     (e.1 '1') e.2 '1' = <Add2 (<Add2 (e.1) '1'>) e.2> '0';
     (e.1 '1') e.2 '0' = <Add2 (e.1) e.2> '1';
     (e.1 '0') e.2 '1' = <Add2 (e.1) e.2> '1';
     (e.1) e.2         = e.1 e.2;
     }
```

Правые части у третьего и четвертого предложений совпадают, поэтому их можно объединить так:

```
(e.1 s.a) e.2 s.b = <Add2 (e.1) e.2> '1';
```

Так как прибавление нуля к числу не должно изменять его значения, то первое предложение можно записать в виде

```
(e.1 '0' ) e.2 '0' = <Add2 (<Add2 (e.1) '0'> ) e.2> '0';
```

Итак, получаем описание функции `Add2`

```
Add2 {
     (e.1 '0') e.2 '0' = <Add2 (<Add2 (e.1) '0'>) e.2> '0';
     (e.1 '1') e.2 '1' = <Add2 (<Add2 (e.1) '1'>) e.2> '0';
     (e.1 s.a) e.2 s.b = <Add2 (e.1) e.2> '1';
     (e.1)     e.2     = e.1 e.2;
     }
```

Теперь видно, чем отличаются первое и второе предложения: вместо символа «`0`» стоит символ «`1`». Поэтому их можно объединить, написав вместо «`0`» и «`1`» переменную `s.a`

```
Add2 {
     (e.1 s.a) e.2 s.a = <Add2 (<Add2 (e.1) s.a>) e.2> '0';
     (e.1 s.a) e.2 s.b = <Add2 (e.1) e.2> '1';
     (e.1)     e.2     = e.1 e.2;
     }
```

Первое предложение соответствует случаю, когда оба числа оканчиваются одинаковой цифрой `s.a`. В этом случае «0 пишем, `s.a` – в уме». Второе предложение – случаю, когда числа оканчиваются разными цифрами. В этом случае «1 пишем, 0 – в уме». Последнее предложение заканчивает сложение.

§ 10. Вычисления в конечных полях

> Со всех сторон выползают монстры – новые спорадические простые группы, которым не видно конца и существование которых устанавливается зачастую при помощи новейших ЭВМ.
>
> А. И. Кострикин. Введение к сборнику «К теории конечных групп», 1979.

Конечные поля рассмотрены в книге [3; пп. 4.4.6, 9.1].

Поле называется конечным, если оно состоит из конечного количества элементов. Для любого простого числа p и любого натурального числа n существует поле, состоящее из p^n элементов. В данном параграфе мы рассмотрим поля, состоящие из p элементов, в п. 13 – произвольные конечные поля.

Множество $\mathbb{Z}_n$ классов вычетов по модулю n образует кольцо. Если p – простое число, то $\mathbb{Z}_p$ образует поле, т. е. в нем можно производить сложение, вычитание, умножение и деление. Мы будем работать со следующими представителями смежных классов:
$0, 1, 2, \ldots, p-1$

Рассмотрим в поле $\mathbb{Z}_{11}$ несколько примеров операций.

$2 + 10 = 1$, так как $2 + 10 = 12 = 1 \pmod{11}$
$2 \times 10 = 9$, так как $2 \times 10 = 20 = 9 \pmod{11}$
$2^{(-1)} = 6$, так как $2 \times 6 = 12 = 1 \pmod{11}$
$5^{(-1)} = 9$, так как $5 \times 9 = 45 = 1 \pmod{11}$

Элементы поля $\mathbb{Z}_p$ будем представлять в виде целых чисел, т. е. в виде последовательности макроцифр.

Нашей задачей является написание функций, осуществляющих операции в поле $\mathbb{Z}_p$. Поскольку эти функции могут понадобиться в других программах, то мы оформим их в виде отдельного модуля. Текст этого модуля приведем в п. 14.3. Мы опишем следующие функции:

- `Zp_Add` – сложение в поле $\mathbb{Z}_p$,
- `Zp_Sub` – вычитание в поле $\mathbb{Z}_p$,
- `Zp_Mul` – умножение в поле $\mathbb{Z}_p$,
- `Zp_Inv` – нахождение обратного элемента по умножению в поле $\mathbb{Z}_p$,
- `Zp_Div` – деление в поле $\mathbb{Z}_p$.

Теперь надо обсудить формат обращения к этим функциям. Результат операции, например, сложения, зависит от слагаемых и от простого числа p. Поэтому можно было бы выбрать такой формат обращения к функции:

```
<Zp_Add (e.M) (e.N) e.P>
```

где `e.M`, `e.N` – слагаемые, `e.P` – простое число.

Этот формат неудобен по следующим причинам. Когда происходит много различных вычислений, то значение `e.P` должно будет присутствовать во всех преобразованиях поля зрения.

Поэтому хотелось бы иметь форматы обращения к функциям `Zp_Add`, `Zp_Sub`, ... такие же, как и для арифметических функций Рефала `Add`, `Sub`, ...

Каким способом получить значение числа p при вычислениях? Одним из способов является использование копилки.

Итак, предлагается следующий режим работы с арифметическими функциями в поле $\mathbb{Z}_p$. Перед самым первым обращением к этим функциям необходимо закопать число в копилке под именем «P» т. е. выполнить `<Br 'P=' e.P>`, где `e.P` – простое число p, представленное в виде последовательности макроцифр.

Например, для поля $\mathbb{Z}_{11}$ пишем

```
<Br 'P=' 11>
```

Формат обращения к функциям следующий:

```
<D (e.M) e.N>
```

если `D` совпадает с одним из имен `Zp_Add`, `Zp_Sub`, `Zp_Mul`, `Zp_Div`, или

```
<Zp_Inv e.M>
```

где `e.M`, `e.N` – целые неотрицательные числа меньшие, чем `e.P`.

Результат замены – целое число, являющееся результатом соответствующей операции в поле $\mathbb{Z}_p$.

Сделаем пояснения к п. 14.3, где изображены функции модулярной арифметики.

Функция `MOD` находит остаток от деления аргумента на число p. Сложение и умножение описываются очевидным образом.

Функция `Divmod` выдает отрицательный остаток, если делимое отрицательно. Поэтому, чтобы не делать проверок на знак остатка, мы в функции `Zp_Sub` сразу прибавляем `e.P` к первому операнду.

Функция `Zp_Div` сводит деление к умножению на обратный элемент.

Наиболее сложной оказывается функция `Zp_Inv`.

Так как m и p взаимно простые числа, то НОД$(m, p) = 1$, поэтому

$$1 = u \times m + v \times p$$

для некоторых целых чисел u, v. Переходя к полю $\mathbb{Z}_p$, получаем $u \times m = 1$, поэтому $u = m^{(-1)}$.

Для практического нахождения m используем следующее замечание.

Если $A = X \times M \pmod{p}$, $B = Y \times M \pmod{p}$, и $A = Q \times B + R$, то $R = (X - Y \times Q) \times M \pmod{p}$.

Мы начинаем с двух очевидных сравнений

$$p = 0 \times M \pmod{p}$$
$$M = 1 \times M \pmod{p}$$

Затем применяем последовательные деления из алгоритма Евклида и получаем для некоторого U

$$1 = U \times M \pmod{p}$$

Поскольку для получения U мы работаем по модулю p, то все операции можно производить в поле $\mathbb{Z}_p$.

Функция `Zp_Inv` делает первый шаг и подготавливает аргумент для функции `Zp_Inv1`. В первой скобке находится значение A, во второй – X, в третьей – B, в четвертой – Y.

Задача 20*. Выражением $GL(n,p)$ обозначают группу невырожденных матриц размера $n \times n$ с коэффициентами из поля $\mathbb{Z}_p$. Напишите функции для вычислений в группе $GL(n,p)$: умножение матриц, обращение матрицы, вычисление определителя матрицы, приведение матрицы к треугольному виду и т. д. Отметим, что все вычисления в группе $GL(n,p)$ осуществляются точно, поэтому нет необходимости применять такие методы вычислительной линейной алгебры, как выбор главного элемента и т. д.

§ 11. Перестановки

Перестановки рассмотрены в книге [3; п. 4.2.4].

Совокупность всех взаимно однозначных отображений множества M, состоящего из n элементов, на себя образует группу, которая называется симметрической группой и обозначается через S_n. Элементы группы S_n называются перестановками. Элементы множества называются символами, и обозначают их либо буквами, либо цифрами.

Будем считать, что элементы множества M являются символами в смысле языка Рефал.

Пусть f – элемент S_n, т. е. f отображает M на себя. Тогда перестановку f будем представлять в поле зрения в виде последовательности из n скобочных выражений вида $(x\ \ f(x))$, где $x \in M$.

Перестановки перемножаются в соответствии с общим правилом композиции отображений:

$$(fg)(x) = f(g(x))$$

Напишем функцию УМН[18] умножения двух перестановок. Формат обращения:

```
<УМН (e.f) e.g>
```

где e.f, e.g – перестановки.

```
УМН {
  (e.1 (s.B s.C) e.2) (s.A s.B) e.3 = (s.A s.C) <УМН (e.1 e.2) e.3>;
  ( ) = ;
    }
```

Объясните, почему скобки (s.B s.C) и (s.A s.B) можно вообще выбрасывать?

Обратная перестановка имеет вид $(f(x)\ x)$, поэтому функция обращения перестановки ОБР записывается так:

```
ОБР {
     (s.A s.B) e.1 = (s.B s.A) <ОБР e.1> ;
                   = ;
    }
```

Для каждой перестановки вводится понятие четности. Напишем функцию ЧЕТ, которая по перестановке выдает в качестве результата замены «0», если перестановка четная, и «1», если нечетная.

[18] Рефал-5 не допускает использования букв Кириллицы в именах идентификаторов. – *Прим. ред.*

```
ЧЕТ { e.1 = <ЧЕТ1 '0' e.1>; }

ЧЕТ1 { s.A (s.B s.B) e.1 = <ЧЕТ1 s.A e.1>;
       s.A (s.B s.C) e.1 (s.D s.B) e.2 =
                     <ЧЕТ1 <ИЗМЧЕТ s.A> e.1 (s.D s.C) e.2>;
       s.A = s.A;
     }

ИЗМЧЕТ { '0' = '1' ;
         '1' = '0' ;
       }
```

Пример 11. Рассмотрим известную игру в «пятнадцать» (см. например, [3; гл.4, п.2]). В этой игре требуется от расположения фишек

$$\begin{array}{cccc} f_1 & f_2 & f_3 & f_4 \\ f_5 & f_6 & f_7 & f_8 \\ f_9 & f_{10} & f_{11} & f_{12} \\ f_{13} & f_{14} & f_{15} & \end{array}$$

перейти к правильному расположению

$$\begin{array}{cccc} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & \end{array}$$

С задачей ассоциируется перестановка f из группы S_{15}, действующая по формуле $f(i) = f_i$. Переход в правильное расположение возможен, если перестановка f – четная.

Написать функцию ИГРА15, которая получает в качестве аргумента 15 чисел f_1, f_2, …, f_{15}, перечисленных через запятую, и результатом замены которой является «1», если правильное расположение достижимо, и «0», если не достижимо.

Решение 3. Сначала строим перестановку функцией ИГРА15А, затем функция ЧЕТ определяет четность перестановки, а функция ИЗМЧЕТ заменяет нули на единицы.

```
ИГРА15 {
     e.1 = <ИЗМЧЕТ <ЧЕТ <ИГРА15А 1 e.1>>>;
       }

ИГРА15А {
     s.A e.1 ',' e.2
             = (s.A <NUMB e.1>) <ИГРА15А <ADD (s.A) 1> e.2>;
     s.A e.1 = (s.A <NUMB e.1>);
        }
```

Числа от 1 до 15 представляются макроцифрами.

Задача 21. В предыдущем примере предполагается, что пустая клетка находится в правом нижнем углу. Обозначив пустую клетку числом 16, мы получаем перестановку из группы S_{16}. Написать функцию определения возможности перехода в правильное расположение, если начальное расположение задается шестнадцатью числами, перечисленными через запятую.

Задача 22. Имеется другой способ записи перестановок – в виде произведения циклов. Например, перестановка $(1,2)(2,3)(3,1)(4,5)(5,4)$ записывается как (1 2 3)(4 5). Написать функции `УМН`, `ОБР`, `ЧЕТ`, работающие с перестановками, записанными в виде циклов.

§ 12. Операции логики высказываний

Необходимые сведения из математической логики приведены в книге [4]. В данном параграфе мы напишем интерпретатор формул логики высказываний `LOGIC`.

Программа `LOGIC` будет обрабатывать приказы двух видов:

1. определения формул,
2. вычисления формул.

Поскольку большие формулы всегда можно разбить на подформулы, предполагается, что каждый приказ подготавливается отдельной строкой.

Приказ определения формул имеет формат

```
e.n '=' e.0
```

где `e.n` – имя переменной, `e.0` – формула логики высказываний.

В результате выполнения этого приказа запоминается определение формулы.

Приказ вычисления формул имеет формат

```
e.0
```

где `e.0` – произвольная формула логики высказываний.

В результате выполнения этого приказа происходит вычисление формулы `e.0`. Предполагается, что формула `e.0` может содержать любые переменные.

Формула логики высказываний состоит из переменных, знаков операций, скобок и констант. Значения всех переменных по умолчанию предполагаются равными истине.

В качестве знаков логических операций выберем следующие:

- `.And.` – конъюнкция,
- `.Or.` – дизъюнкция,
- `.Imp.` – импликация,
- `.Not.` – отрицание.

Константу «*истина*» будем обозначать через «`1`», «*ложь*» – через «`0`».

Старшинство операций определим в следующем порядке: отрицание, конъюнкция, дизъюнкция, импликация. Впрочем, всегда можно использовать скобки.

Признаком конца приказов служит приказ

```
END
```

Пример 12. Пусть мы хотим вычислить значение формулы

```
(A .And. (B .Or. C)) .Imp. ((A .And. B) .Or. (A .And. C))
```

при `A = 0`, `B = 0`, `C = 1`.

Тогда можно подготовить такие приказы:

```
F1 = A .And. (B .Or. C)
F2 = (A .And. B) .Or. (A .And. C)
F = F1 .Imp. F2
A = 0
B = 0
F
END
```

В результате вычисления формулы F напечатается

```
= 1
```

Текст интерпретатора формул логики высказываний приведен в п. 14.4.

Функция `Wwod` осуществляет ввод одной строки, убирание на ней всех пробелов, удобные для работы замены, а затем спаривание скобок.

Функции `LOGIC` и `Begin` организуют ввод приказов и проверку на окончание. Каждый приказ обрабатывается функцией `Rab`.

Выполнение приказа-определения заключается в закапывании под именем `e.N` формулы `e.0`.

Выполнение приказа-вычисления осуществляет функция `Wich`, которая анализирует формулу и строит функциональные термы для вычисления логических операций (функции `Imp`, `Or`, `Amd`, `Not`).

Предложение

```
  (e.1) = <Wich e.1>;
```

осуществляет вхождение внутрь скобок.

Отметим, что все приказы-определения запоминаются и их можно использовать в дальнейшем. Например, для того, чтобы вычислить значение формулы из предыдущего примера при `A = 0`, `B = 0`, `C = 0`, достаточно набрать приказы:

```
C=0
F
```

В результате будет напечатано:

```
= 1
```

Объясните, как происходит присвоение переменным значения «1» по умолчанию.

Задача 23. Напишите Рефал-программу приведения формулы логики высказываний к конъюктивно и дизъюктивно нормальным формам.

Задача 24. Напишите функцию-предикат определения тождественной истинности формул логики высказываний.

Задача 25. Напишите функцию-предикат определения равносильности двух формул.

Задача 26. Расширьте возможности программы `LOGIC`. Например, добавьте возможность определения и вычисления логических функций от каких-то переменных.

§ 13. Действия над полиномами

Для работы с многочленами нужно выбрать представление многочлена в поле зрения. Поскольку многочлен

$$a_n x^n + \ldots + a_2 x^2 + a_1 x + a_0$$

определяется своими коэффициентами $a_n, \ldots, a_2, a_1, a_0$, то естественно представлять полином в поле зрения в виде

`(e.n) ...(e.2) (e.1) (e.0)`

где `e.i` – объектное выражение, определяющее коэффициент a_i.

Задача 27. Написать функции сложения, вычитания, умножения двух многочленов с целыми коэффициентами.

Задача 28. Написать функции сложения, вычитания, умножения и деления с остатком для кольца многочленов с рациональными коэффициентами (рациональное число задается парой целых чисел). Коэффициенты многочленов можно выбирать из произвольного кольца.

Пример 13. Написать функции сложения, умножения и деления с остатком для многочленов с коэффициентами из поля $\mathbb{Z}_2$. (Так как в поле $\mathbb{Z}_2$ выполняется равенство $1 + 1 = 0$, то операция вычитания совпадает со сложением.)

Решение 4. Многочлен, у которого $a_n = 1$, а остальные n коэффициентов равны 0 или 1, будем представлять в поле зрения в виде последовательности из $n + 1$ символов, каждый из которых может принимать значения 0 или 1. Нулевой многочлен будем представлять в виде пустого выражения.

Функция сложения двух многочленов описывается функциями `Z2X_Add`, `Z2X_Add1`, `Delete_0` (см. п. 14.5). Функция `Delete_0` убирает впередистоящие нули, `Z2X_Add1` – осуществляет сложение. Функция `Z2X_Mul` умножает два многочлена. Эта функция моделирует «умножение столбиком».

Напишем функцию `Z2X_Div` деления с остатком двух многочленов. Деление всегда возможно, т. к. старший коэффициент равен 1. Вспомогательная функция `Z2X_Div1` отщепляет от делимого столько символов, какова длина делителя. Функция `Z2X_Div2` осуществляет фактически «деление столбиком», обращаясь каждый раз к функции `Z2X_Div3` для проверки окончания вычислений.

С многочленами тесно связаны поля, состоящие из $q = p^n$ элементов, $n > 1$.

Конструкция поля $GF(q)$, состоящего из q элементов, аналогична конструкции поля $\mathbb{Z}_p$. Только вместо простого числа выбирается некоторый неприводимый многочлен $p(x)$, степени n с коэффициентами из поля $\mathbb{Z}_p$.

Элементами поля $GF(q)$ являются всевозможные остатки от деления многочленов на многочлен $p(x)$, т. е. классы вычетов по модулю многочлена $p(x)$. В

качестве результата операции в поле $GF(q)$ берется остаток от деления на $p(x)$ результата обычной операции над многочленами.

В качестве иллюстрации действий в поле $GF(q)$ мы напишем функции при $p = 2$.

Так же как и в примере модулярной арифметики (п. 10), неприводимый многочлен $p(x)$ закапывается в копилке под именем «PX».

Обозначим функции сложения, умножения, деления и нахождения обратного элемента через F8_Add, F8_Mul, F8_Div, F8_Inv. Форматы обращений пусть будут такими же, как и для функций модулярной арифметики п. 14.5.

Отметим, что эти функции очень похожи на функции п. 14.3.

Задача 29. Написать функции для вычислений в $GF(q)$ для произвольного простого числа q. Естественно, прийдется использовать функции Fp_Add, ...

Задача 30. Написать функцию, которая по простому числу p и натуральному числу n строит неприводимый многочлен $p(x)$ степени n.

§ 14. Примеры программирования

14.1. Вычисление числа с точностью 2000 знаков после запятой. Дополнительные пояснения к программе даны в п. 6.6, пример 7. Программа также доступна на электронной странице:
http://refal.botik.ru/Korlyukov/examples/e2000.ref.

```
*****************************************************************

$ENTRY Go { =
*             Для контрольного пуска на счет:
*    <Open 'w' 1 'ttttt.txt'> <Putout 1 <E2000 >>;
     <Prout <E2000 >>;
          }
*
*             Перевод чисел в макроцифры

CVB   {   e.1 = <CVB1 ( 0 ) e.1>; }

CVB1  {
        (e.N) s.A e.1 = <CVB1 (<Add (<Numb s.A>)
                                    <Mul (e.N) 10 >>) e.1>;
        (e.N)         = e.N;
      }

*            Перевод макроцифр в последовательность цифр
CVD  {  e.1 = <CVD1 <Divmod (e.1) 10 >>; }
CVD1  {
        ( 0 )  s.X =           <Symb s.X>;
        ( e.1) s.X = <CVD e.1> <Symb s.X>;
      }
```

```
*            Вычисление числа e = 2.71828...
*            с точностью 2000 знаков после запятой

E2000  {  = <E2001  1 ( 1 ) ( 1 )>; }

E2001  {
        1001 (e.1) (e.3) = <E2002 (e.1) (e.3)>;
        s.A   (e.1) (e.3) =
                           <E2001 <Add (s.A) 1>
                                  (<Add (<Mul (e.1) s.A>) 1 >)
                                  (<Mul (e.3) 100 >) >;
       }

E2002  {
       (e.1) (e.3) = <E2003 <CVD  <Div1 2  <Mul (e.1) e.3>> >>;
       }
Div1  {  1001  e.1 = e.1;
         s.a   e.1 = <Div1 <Add (s.a) 1> <Div (e.1) s.a>>;
      }

E2003  { s.A e.1 = s.A '.' e.1 ; }
```

14.2. Программа, порождающая текст программы на языке Basic. Пояснения к программе даны в п. 8. Программа также доступна на электронной странице:
http://refal.botik.ru/Korlyukov/examples/gener.ref.

```
$ENTRY  Go  {
         = <Open 'w' 1 'GN.BAS'>
*             Для контрольного пуска на счет:
           <Rab <Spar <Ubpr <Vvod  <Prout 'F(t)= ...  the end - &'>
                           <Card > >>>>;
            }
*                    Ввод до символа '&'
Vvod  {
       e.1 '&' e.2 = e.1;
       e.1         = e.1 <Vvod <Card >>;
      }
*                    Удаление всех пробелов
Ubpr  {
       e.1 ' ' e.2 = e.1 <Ubpr e.2>;
       e.1         = e.1;
      }
```

```
*          Организация вывода результирующей программы
Rab  {
       e.1 = <Put 1 '10 t = 0.0'>
             <Put 1 '20 FOR I=1 TO 100'>
             <Put 1 '30 t = t + 0.01'>
             <Put 1 '40 F = ' <Rassk e.1>>
             <Put 1 '50 FDt = ' <Rassk <Easy <Diff ('t') e.1>>>>
             <Put 1 '60 PRINT "t="; t, "F="; F, "FD="; FDt'>
             <Put 1 '70 NEXT I'>
             <Put 1 '80 END'>;
      }
*                  Расспаривание скобок для вывода
Rassk  {
      e.1 (e.2) e.3 = e.1 '(' <Rassk e.2 ')' e.3>;
      e.1           = e.1;
       }

*                            Спаривание скобок

Spar  {  e.1 = <Spar1 ('*') e.1 >; }

Spar1  {
      (e.1)       '(' e.3 = <Spar1 ((e.1))     e.3 >;
      ((e.1) e.2) ')' e.3 = <Spar1 (e.1 (e.2)) e.3 >;
      ('*'e.1)    ')' e.3 = 'error' e.1 ')'   e.3;
      (e.1)       s.A e.3 = <Spar1 (e.1 s.A)   e.3 >;
      ('*'e.1)            = e.1;
      ((e.1) e.2)         = 'error' e.1 '(' e.2;
       }

* Функция дифференцирования выражения e.z по переменной e.t
*                       Обращение: <Diff (e.t) e.z>
Diff  {
  (e.t) 'ln'  (e.1)  = (<Diff (e.t) e.1>) '/' (e.1);
  (e.t) 'sin' (e.1)  =  'cos' (e.1) '*' (<Diff (e.t) e.1>);
  (e.t) 'cos' (e.1)  = ('-sin' (e.1)) '*' (<Diff (e.t) e.1>);
  (e.t) 'tg'  (e.1)  = (<Diff (e.t) e.1>) '/' ('cos'(e.1) '^2');
  (e.t) 'ctg' (e.1)  = ('-' (<Diff (e.t) e.1>)) '/' ('sin'(e.1) '^2');
  (e.t) 'exp' (e.1)  = (<Diff (e.t) e.1>) '*exp' (e.1);
  (e.t) e.1 '+' e.2  = <Diff (e.t) e.1> '+' <Diff (e.t) e.2>;
  (e.t) e.1 '-' e.2  = <Diff (e.t) e.1> '-' <Diff (e.t) e.2>;
  (e.t) e.1 '*' e.2  = (<Diff (e.t) e.1> '*' e.2)
                        '+' (e.1 '*' <Diff (e.t) e.2>);
  (e.t) e.1 '/' e.2  = ( (<Diff (e.t) e.1> '*' e.2)
                          '-' (e.1 '*' <Diff (e.t) e.2>) )
                       '/' (e.2 '^2');
```

```
  (e.t) e.1 ':' e.2  = ( (<Diff (e.t) e.1> '*' e.2)
                         '-' (e.1 '*' <Diff (e.t) e.2>) )
                       ':' (e.2 '^2');
  (e.t) e.1 '^' e.2  = (e.2 '*' ((e.1 '^' (e.2 '-1')) '*'
                         <Diff (e.t) e.1>)) '+'
                        ( e.1 '^' e.2 ) '*ln' (e.1) '*'
                         (<Diff (e.t) e.2>);
  (e.t) (e.1) = ( <Diff (e.t) e.1> );
  (e.t) e.t  = '1';
  (e.t) e.1  = '0';
     }

*  Один из вариантов возможного упрощения полученной функции

Easy  {
     e.1 '+0'      = <Easy e.1 >;
     e.1 '-0'      = <Easy e.1>;
     '0+' e.1      = <Easy e.1>;
     '0-' e.1      = <Easy '-' e.1>;
     e.1 '+0+' e.2 = <Easy e.1 '+' e.2>;
     e.1 '-0+' e.2 = <Easy e.1 '+' e.2>;
     e.1 '+0-' e.2 = <Easy e.1 '-' e.2>;
     e.1 '-0-' e.2 = <Easy e.1 '-' e.2>;
     e.1 '*1+' e.2 = <Easy e.1 '+' e.2>;
     e.1 '+1*' e.2 = <Easy e.1 '+' e.2>;
     e.1 '*1-' e.2 = <Easy e.1 '-' e.2>;
     e.1 '-1*' e.2 = <Easy e.1 '-' e.2>;
     e.1 '*1'      = <Easy e.1>;
     '1*' e.1      = <Easy e.1>;
     '-1*' e.1     = <Easy '-'e.1>;
     e.1 '*-1'     = <Easy '-'e.1>;
     e.1           = e.1;
       }

*********************************************************************
```

14.3. Арифметика в конечных полях. Пояснения к программе даны в п. 10. Программа также доступна на электронной странице: http://refal.botik.ru/Korlyukov/examples/z_p.ref.

```
*********************************************************************
$ENTRY  Go  {  =
*             Для контрольного пуска на счет:
               <Br 'P=' 7>
               <Prout <Zp_Add ( 5 ) 4> ' = ' 2>
               <Prout <Zp_Mul ( 5 ) 4> ' = ' 6>
               <Prout <Zp_Inv 5 >       ' = ' 3>;   }
```

```
$ENTRY Zp_Add  { (e.A) e.B = <MOD <Add (e.A) e.B>>; }

MOD    {  e.1 = <MOD1 <Divmod (e.1) <Cp 'P'> >>; }

MOD1   { (e.Q) e.R = e.R; }



$ENTRY Zp_Sub {
        (e.A) e.B = <MOD <Sub  (<Add (e.A) <Cp'P'>>) e.B>>;
              }

$ENTRY Zp_Mul  { (e.A) e.B = <MOD <Mul (e.A) e.B>>;  }

$ENTRY Zp_Div  { (e.A) e.B = <Zp_Mul (e.A) <Zp_Inv e.B>>; }

$ENTRY Zp_Inv  {  e.M = <Zp_Inv1 (<Cp 'P'>) ( 0 ) (e.M) ( 1 )>; }

Zp_Inv1  {
       (e.A) (e.X) ( 1 ) (e.Y) = e.Y;
       (e.A) (e.X) (e.B) (e.Y) = <Zp_Inv1 (e.B) (e.Y)
                          <Zp_Inv2 (e.X) (e.Y)  <Divmod (e.A) e.B>>>;
        }

Zp_Inv2  {  (e.X) (e.Y) (e.Q) e.R = (e.R)
                        (<Zp_Sub (e.X) <Zp_Mul (e.Y) e.Q> >) ; }

*****************************************************************
```

14.4. Интерпретатор формул логики высказываний. Пояснения к программе даны в п. 12. Программа также доступна на электронной странице:
http://refal.botik.ru/Korlyukov/examples/logic.ref.

```
*****************************************************************

*             Для контрольного пуска на счет:
$ENTRY Go { = <LOGIC > ; }

* ----------------------------------------------
*    Программа, интерпретирующая формулы
*    логики высказываний.
*    "1" - истина, "0" - ложь
* ----------------------------------------------
$ENTRY   LOGIC {  = <Begin  <Wwod >>; }

Wwod  { = <Spar <Zam  <Card >>>;  }
```

```
Begin  {  'END' = ;
           e.1  = <Rab e.1> <LOGIC>;
       }

Rab  {
         e.1 '=' e.2 = <Rp  e.1 '=' e.2>;
         e.1         = <Prout  ' =  ' <Wich e.1>>;
     }

Wich  {
         e.1  Imp  e.2 = <Imp <Wich e.1> <Wich e.2>>;
         e.1  Or   e.2 = <Or  <Wich e.1> <Wich e.2>>;
         e.1  And  e.2 = <And <Wich e.1> <Wich e.2>>;
         Not e.1       = <Not <Wich e.1>>;
         (e.1)         = <Wich e.1>;
                '1' = '1';
                '0' = '0';
                    = '1';
                e.1 = <Wich <Cp e.1>>;
     }

Imp  { '10' = '0';
       e.1 = '1';  }

Or  {  '00' = '0';
       e.1 = '1';  }

And  { '11' = '1';
       e.1 = '0';  }

Not  {  '0' = '1';
       '1' = '0';  }


Spar  {
       e.1 = <Spar1 ('*') e.1>;
      }

Spar1  {
        (e.1)      '('  e.3 = <Spar1 ( (e.1) )   e.3>;
        ((e.1) e.2) ')' e.3 = <Spar1 (e.1 (e.2)) e.3>;
        ('*'   e.1) ')' e.3 = 'error'  e.1 ')' e.3 ;
        (e.1)       s.A e.3 = <Spar1 (e.1 s.A)   e.3>;
        ( '*' e.1 )         = e.1;
        ( (e.1) e.2 )       = 'error' e.1  '(' e.2 ;
       }
```

```
Zam   {  ' '     e.1 =     <Zam e.1>;
         '.And.' e.1 = And <Zam e.1>;
         '.Or.'  e.1 = Or  <Zam e.1>;
         '.Imp.' e.1 = Imp <Zam e.1>;
         '.Not.' e.1 = Not <Zam e.1>;
         s.a     e.1 = s.a <Zam e.1>;
                     = ;
       }
```

****

14.5. Действия над полиномами. Арифметика поля Галуа $GL(2^p)$. Дополнительные пояснения к программе даны в п. 13. Программа также доступна на электронной странице:
http://refal.botik.ru/Korlyukov/examples/f8_i2.ref.

****

```
*             Для контрольного пуска на счет:
$ENTRY Go { =
* x^2 + x + 1 -- неприводимый многочлен.
* Вычислить элемент, обратный к x + 1.
  <Prout  <F8_Int  Inv ( '111' ) '11'> ' = 10'>

* x^3 + x + 1 -- неприводимый многочлен.
* Сложить x^2 + x + 1  и x^2 + x.
  <Prout  <F8_Int  Add ( '1011' )  ( '111' ) '110'> ' = 1'>

* x^3 + x + 1 -- неприводимый многочлен.
* Умножить x^2 + x + 1 на x^2 + x.
  <Prout  <F8_Int  Mul ( '1011' )  ( '111' ) '11'> ' = 10'>;
} /*  Для тестирования. */

*   Арифметика конечного поля Галуа характеристики 2.
*   e.PX - неприводимый многочлен над полем Z_2 (из двух элементов),
*   e.1 - аргументы
*   e.PX и e.1 кодируются последовательностями '0' и '1'.
*   Примеры:  '101'  == x^2 + 1
*             '1011' == x^3 + x + 1
*      /* пусто */   == 0
*

F8_Int   {
         s.Oper ( e.PX ) e.1 = <Br 'PX=' e.PX>
                               <F8_Int1 s.Oper e.1> ;
         }
```

```
F8_Int1  {
         Add (e.1) e.2 = <F8_Add (e.1) e.2>;
         Sub (e.1) e.2 = <F8_Add (e.1) e.2>;
         Mul (e.1) e.2 = <F8_Mul (e.1) e.2>;
         Div (e.1) e.2 = <F8_Div (e.1) e.2>;
         Inv e.1       = <F8_Inv e.1>;
         }

* Сложение многочленов над полем Z_2.
* Вычитание совпадает со сложением.
$ENTRY Z2X_Add  { (e.1) e.2 = <Delete_0 <Z2X_Add1  (e.1) e.2>>; }

Z2X_Add1  {
          (e.1 s.A) e.2 s.A = <Z2X_Add1 (e.1) e.2> '0' ;
          (e.1 s.A) e.2 s.B = <Z2X_Add1 (e.1) e.2> '1' ;
          (e.1)     e.2     = e.1 e.2 ;
          }

* Удаление лидирующих нулей кодировки многочлена.
Delete_0  {
          '0' e.2 = <Delete_0 e.2>;
          '1' e.2 = '1' e.2;
              e.1 =  ;
          }

* Умножение многочленов над полем Z_2.
* Школьный алгоритм умножения "столбиком".
$ENTRY Z2X_Mul  {
         (e.1)         = ;
         (  )      e.2 = ;
         (e.1) s.A e.2 = <Z2X_Mul1 (e.1) (e.1) e.2>;
         }

Z2X_Mul1  {
 (e.1) (e.2)         = e.1;
 (e.1) (e.2) '0' e.3 = <Z2X_Mul1 (e.1 '0' ) (e.2) e.3>;
 (e.1) (e.2) '1' e.3 =
               <Z2X_Mul1 (<Z2X_Add (e.1 '0' ) e.2>) (e.2) e.3>;
          }

* Алгоритм Евклида деления многочленов над полем Z_2.
* Школьный алгоритм деления "столбиком".
* <Z2X_div  (e.a) e.b> ==> e.q (e.rest)
$ENTRY Z2X_Div  {
         (   ) e.1 = ( );
         (e.1) e.2 = <Z2X_Div1 ( ) (e.1) ( ) e.2>; }
```

```
Z2X_Div1  {
  (e.1) (s.A e.2) (e.3) s.B e.4 =
                      <Z2X_Div1 (e.1 s.A) (e.2) (e.3 s.B) e.4>;
  (e.1) ( )       (e.3) s.B e.4 = (e.1);
  (e.1) (e.2)     (e.3)         = <Z2X_Div2 (e.1) (e.2) (e.3)>;
         }

Z2X_Div2  {
  ('1'e.1) (e.2) (e.3) =
           '1' <Z2X_Div3 (<Z2X_Add1 ('1'e.1)e.3>)(e.2)(e.3)>;
  ('0'e.1) (e.2) (e.3) = '0' <Z2X_Div3 ('0' e.1) (e.2) (e.3)>;
         }

Z2X_Div3  {
       ('0'e.1) ( )       (e.3) = (<Delete_0 e.1>);
       ('0'e.1) (s.B e.2) (e.3) = <Z2X_Div2 (e.1 s.B) (e.2) (e.3)>;
          }

* Сложение в поле Галуа характеристики 2.
$ENTRY F8_Add  { (e.1) e.2 = <Z2X_Add (e.1) e.2>; }

* Умножение в поле Галуа характеристики 2.
$ENTRY F8_Mul  { (e.1) e.2 = <Mod_PX <Z2X_Mul (e.1) e.2>>; }

Mod_PX  {  e.1, <Cp 'PX' >: e.P = <Mod_PX1 <Z2X_Div (e.1) e.P>>; }
Mod_PX1 {  e.Q (e.R) = e.R; }

* Деление в поле Галуа характеристики 2.
$ENTRY F8_Div    {  (e.1) e.2 = <F8_Mul (e.1) <F8_Inv  e.2>>; }

* Вычисление обратного элемента в поле Галуа характеристики 2.
* Алгоритм Евклида: GCD(M,p) = U*M + V*p
*                           1  = U*M
*                           U  = M^(-1)
/*
  Замечание: A = X*M (mod p)
             B = Y*M (mod p)
           и A = Q*B + R,
  тогда  R = (X-Y*Q)*M (mod p)
  Мы начинаем с p = 0*M (mod p)
                M = 1*M (mod p)
*/
$ENTRY F8_Inv  {
       e.m, <Cp 'PX' >: e.P = <F8_Inv1  ( e.P ) ( ) (e.m) ( '1' )>;
        }
```

```
F8_Inv1  {
      (e.A) (e.X) ('1') (e.Y) = e.Y;
      (e.A) (e.X) (e.B) (e.Y) = <F8_Inv1 (e.B) (e.Y)
                          <F8_Inv2 (e.X) (e.Y) <Z2X_Div (e.A) e.B>>>;
         }

F8_Inv2  {
  (e.X) (e.Y) e.Q (e.R) = (e.R) (<F8_Add (e.X) <F8_Mul (e.Y) e.Q>>);
         }
```

****

Список литературы

[1] *Алгоритмический язык рекурсивных функций РЕФАЛ. Руководство программиста*, Программное обеспечение СМ ЭВМ. Операционная система с разделением функций. РАФОС, **5(8)**.

[2] *Базисный Рефал и его реализации на вычислительных машинах: Методические рекомендации*, ЦНИПИАСС, М., 1977.

[3] А. И. Кострикин, *Введение в алгебру*, Наука, Москва, 1977.

[4] Э. Мендельсон, *Введение в математическую логику*, Наука, Москва, 1976.

[5] В. Ф. Турчин, "Метаязык для формального описания алгоритмических языков", *Цифровая вычислительная техника и программирование*, Советское радио, М., 1966, 116–119.

[6] V. F. Turchin, *REFAL-5 programming guide and reference manual*, (*Прим. ред.:* переработанное и расширенное издание 1999 года доступно как zipped html-файл: http://refal.botik.ru/book/refal-book-html.zip), New England Publishing Co., Holyoke, 1989.

Ниже следуют ссылки, добавленные редактором.

[7] Р. Ф. Гурин, Ю. А. Климов, А. Ю. Орлов, С. А. Романенко, *Рефал+: исполняемые модули и исходные тексты*, ([online]: http://rfp.botik.ru/), 2000.

[8] Р. Ф. Гурин, С. А. Романенко, *Язык программирования РЕФАЛ ПЛЮС*, (электронная версия доступна на странице: http://www.refal.net/doc_ref_plus.html), ИНТЕРТЕХ, Москва, 1991.

[9] В. Ф. Турчин, *Программирование на языке РЕФАЛ*, Препринт №43, ИПМ АН СССР, 1971.

[10] В. Ф. Турчин, Д. В. Турчин, А. П. Конышев, А. П. Немытых, *Рефал-5: исполняемые модули и исходные тексты*, ([online]: http://www.botik.ru/pub/local/scp/refal5/), 2000.

[11] V. F. Turchin, *REFAL-5 programming guide and reference manual*, (переработанное и расширенное издание доступно как zipped html-файл: http://refal.botik.ru/book/refal-book-html.zip), 1999.

А. В. Корлюков (A. V. Korlyukov)
Гродно, Гродненский государственный университет

Си Гао, А. П. Немытых

CREFAL: компилятор из языка программирования Рефал-5 в язык сборки

Система программирования Рефал-5 [19], как и большинство систем, поддерживающих функциональные языки программирования, является полуинтерпретатором: программы компилируются из языка Рефал-5 в промежуточный язык сборки, далее результат компиляции интерпретируется.

В систему программирования Рефал-5 входит компилятор `refc` из Рефала-5 в язык сборки. По историческим причинам `refc` реализован на языке «Си» и представляет собой бинарный модуль, исполняемый соответствующей операционной системой.

В данной статье кратко описываются некоторые свойства `crefal` – другого компилятор из Рефала-5 в язык сборки. Компилятор `crefal` реализован авторами статьи на языке Рефал-5. Дается сравнение компиляторов `refc` и `crefal`.

Библ. 19 наим.

Ключевые слова: Рефал, компилятор, язык сборки, метавычисления, тестирование.

Содержание

Работа выполнена в 2004 г.

Введение

Реализация Рефала-5 [16] (автор диалекта В. Ф. Турчин [19]) представляет собой полуинтерпретатор: Рефал-программа компилируется на промежуточный язык `RASL`[1] и далее результат компиляции интерпретируется.

Рефал-5 является консольным приложением [16, 11].

Начальный вариант исходных текстов соответствующих компилятора refc и интерпретатора был написан на языке программирования «Си» в конце 1980-х гг. Дмитрием В. Турчиным под руководством Валентина Фёдоровича Турчина. Далее система программирования Рефал-5 поддерживалась несколькими авторами [16, 11].

Документацией к данной реализации является книга В. Ф. Турчина, написанная на английском языке [19]. На странице [19] можно читать эту книгу, находясь в сети, либо скопировать архив на ваш компьютер. В online версии книги можно исполнять (и изменять) Рефал-программы прямо в интернете – по ходу чтения книги. На странице [15] имеется русский перевод устаревшей версии книги В. Ф. Турчина. В этой версии книги используется устаревший синтаксис Рефала-5.

Решение о реализации компилятора из Рефала-5 в `RASL` на языке низкого уровня – в данном случае языке «Си» – является достаточно нетрадиционным для диалектов Рефала. Обычно компиляторы в языки сборки писались непосредственно на языке Рефал [12, 2, 5], который и разрабатывался как инструмент для решения подобных задач. Хотя были и исключения.[2]

Реализация, поддержка и развитие на языке «Си» компилятора в `RASL` являются намного более трудоемкими, по сравнению с решением тех же задач на языке Рефал. По-видимому, основной причиной для такой реализации[3] была низкая производительность вычислительных машин 1980-х годов: исполнение бинарного модуля непосредственно операционной системой происходит быстрее, чем компиляция посредством интерпретации `rasl`-модуля.

В данной заметке мы сообщаем о компиляторе `crefal` из Рефала-5 в `RASL`.

При разработке компилятора `crefal` авторы ставили перед собой три цели:

- добавить в систему программирования Рефал-5 легко поддерживаемый и развиваемый компилятор в язык сборки;
- реализовать более точную и подробную, по сравнению с выдаваемой компилятором `refc`, систему сообщений об ошибках в компилируемой программе;
- обучение технологии реализации Рефала первого автора – китайского студента.[4]

[1] От Refal ASsembler Language.

[2] Начальная версия компилятора диалекта Рефал+ [4] была написана на языке «Си». Этот компилятор собирает «Си»-программу из макросов исходных текстов `rasl`-операторов. Далее эта программа транслируется компилятором языка «Си».

[3] Работа над реализацией проводилась в г. Нью-Йорке.

[4] Для этой цели был выбран студент университета г. Ухань (в англоязычном варианте транскрипции – Wuhan). Что укладывается в историческую традицию распространения элементов европейской культуры в китайской цивилизации. Есть некоторые основания полагать, что в XIII-ом веке в Ухани побывал итальянский путешественник Марко Поло. Во второй половине XIX-ого века русские купцы построили в Ухани фабрики по производству «кир-

Первые две цели достигнуты. И мы надеемся, что один из будущих студентов добавит к нашему компилятору `crefal` новые алгоритмы оптимизации Рефал-программ, которых нет в компиляторе `refc`.

§ 1. `crefal` vs. `refc`

Компилятор `refc` написан на языке «Си», как и интерпретатор Рефала-5, и, конечно, более эффективен по времени исполнения[5], чем `crefal`. Время исполнения `crefal` включает в себя накладные расходы интерпретации Рефал-программы `crefal.ref`, но это время исполнения на современных вычислительных машинах[6] вполне приемлемо.

О гибкости реализации `crefal.ref` – с точки зрения поддержки и развития – мы уже упомянули в п. 2.

1.1. Эффективность порождаемого `rasl`-кода. Компилятор `crefal` разрабатывался в два этапа. В исходных текстах компилятора `crefal` эти два этапа композиционно разделены с целью облегчения поиска ошибок, которые могут обнаружиться в будущем.

Первый этап трансляции. На первом этапе решалась нижеследующая задача.

Пусть дана произвольная Рефал-программа `p`. `crefal` должен построить `rasl`-код программы `p`, *текстуально* совпадающий с выходом компилятора `refc`, когда `refc` транслирует программу `p`.

Это свойство компилятора `crefal` позволило автоматизировать тестирование `crefal` после первого этапа его разработки (см. п. 2).

Перечислим традиционные для реализаций диалектов Рефала `rasl`-операторы, поддерживающие элементы оптимизации, которые используют оба рассматриваемые компилятора для построения `rasl`-кода [1, 2, 12, 5]:

1. отождествление константной строки подряд стоящих в образце символов посредством цикла по этой строке, а не отдельно каждого символа;
2. построение константной строки подряд стоящих в Рефал-выражении символов посредством цикла по этой строке, а не отдельно каждого символа;
3. объединение в один оператор композиции нескольких операторов, не совпадающих с указанными выше операторами работы с символами;
4. перенос значения[7] `e/t`-переменной из Рефал-образца в выражение, которое строится согласно правой части соответствующего Рефал-предложения; эта правая часть начинается после знака равенства `=`.

пичного» чая. В 1876 г. в городе, в российской концессии, открыта православная церковь св. Александра Невского. Здание храма существует и поныне. Первый мост через реку Янцзы был спроектирован российским инженерами и построен в г. Ухань в 1957 году – в сотрудничестве с Россией. – *Прим. ред.*

[5] Т. е. времени компиляции.

[6] Включая «карманные» вычислительные машины.

[7] Т. е. не копирование структур этого выражения, а передвижение самих структур из одного существующего Рефал-выражения в выражение, которое строится в данный момент Рефал-машиной.

Компилятор `refc` не использует оператор `TPL` переноса значений переменных для построения выражений *expr*, входящих в конструкции Рефал-условий и блоков. Эти выражения синтаксически обрамляются знаками запятой и двоеточия: `,` *expr* `:` . Здесь перенос значений указанных переменных может привести к неверному результату вычисления программы, поскольку, например, значение структуры представления результата вычисления *expr* после перехода Рефал-машины к вычислению выражений, стоящих после знака двоеточия, или отката Рефал-машины для продолжения отождествления с образцом, стоящим перед данной запятой, в общем случае, не определено (см. также пояснения, данные к примеру 1.1).

Компиляторы `refc` и `crefal`, *используя один и тот же алгоритм на первом этапе преобразования*, строят дерево отождествления T, вынося общие префиксы последовательно идущих операторов отождествления двух или нескольких рядом стоящих Рефал-предложений на ребро дерева T, входящее в вершину ветвления оставшихся частей этих предложений.

Мы опускаем описание подробностей этого алгоритма[8]. Заметим только, что:

- если не удается непосредственно удлинить текущее значение префикса π, то происходит попытка декомпозиции `rasl`-операторов, которые допускают декомпозицию в рамках синтаксиса языка `RASL`; после такого преобразования может появиться возможность удлинения значения префикса π;
- в префикс отождествления не выносятся: операторы работы с удлинением значений открытых `e`-переменных, операторы отождествления из конструкций Рефал-блоков и Рефал-условий.

Второй этап трансляции. На втором этапе разработки нами были добавлены[9] в `crefal` простейшие механизмы оптимизации возвращаемого `rasl`-кода и устранены некоторые шероховатости этого кода.

Нетрудно заметить, что при более тонком анализе область применения описанных выше оптимизаций, проводимых компилятором `refc`, может быть расширена.

Рассмотрим, например, описанное выше ограничение на применение оператора `TPL` переноса значений переменных из образцов. Одна из причин этого ограничения состоит в том, что вызов функции может разрушить собственный аргумент, если `rasl`-код вызываемой функции также использует оператор `TPL`.

Пример 1.1.

```
$ENTRY F {
 e.x (e.y) A = e.y;
 e.x (e.y) B, <G e.x>: True = e.x e.y;
}
```

[8]Для разных реализаций Рефала соответствующие алгоритмы могут существенно отличаться и зависеть от свойств конкретного интерпретатора Рефала.

[9]К существующим в компиляторе `refc`.

```
G {
   = True;
 A = False;
 t.z e.u = <G e.u>;
}
```

Функция G из примера 1.1 разрушает значение своего аргумента e.x. После окончания вычисления вызова <G e.x_0> при любом конкретном выражении e.x_0 структура, представляющая выражение e.x_0 в поле зрения Рефал-машины, будет заменена на пустое выражение или составной символ A, поскольку каждый шаг вычисления этого вызова убирает из поля зрения первый терм своего аргумента.

Предположим, что значение аргумента вызова <G e.x_0> было перенесено[10] из входного аргумента вызова функции F и Рефал-машина запомнила новый $\mathcal{J}$ адрес выражения e.x_0 с целью его дальнейшего использования[11] для построения правой части ρ второго предложения функции F.

Использование выражения *expr*, представленного в структуре Рефал-машины по адресу $\mathcal{J}$, для построения ρ, приведет к ошибке, даже если в момент построения ρ выражение *expr* будет доступно по этому адресу[12], поскольку вычисление вызова <G e.x_0> приведет к изменению значения его аргумента.

Рассмотрим другой пример.

Пример 1.2.

```
$ENTRY F {
 e.x (e.y) A = e.y;
 e.x (e.y) B, <G e.x>: True = e.y;
/* e.w = e.w; */
}
```

где функция G определена в предыдущем примере 1.1.

В этом примере правая часть второго предложения не использует значения переменной e.x. Как следствие, в данном случае компилятор crefal строит rasl-код, который использует оператор переноса TPL для построения выражения <G e.x> в конструкции условия второго предложения функции F.

Попробуем снять комментарии с последней строки определения F и вычислить вызов <F A (D) B>. В этом случае указанный выше перенос значения переменной e.x снова привел бы к ошибке, поскольку переменная e.w примет в качестве значения ошибочное выражение (D) B. По определению, значение переменной e.w должно равняться выражению A (D) B.

Описанную ситуацию можно наблюдать и при трассировке вычисления вызова <F A (D) B>, когда третья строка программы закомментирована, а интерпретируемый rasl-код построен компилятором crefal. Читатель может сравнить показанные на рисунках 1, 2 сообщения Рефал-машины при попытке вычисления вызова <F A (D) B> функцией, определенной в примере 1.2.

[10] Посредством оператора TPL – без построения копии выражения e.x_0.

[11] Снова посредством оператора TPL.

[12] Что не очевидно и зависит от свойств конкретного Рефал-интерпретатора, т. к. вызов <G e.x_0> завершил свою работу и память, отведенная для него, должна быть освобождена Рефал-машиной.

Наблюдаемое на рис. 2 свойство `rasl`-кода затрудняет отладку исходной программы, но уменьшает время ее исполнения. В исходных текстах компилятора `crefal` предусмотрена возможность выбора между отладочной и окончательной[13] трансляциями программы. В данной версии компилятора `crefal` эта возможность недоступна для стороннего пользователя.

```
Refal system Error: Recognition impossible.
.......................
*** Active function: F$1
*** Active expression:
<F A (D) B>

*** The View Field:
<STOP$ <Prout <F A (D) B>>>
```

Рис. 1. Сообщение Рефал-машины при попытке интерпретации `rasl`-кода, построенного компилятором `refc`.

```
Refal system Error: Recognition impossible.
.......................
*** Active function: F$1
*** Active expression:
<F (D) B <F$1 False>>

*** The View Field:
<STOP$ <Prout <F (D) B <F$1 False>>>>
```

Рис. 2. Сообщение Рефал-машины при попытке интерпретации `rasl`-кода, построенного компилятором `crefal`.

Полезно оценить времена вычислений функции `F` в следующих двух моделях интерпретации: (1) при использовании `rasl`-оператора переноса `TPL`, (2) когда вместо оператора `TPL` используется `rasl`-оператор копирования значения переменных выражений оператора `TPL`.

За логическую единицу времени примем время представления/построения в поле зрения Рефал-машины синтаксической единицы[14] Рефал-выражения, пренебрегая другими накладными расходами. Будем считать, что отождествление происходит во втором предложении функции `F`.

В первой модели интерпретации время вычисления вызова `<F` d_1 `(`d_2`) B>`, где d_1, d_2 суть произвольные строки символов, равномерно ограничено по длине входной строки d_2. Это время линейно зависит от длины входной строки d_1 (обозначим эту длину n), так как при вычислении функции `G` время построения синтаксических единиц равно $3n + 1$. Три синтаксических единицы `<, G,`

[13] Для исполнения транслируемой программы.

[14] Примеры синтаксических единиц: идентификатор, составной символ, левая структурная скобка, правая структурная скобка, левая скобка вызова функции, правая скобка вызова функции.

> строятся на каждом шаге рекурсии и одна – при выходе из рекурсии. Построения структур значений переменной `e.u` не происходит. Они уже существуют в поле зрения Рефал-машины и можно только проставить указывающие на них ссылки, что и делает оператор `TPL`.

Во второй модели интерпретации время вычисления того же самого вызова линейно зависит от длины входной строки d_1, так как заново будет построена ее копия. На каждом шаге рекурсии функции `G` будет копироваться значение переменной `e.u`, соответствующее этому шагу. Каждый шаг уменьшает длину значения переменной `e.u` на единицу. Складываем времена построения синтаксических единиц при вычислении функции `G`:

$$3n + ((n-1) + (n-2) + \ldots + 1) + 1 = n^2 + 3n + 1$$

Вывод: во второй модели интерпретации время вычисления вызова `<F` d_1 `(`d_2`) B>` имеет худший, по сравнению с первой моделью, порядок по длине входной строки d_1.

1.2. Сообщения об ошибках и предупреждениях. К сожалению, компилятор `refc` может как выдавать слишком не точные сообщения об ошибках, так и указывать неверное место их расположения.

Рассмотрим простейший пример Рефал-программы `FibWords`, содержащей синтаксические ошибки.

Пример 1.3.

```
$ENTRY FibWords {
    = 'a';
 'I = 'b';
 'II' e.n = <FibWords e.m> <FibWords I e.n;
 ( = <Prout "Wrong argument.>;
}
```

Листинг, порождаемый компилятором `refc`, данный на следующей странице (рис. А), неадекватен синтаксису программы:

- неверно сообщены номера строк, в которых отсутствуют закрывающие кавычки;
- информация о типе этих кавычек потеряна;
- в сообщении не указаны типы скобок, баланс которых нарушен (см. строки 4, 5 исходной программы);
- последние две строки листинга сообщают о несуществующих ошибках.

Кроме того, нет никакого указания на локализацию позиции ошибок в строках, где эти ошибки найдены.

Для выбранной нами тестовой программы `FibWords` опытный пользователь легко разберется с уточнением ошибок. В реальной программе, Рефал-предложения в которой не столь короткие, от программиста потребуется время на размышление, чтобы исправить ошибки баланса кавычек и скобок.[15]

[15] Конечно, если он не использует внешних инструментов подсчета этих скобок.

```
    1:   $ENTRY FibWords {
    2:       = 'a';
    3:    'I = 'b';
    4:    'II' e.n = <FibWords e.m> <FibWords I e.n;
Error:  6. No closing quote  on line     4.
Error: 10. Undefined variable in the right side: e.m  on line     4.
Error: 12. Unbalanced brackets in expression  on line     4.
    5:    ( = <Prout "Wrong argument.>;
Error: 12. Unbalanced brackets in expression  on line     5.
    6:   }
Error:  6. No closing quote  on line     6.
Error: 12. Unbalanced brackets in expression  on line     6.
Error: 102. Expected '}'  on line     6.
```

Рис. A. Листинг, порождаемый компилятором `refc` при трансляции программы `FibWords`.

```
    1:   $ENTRY FibWords {
    2:       = 'a';
    3:    'I = 'b';
Error on line 3: No closing quote ' in ;
    4:    'II' e.n = <FibWords e.m> <FibWords I e.n;
Error on line 4: Undefined variable e.m  in an expression
Error on line 4: Unbalanced left angle bracket starting
                 on line 4 in <FibWords I e.n ) ::: expected '>'
    5:    ( = <Prout "Wrong argument.>;
Error on line 5: Illegal character '=' (61)
Error on line 5: No closing quote " in Wrong argument.>;
    6:   }
Error on line 6: Unbalanced left angle bracket starting
                 on line 5 in <Prout Wrong argument.>; )
                 ::: expected '>'
Error on line 6: Unbalanced left parenthesis starting
                 on line 5 (Prout Wrong argument.>;
                 ::: expected ')'
crefal: 7 errors found in function FibWords
crefal: 7 syntax errors found.
```

Рис. B. Листинг, порождаемый компилятором `crefal` при трансляции программы `FibWords`.

На предыдущей странице (рис. B) также показан листинг синтаксического анализа программы `FibWords`, построенный компилятором `crefal`. Обратим внимание на локализацию найденных ошибок в строках и Рефал-предложениях программы.

Например:

- первое сообщение говорит о том, что в строке 3 третья одинарная кавычка не закрыта и она открывает пустую строку символов (первая кавычка закрыта второй, образуя строку из четырех символов: `'I = '`);
- первое сообщение о строке 6 говорит, что не найдена закрывающая скобка вызова функции `Prout`, начинающегося на строке 5: угловая скобка `'>'` на строке 5 включена в составной символ, имя которого открывает незакрытая двойная кавычка.

Приведем еще один пример программы с синтаксическими ошибками и листингами анализа этой программы двумя сравниваемыми нами компиляторами.

Пример 1.4.

```
$EXTRN G, F1, G1;

$ENTRY F {
 e.x (e.y) e.z 'aa' = <F C (<F1 (A <G1 (B )> )>>;
}
```

```
    1:   $EXTRN G, F1, G1;
    2:
    3:   $ENTRY F {
    4:    e.x (e.y) e.z 'aa' = <F C (<F1 (A <G1 (B )> )>>;
Error: 111. Unbalanced < and >  on line     4.
    5:   }
```

Рис. C. Листинг, порождаемый компилятором `refc` при трансляции программы `F`.

```
    1:   $EXTRN G, F1, G1;
    2:
    3:   $ENTRY F {
    4:    e.x (e.y) e.z 'aa' = <F C (<F1 (A <G1 (B )> )>>;
Error on line 4: Unbalanced left parenthesis starting
                 on line 4 (<F1 (A <G1 (B )> )>  ::: expected ')'
Error on line 4: Unbalanced right angle bracket starting with >;
Error on line 4: Unbalanced left angle bracket starting
                 on line 4 in <F C <F1 (A <G1 (B )> )> )
                 ::: expected '>'
    5:   }
crefal: 3 errors found in function F
crefal: 3 syntax errors found.
```

Рис. D. Листинг, порождаемый компилятором `crefal` при трансляции программы `F`.

§ 2. О тестировании компилятора `crefal`

После первого этапа построения компилятора `crefal` (см. п. 1.1) `rasl`-код, который он порождает при трансляции Рефал-программы `p`, совпадает с `rasl`-кодом для этой программы `p`, порождаемым компилятором `refc`.

Это свойство позволило нам автоматизировать тестирование версии компилятора `crefal`, полученной на первом этапе разработки.

На Рефале-5 была написана функция `RandomPrg`, генерирующая случайную Рефал-5 программу, не содержащую синтаксических ошибок. Вызовы функции `RandomPrg` вычислялись в цикле, который последовательно запускал трансляцию программы, построенную функцией `RandomPrg`, компиляторами `refc` и `crefal`. Эти два результата трансляции сравнивались побайтно. Цикл прекращал работу, если эти результаты не совпадали, и выдавал тест на ошибку.

В исходных текстах компилятора `crefal` эти два этапа разработки компилятора `crefal` (см. п. 1.1) композиционно разделены с целью облегчения поиска ошибок, которые могут обнаружиться в будущем.

§ 3. Заключение

Требуется дальнейшая содержательная работа по оптимизации `rasl`-кода, возвращаемого компилятором `crefal`. В частности, реализация в компиляторе `crefal` некоторых алгоритмов оптимизации Рефал-программ, используемых в суперкомпиляторе SCP4 [7, 6].

Мы также надеемся построить удобный *формальный* язык сообщений о синтаксических ошибках в транслируемых программах. Фиксация такого языка позволит анализировать эти сообщения программами, написанными пользователями Рефала-5.

Список литературы

[1] *Базисный Рефал и его реализация на вычислительных машинах*, ЦНИПИАСС, Москва, 1977.

[2] Е. А. Гайдар, И. М. Игнатович, В. Ф. Козадой, А. П. Немытых, В. А. Пинчук, С. В. Чмутов, "Реализация системы программирования FLAC", *Сборник трудов по функциональному языку программирования Рефал*, **I**, Издательство Сборник, Переславль-Залесский, 2014(1988), 43–91.

[3] К. Гао, А. П. Немытых, *CREFAL: компилятор Рефала-5 в язык сборки*, ([online]: http://www.botik.ru/pub/local/scp/refal5/), 2004.

[4] Р. Ф. Гурин, С. А. Романенко, *Язык программирования РЕФАЛ ПЛЮС*, ИНТЕРТЕХ, Москва, 1991.

[5] Арк. В. Климов, *Рефал-6* http://www.refal.org/~arklimov/refal6, 2003.

[6] А. П. Немытых, В. Ф. Турчин, *Суперкомпилятор SCP4: исходные тексты, on-line демонстрация*, ([online]: http://www.botik.ru/pub/local/scp/refal5/), 2000.

[7] А. П. Немытых, *Суперкомпилятор SCP4: общая структура*, ISBN 978-5-382-00365-8, Издательство УРСС, Москва, 2007.

[8] А. П. Немытых, *Рефал-5 на Windows Mobile 5.0: исполняемый модуль интерпретатора*, ([online]: http://www.botik.ru/pub/local/scp/refal5/), 2008.

[9] А. П. Немытых, *Парсер Рефала-5, написанный на Рефале-5: rsl-модуль*, ([online]: http://www.botik.ru/pub/local/scp/refal5/), 2009.

[10] А. П. Немытых, "Лекции по языку программирования Рефал", *Сборник трудов по функциональному языку программирования Рефал*, **I**, Издательство Сборник, Переславль-Залесский, 2014, 118–165.

[11] А. П. Немытых, "Заметка о переносе реализации Рефала-5 на операционную систему Windows Mobile 5.0", *Сборник трудов по функциональному языку программирования Рефал*, **I**, Издательство Сборник, Переславль-Залесский, 2014, 166–169.

[12] С. А. Романенко, "Реализация Рефала-2", Препринт, ИПМ: им. М. В. Келдыша АН СССР, Москва, 1987.

[13] В. Ф. Турчин, "Метаязык для формального описания алгоритмических языков", *Цифровая вычислительная техника и программирование*, Советское радио, Москва, 1966, 116–119.

[14] В. Ф. Турчин, *Программирование на языке РЕФАЛ*, Препринт №43, ИПМ АН СССР, 1971.

[15] В. Ф. Турчин, *РЕФАЛ-5. Руководство по программированию и справочник*, ([online]: http://www.refal.org/rf5_frm.htm русский перевод *устаревшей* версии книги [19].).

[16] В. Ф. Турчин, Д. В. Турчин, А. П. Конышев, А. П. Немытых, *Рефал-5: исполняемые модули и исходные тексты*, ([online]: http://www.botik.ru/pub/local/scp/refal5/), 2000.

[17] S. V. Chmutov, E. A. Gaydar, I. M. Ignatovich, V. F. Kozadoy, A. P. Nemytykh, V. A. Pinchuk, "Implementation of the symbol analytic transformation language FLAC", *LNCS*, **429** (1990), 276.

[18] V. F. Turchin, "The concept of a supercompiler", *ACM Transactions on Programming Languages and Systems*, **8** (1986), 292–325.

[19] V. F. Turchin, *REFAL-5 programming guide and reference manual*, (переработанное и расширенное издание 1999 года доступно как zipped html-файл: http://refal.botik.ru/book/refal-book-html.zip), New England Publishing Co., Holyoke, 1989.

Си Гао (Xi Gao)
Университет города Ухань, г. Ухань, Китай
E-mail: gaoxi@gliet.edu.cn

А. П. Немытых (A. P. Nemytykh)
Переславль-Залесский, ИПС РАН
E-mail: nemytykh@math.botik.ru

А. П. Немытых

Заметка о развитии библиотеки встроенных функций языка программирования Рефал-5

В заметке описываются новые возможности библиотеки встроенных функций Рефала-5, т. е. реализованные после публикации монографии В. Ф. Турчина «Refal-5: programming guide and reference manual» [10].

Библ. 10 наим.

Ключевые слова: Рефал, метавычисления.

Содержание

Работа выполнена в 2006 г.

Библиотека Рефала-5 состоит из двух частей.

Первая часть состоит из функций, написанных на языке реализации системы Рефала-5 – в данном случае на языке программирования «Си», и доступных по умолчанию из любой Рефал-5 программы.

Вторая часть библиотеки состоит из функций, написанных непосредственно на языке Рефал-5 и поставляемых в библиотечном модуле `reflib.ref`, который необходимо загрузить при использовании функций из этого модуля. В модулях, использующих данные функции, имена этих функций должны быть перечислены в списке имен внешних функций посредством объявления `$EXTERN`.

§ 1. О встроенных функциях, написанных на языке реализации Рефала-5

В этом разделе мы рассмотрим новые возможности встроенных функций системы Рефал-5, написанных непосредственно на языке реализации интерпретатора Рефала-5 – языке программирования «Си».

1.1. Операции ввода-вывода. Одним из аргументов функций[1] доступа к файлам операционной системы является дескриптор файла, с которым будет работать соответствующая функция. Дескриптор файла является уникальной меткой, которая присваивается файлу в момент его открытия или создания. Все последующие действия с файлом могут происходить только через эту метку. После закрытия файла его дескриптор становится свободным для присвоения другим файлам. В момент завершения или прерывания работы интерпретатора все открытые файлы автоматически закрываются.

Дескриптором файла может являться макро-цифра от 1 до 39 включительно. Т. е. диапазон дескрипторов файлов расширен. В некоторых операциях ввода-вывода также допускается использования дескриптора 0 для стандартного ввода; в этом случае действий открытия стандартного ввода производить не нужно.

Операция `Open`. Область допустимых значений первого аргумента `s.mode` функции открытия или создания файла

```
<Open s.mode s.file-descriptor e.file-name>
```

расширена и зависит от языка реализации интерпретатора Рефала-5.

Базисные допустимые значения `s.mode` суть следующие одно-буквенные строки: `'w'`, `'W'` (открыть файл `e.file-name` для записи), `'r'`, `'R'` (открыть для чтения), `'a'`, `'A'` (открыть для добавления).

В действительности файл может быть открыт для любого режима действий с файлами, поддерживаемого в языке реализации.

Также допускается использование составного символа, определяющего режим доступа к файлу. Например, значение `"wb"` аргумента `s.mode` указывает на то, что файл будет открыт для бинарной записи.

Имя файла `e.file-name` может быть пустым выражением. В этом случае, если файл открывается для записи/добавления, то операция `Open` произведет

[1] Которые, конечно, не являются функциями в строгом математическом смысле.

попытку открытия/создания файла REFALdescr.DAT, где descr есть десятичное представление дескриптора s.file-descriptor. Попытка открыть файл с пустым именем для чтения приведет к ошибке исполнения.

Операция Write. Результат новой операции записи в файл

```
<Write s.file-descriptor e.expression>
```

отличается от результата вывода Putout только в следующем: Write не добавляет символа новой строки непосредственно после вывода выражения e.expression, который вставляет Putout (см. описание этой функции в [10]).

1.2. Функции арифметики. Напомним, что в диалекте Рефал-5 целые числа представляются знаком числа и последовательностью макро-цифр в системе счисления по основанию 2^{32}.[2] Отрицательные числа начинаются с символа '-'. Символ '+' перед неотрицательными числами может быть опущен.

Встроенные арифметические функции возвращают неотрицательные числа без знака и лидирующих нулей.

К ранее существовавшим арифметическим функциям добавлены нижеследующие операции.

Операция Random.

```
<Random e.macrodigit> :: e.integer
```

где e.macrodigit есть либо s.macrodigit, либо '+' s.macrodigit.

Вызов операции <Random e.macrodigits> возвращает неотрицательное случайное число, число макро-цифр в котором меньше либо равно значению аргумента.

Операция RandomDigit.

```
<RandomDigit e.macrodigit> :: s.macrodigit
```

где аргумент e.macrodigit может принимать значения, аналогичные аргументу операции Random, возвращает неотрицательную случайную макро-цифру, меньшую либо равную значению аргумента.

1.3. Системные функции. Под системными функциями мы понимаем операции взаимодействия с операционной системой и опроса технических свойств реализации интерпретатора системы программирования Рефал-5.

Добавлены нижеследующие возможности общения с операционной системой.

Запрос GetPID.

```
<GetPID> :: s.macrodigit
```

возвращает уникальный идентификационный номер процесса текущей интерпретации Рефал-5 программы, исполняемой под данной операционной системой.

Запрос GetPPID.

```
<GetPPID> :: s.macrodigit
```

возвращает

- идентификационный номер процесса, для которого текущий процесс интерпретации Рефал-5 программы является непосредственным потомком («сыном»), исполняемым под данной операционной системой, если исполнение происходит под операционной системой семейства Linux (Unix);

[2] Точнее в системе счисления по основанию 2^N, где N – длина машинного слова.

- функция не определена, если интерпретация происходит под операционной системой семейства Windows.[3]

Функция `ListOfBuiltin`.

`<ListOfBuiltin> :: e.built-in-functions-info`

возвращает последовательность выражений в скобках. Каждое такое выражение содержит информацию о встроенной функции языка Рефал-5, написанной на языке программирования «Си».

Более точно:

```
e.built-in-functions-info
        ::= t.finfo e.built-in-functions-info | EMPTY
t.finfo ::= (s.fnumber s.built-in-function-name s.ftype)
s.ftype ::= regular | special
s.fnumber ::= s.macrodigit
EMPTY ::= пустое выражение
```

где `s.fnumber` есть дескриптор встроенной функции, присваиваемый интерпретатором Рефала-5. Данный дескриптор зависит только от реализации интерпретатора и не зависит от процесса конкретного исполнения этого интерпретатора.

Отметим также, что здесь `regular` и `special` суть данные – составные символы Рефала-5.

Функция `ListOfBuiltin` может использоваться для реализации некоторых функций анализа Рефал-программ, не зависящих от конкретной версии интерпретатора языка программирования Рефал-5.

Функция `SizeOf`.

`<SizeOf s.character> :: s.macrodigit`

возвращает размер базисного[4] типа данных в языке реализации интерпретатора языка программирования Рефал-5.[5] Этот размер равен числу единиц типа данных `char`, которые используются для представления типа данных, код которого задан значением аргумента `s.character`.

Следовательно, `<SizeOf 'c'> = 1`, где `'c'` есть Рефал-код для типа данных `char` языка программирования «Си».

Еще один пример: `<SizeOf 'l'> = 4`. Здесь `'l'` есть Рефал-код для типа данных `long` языка программирования «Си».

Функция `SizeOf` может использоваться для тех же целей, что и функция `ListOfBuiltin`.

§ 2. О развитии библиотеки `reflib.ref`

Другая часть стандартных функций Рефала-5 написана на самом Рефале-5. Определения этих функций можно найти в модуле `reflib.ref`.

Функции из модуля `reflib.ref`, объявленные как `$ENTRY`, можно использовать как функции из любого обычного Рефал-модуля. Т. е. модуль `reflib.ref`

[3] Фактически под Windows эта функция совпадает с функцией `GetPID`.

[4] Т. е. определенного в самом языке, а не в программе написанной на языке.

[5] Т. е. в языке программирования «Си».

должен быть загружен (подлинкован) к пользовательским модулям, если пользователь использует функции из `reflib.ref`.

Например,

`refgo program`$_1$`+program`$_2$`+reflib`

К функциям из этой части библиотеки, ранее описанным В. Ф. Турчиным в [10], добавлены нижеследующие функции.

Функция `Pair`. Результатом преобразования входной строки `e.string` посредством функции `Pair` является Рефал-выражение `e.expression`.

```
<Pair e.string> :: e.expression
```

- если строка `e.string` содержит символы `'('` и `')'` и эти символы-скобки сбалансированы, тогда функция заменяет все символы-скобки на соответствующие структурные скобки Рефала, оставляя неизменной остальную часть своего аргумента;
- если баланс скобок-символов нарушен во входном аргументе, тогда результат функции есть пустое выражение; и функция печатает сообщение об ошибке.

Функция `PairArg`.

```
<PairArg s.macrodigit> :: e.expression
```

возвращает аргумент командной строки[6] с порядковым номером `s.macrodigit`, преобразованный по правилу, указанному выше для аргумента функции `Pair`.

Например, если командная строка запуска исполняемого модуля интерпретатора была

```
refgo program+reflib abc (123)z
```

тогда результатом вызова `<PairArg 2>` в программе `program` будет Рефал-выражение `('123') 'z'`.

Заметим, что аргументы командной строки не должны начинаться с символа `'-'`, который является началом аргумента исполняемого модуля интерпретатора, а не программы, которую он интерпретирует.

Функция `Input`.

```
<Input e.channel> :: e.expression
```

преобразует («парсирует») первую строку из канала `e.channel` в Рефал-выражение. Правила преобразования описаны в монографии В. Ф. Турчина (см. [10; п. 2.3 Input-Output]).

Конец строки определяется вводом в канал пустой строки или символа конца файла. Последующий вызов функции `Input` с тем же самым аргументом преобразует следующую строку в канале и т. д. Здесь `e.channel` есть либо дескриптор файла (в данном случае макро-цифра от 0 до 19), либо имя файла.

Если в преобразуемой строке найдена синтаксическая ошибка, то функция возвращает пустое выражение и сообщает об ошибке.

Функция `InputArg`.

```
<InputArg s.macrodigit> :: e.expression
```

преобразует аргумент командной строки интерпретатора с порядковым номером `s.macrodigit` аналогично функции `Input`.

[6] Исполняемых модулей `refgo`, `reftr`.

Операция `Pprout`.

```
<Pprout e.expression> ::
```

выводит свой аргумент в отформатированном виде в канал стандартного вывода. Результат вызова – пустое выражение. Описание формата вывода можно найти в [10; п. 6.7].

Операции `Xxout, Xxinr, Xxin`. Описание операций `Xxout, Xxinr, Xxin` даны в монографии В. Ф. Турчина (см. [10; п. 2.3 Input-Output]).

§ 3. Метавычисления

Рефал разрабатывался В. Ф. Турчиным как метаязык.

Первая статья Турчина, в которой были изложены основные идеи разработки Рефала, была опубликована в 1966 году и называлась «Метаязык для формального описания алгоритмических языков» [5].

К сожалению, часть из этих идей Турчина никогда не были реализованы. Однако язык программирования Рефал ориентирован на поддержку метавычислений и используется для разработки автоматических инструментов компиляции, оптимизации и преобразования программ. В частности, на Рефале написаны компиляторы и оптимизаторы программ на Рефале.

Первая задача, которую нужно решать при анализе программ – это задача представления (кодирования) анализируемой программы в терминах языка реализации анализатора. Обычно такой программный инструмент называют «парсером». При преподавании элементов преобразования программ возникает проблема постановки задач для самостоятельного решения. Чтобы приступить к решению упражнения по-существу, студент вынужден реализовать «парсер», что делает задачу слишком трудоемкой и неприемлемой в качестве упражнения. Для решения упражнения по-существу студенту нужно предоставить возможность использования готового инструмента кодирования программ.

С этими целями был реализован на Рефале-5 «парсер» программ, написанных на Рефале-5, который представляет собой отдельный Рефал-модуль `prefal.rsl` [4].

Функция `prefal`.

```
<prefal e.fname> :: e.encoded-module
```

преобразует Рефал-5 программу, данную как последовательность строк в файле `e.fname`, в Рефал-выражение `e.encoded-module`.

В отличие от исходной программы, результат преобразования дан в жестких структурах и является объектом, синтаксические единицы которого можно обнаружить простейшей Рефал-программой.

Комментарии исходной программы не удаляются и не преобразуются.[7] Они обрамляются структурными скобками вида `(Comment e.chars-comment)`, которые могут встретиться в любом месте результата парсирования.

[7] Т. е. «парсер» можно использовать, например, для автоматического форматирования программ.

В данной версии «парсера» сообщения о синтаксических ошибках в преобразуемой программе выводятся на печать. Т. е. они не входят в результат парсирования и не могут быть непосредственно обработаны программой, которая вызвала «парсер».

Перед тем как дать определение структуры выражения `e.encoded-module`, напомним, что для использования «парсера» в программу-анализатор `pgm` необходимо вставить объявление

```
$EXTERN prefal;
```

и загрузить программу парсирования.

Если Рефал-модуль `prefal.rsl` находится в вашем рабочем каталоге, то командная строка запуска Рефал-интерпретатора может выглядеть следующим образом:

```
refgo pgm+prefal
```

Определение структуры кодировки. Структуры, соответствующие комментариям, в определении опущены.

```
e.encoded-module ::= t.interface e.funcs
e.funcs  ::= t.fbody+
t.fbody  ::= (t.ftype t.fname t.sentence+)
    t.ftype ::= (s.ftype e.comments)
    s.ftype ::= ENTRY | LOCAL
    t.fname ::= (s.fname e.comments)
    t.sentence  ::= ((e.left-side) '=' (e.right-side))
                  | (e.left-side e.end)
    e.left-side ::= e.patt t.condition*
    t.condition ::= (Condition t.expr (Pattern e.patt))
    e.right-side ::= e.expr
    e.end ::= (Condition t.expr (Block t.sentence+))
          t.expr ::= (Expression e.expr)
          e.patt ::= t.term e.patt | (Bracket e.patt) e.patt
                   | EMPTY
          e.expr ::= t.term e.expr | (Bracket e.expr) e.expr
                   | t.fcall e.expr | EMPTY
          t.term ::= t.variable | t.number | t.symbol
          t.symbol ::= t.word | s.char
          t.word ::= (Word WORD_NAME)
          s.char ::= CHARACTER
          t.number ::= (MacroDigit MACRO_DIGIT)
          t.variable ::= (Variable s.vtype VARIABLE_NAME)
               s.vtype ::= 's' | 't' | 'e'
          t.fcall ::= (Call s.fname e.expr)
               s.fname ::= FUNCTION_NAME

t.interface ::= t.extern
    t.extern ::= (EXTERN s.fname*)
```

```
EMPTY ::= пустое выражение
CHARACTER ::= символ
WORD_NAME ::= составной символ
MACRO_DIGIT ::= макро-цифра
VARIABLE_NAME ::= составной символ
FUNCTION_NAME ::= идентификатор
```

Список литературы

[1] К. Гао, А. П. Немытых, *CREFAL: компилятор Рефала-5 в язык сборки*, ([online]: http://www.botik.ru/pub/local/scp/refal5/), 2004.

[2] А. П. Немытых, В. Ф. Турчин, *Суперкомпилятор SCP4: исходные тексты, on-line демонстрация*, ([online]: http://www.botik.ru/pub/local/scp/refal5/), 2000.

[3] А. П. Немытых, *Суперкомпилятор SCP4: общая структура*, ISBN 978-5-382-00365-8, Издательство УРСС, Москва, 2007.

[4] А. П. Немытых, *Парсер Рефала-5, написанный на Рефале-5: rsl-модуль*, ([online]: http://www.botik.ru/pub/local/scp/refal5/), 2009.

[5] В. Ф. Турчин, "Метаязык для формального описания алгоритмических языков", *Цифровая вычислительная техника и программирование*, Советское радио, Москва, 1966, 116–119.

[6] В. Ф. Турчин, *Программирование на языке РЕФАЛ*, Препринт №43, ИПМ АН СССР, 1971.

[7] В. Ф. Турчин, *РЕФАЛ-5. Руководство по программированию и справочник*, ([online]: http://www.refal.org/rf5_frm.htm русский перевод *устаревшей* версии книги [10].).

[8] В. Ф. Турчин, Д. В. Турчин, А. П. Конышев, А. П. Немытых, *Рефал-5: исполняемые модули и исходные тексты*, ([online]: http://www.botik.ru/pub/local/scp/refal5/), 2000.

[9] V. F. Turchin, "The concept of a supercompiler", *ACM Transactions on Programming Languages and Systems*, **8** (1986), 292–325.

[10] V. F. Turchin, *REFAL-5 programming guide and reference manual*, (переработанное и расширенное издание 1999 года доступно как zipped html-файл: http://refal.botik.ru/book/refal-book-html.zip), New England Publishing Co., Holyoke, 1989.

А. П. Немытых (A. P. Nemytykh)
Переславль-Залесский, ИПС РАН
E-mail: nemytykh@math.botik.ru

А. П. Немытых

О некоторых понятиях суперкомпиляции – метода специализации программ

В данных заметках автор начинает описывать свое понимание современного состояния развития идей В. Ф. Турчина в области автоматического преобразования программ, известных как суперкомпиляция. К сожалению, на данный момент существует путаница даже в базовых понятиях суперкомпиляции. Такое печальное положение отчасти связано с попытками наводить наукоподобие там, где простыми словами всё можно объяснить хорошему школьнику; есть и другие причины.

Автор основывается на опыте построения суперкомпиляторов SCP3 [13] и SCP4 [7]; в реализации первого автор принимал участие совместно с В. Ф. Турчиным, второй разрабатывался и реализовывался под научным руководством Валентина Фёдоровича.

Библ. 18 наим.

Ключевые слова: Рефал, суперкомпиляция, оптимизация, специализация программ, метавычисления.

Содержание

Статья написана в 2011 г.

§ 1. О постановке задач на специализацию программ

Рассмотрим программу `P`.

С точки зрения разработчика, `P` состоит из набора подпрограмм (функций и/или процедур) и входной точки (в языке Си она называется `main`, в языке Рефал-5 – `Go`), которая, грубо говоря, находится в общем положении: то есть, аргументы входной точки могут быть произвольны. Например, `<Go e.x>`. Здесь `e.x` – «переменная» (см. уточнение ниже), которая может принять в качестве значения любое данное языка Рефал, а угловые скобки означают вызов функции.

Разработчик реализует программу для широкого круга пользователей.

С точки зрения конкретного пользователя, входная точка программы является полностью фиксированной. Чтобы запустить `P`, пользователь должен задать конкретные значения аргументов входной точки. В нашем случае: $\texttt{<Go e.x}_0\texttt{>}$.

Специализатор программ находится между разработчиком и пользователем. В первом приближении это означает, что входная точка программы `P` частично задана, а частично неизвестна. Например, `<Go ('string') e.x>`.

Наша оговорка «в первом приближении» существенна: здесь есть много тонкостей, некоторые из которых мы рассмотрим в этой заметке; к более сложным, возможно, вернёмся позже. Однако принципиальный момент мы уже отметили: для того чтобы запустить суперкомпилятор, недостаточно подать ему на вход преобразуемую программу; нужно еще поставить задачу специализации этой программы. Язык постановки таких задач В. Ф. Турчин назвал языком MST-схем (см., например, [18, 7]).

Обратим внимание на следующее: наше описание задачи

```
<Go ('string') e.x>
```

не вполне корректно. Любой язык программирования предполагает, что переменная принимает своё значение в процессе интерпретации программы. Мы же предполагаем, что `e.x` уже заданное входное значение, но для нас оно неизвестно. Если читатель когда-то в школе решал квадратные уравнения с параметром, то он вспомнит, что переменные в процессе решения такого уравнения также должны принять какие-то значения, а значения параметров уравнения, хотя и заданы, но неизвестны школьнику: он вынужден рассматривать всевозможные значения параметров и в зависимости от их свойств выписывать решения уравнения. Разным параметрам могут соответствовать разные решения уравнения (значения переменных, при которых уравнение обращается в тождество).

В наших заметках мы ещё неоднократно будем обращаться к этому школьному примеру. Здесь же подытожим сказанное: суперкомпилятор должен уметь различать переменные и параметры. Первое уточнение нашей задачи на специализацию (MST-схемы):

```
<Go ('string') #e.x>
```

Мы ввели синтаксическое понятие параметра, обозначив его приставкой `#`.

Итак, для постановки задач на специализацию программ возникает необходимость в языке параметров. Как этот язык связан, например, с языком описания переменных в языке программирования, программы на котором мы

хотим специализировать? Выше, на примере, мы показали, как мог бы быть определен простейший язык параметров: каждый тип переменных определяет соответствующий тип параметров. Насколько выразителен этот простейший язык? Посредством его можно описывать несложные задачи специализации, но, например, нельзя *адекватно* сформулировать классическую задачу специализации интерпретатора по данной программе. Если рассмотреть аналог такого простейшего языка параметров в языке Haskell [14], тогда, например, нельзя сформулировать задачу доказательства факта конечности времени работы данной программы (написанной на языке Haskell) на любых её входных данных – как задачу специализации.

Выбор языка параметров критичен не только с точки зрения формулировки задач на специализацию, но и с точки зрения получения удовлетворительного результата специализации. Это нетрудно понять: любой специализатор, по существу, сводит стартовую задачу специализации к последовательности промежуточных задач специализации, которые также описываются посредством выбранного языка параметров.

Рассмотрим прямую интерпретацию языков программирования. Примером может служить оболочка операционной системы Linux (`shell`). В `shell` входной точкой (командой, запросом) является любая строка, удовлетворяющая определенным синтаксическим правилам. Т. е. язык описания входных точек намного более выразителен, чем в Си или Рефале. Например, его синтаксис позволяет описать вызов композиции функций. В языке MST-схем также можно описывать подобные синтаксические конструкции. MST-схема

```
<F #e.y <G #e.x>>
```

ставит следующую задачу специализации программы `P`, в которой определены функции `F` и `G`: проспециализировать `P`, если известно, что `P` будет использоваться только посредством вызова указанной выше композиции функций `G` и `F`.

О некоторых других тонкостях, связанных с постановкой задач на специализацию, неподготовленный читатель может ознакомиться, например, в статье [8]. На электронной странице [5] описано несколько простых примеров задач специализации, ознакомившись с которыми, читатель может просуперкомпилировать эти примеры непосредственно со следующей электронной страницы [5].

§ 2. О прогонке

Вернёмся к нашему основному примеру – квадратному уравнению.

Рассмотрим алгоритм его решения $\mathcal{U}$. Если коэффициенты уравнения полностью известны (являются действительными числами), то алгоритм $\mathcal{U}$ *однозначно определяет последовательность действий*, которая приведёт либо к вычислению действительных корней данного уравнения, либо к выяснению факта отсутствия таких корней (т. е. к состоянию «аварийной остановки»).

Другой школьный алгоритм ${}_{\mu}\mathcal{U}$, позволяющий решать квадратные уравнения с параметрами, является *метарасширением* алгоритма $\mathcal{U}$. Некоторые из

коэффициентов уравнения могут быть известны, а некоторые заданы параметрами: т. е. хотя и *заданы*, но неизвестны. Известно лишь, что это какие-то действительные числа, но неизвестно, какие именно. В соответствии с этим, в процессе решения квадратного уравнения с параметрами строится конечное дерево Г, узлы которого являются точками ветвления параметров (`if`, `case`). Ребра дерева Г помечены предикатами – условиями, уточняющими (сужающими) значения параметров таким образом, чтобы можно было однозначно проделать несколько действий первого алгоритма $\mathcal{U}$ равномерно по значениям параметров – до следующей точки ветвления. На листах дерева Г написаны параметризованные значения корней решаемого уравнения. Композиция последовательности предикатов от корня Г к листу определяет условие, при котором заданное уравнение имеет корни, помечающие этот лист.

Итак, результатом алгоритма ${}_{\mu}\mathcal{U}$ является помеченное ориентированное корневое дерево Г, и это дерево определено в терминах параметров.

Заметим, что (1) Г может оказаться пустым (при любых значениях параметров уравнение не имеет корней); (2) иначе корень этого дерева помечен уравнением (описанием постановки задачи).

Если условно считать, что школьник выполняет каждое арифметическое действие над действительными числами за единицу времени независимо ни от длины этих чисел, ни от других их свойств, тогда алгоритмы $\mathcal{U}$ и ${}_{\mu}\mathcal{U}$ можно описать, не используя программистского понятия цикла. Дерево Г всегда является конечным деревом.

Таким образом, о понятии *прогонки* читателю рассказывал в средней школе его учитель математики[1]; выше мы только напомнили читателю это понятие. Еще раз перескажем то же самое, но теперь уже в программистских терминах.

Пусть дан некоторый алгоритмически полный язык программирования $\mathcal{L}$. Алгоритм любого интерпретатора `Int` языка $\mathcal{L}$ описывается текстом конечной длины. Это простое замечание приводит нас к следующей структуре интерпретатора `Int`:

```
Int(p,d) {
   инициализация;
   while (вычисления не завершены)
         { STEP; }
   return (результирующее состояние);
}
```

Здесь `STEP` есть минимальный логически замкнутый шаг, включающий в себя все механизмы, необходимые для интерпретации любой $\mathcal{L}$-программы `p` на любых входных данных `d`. `STEP` изменяет глобальное состояние интерпретатора, которое и является аргументом `STEP`. В данной выше схеме этот аргумент явно не указан. Если физическое время T вычисления `STEP` равномерно ограничено по размеру его входных данных, тогда число шагов, необходимое для вычисления `p` на `d`, можно рассматривать как логическое время вычислений, которое *адекватно* отражает соответствующее физическое время вычислений.

[1] К счастью, не произнося само слово «прогонка»

В более технических терминах условие равномерной ограниченности T означает отсутствие в `STEP` проходов по кускам его входных данных, размер которых (кусков) неизвестен разработчику интерпретатора `Int`, но при каждом конкретном запуске `Int`, конечно, может быть известен пользователю.

Одним из базовых механизмов суперкомпиляции является прогонка данной параметризованной конфигурации (параметризованного состояния стека вызовов функций специализируемой $\mathcal{L}$-программы). Алгоритм прогонки является метарасширением интерпретации одного шага $\mathcal{L}$-машины, преобразующего данное конкретное состояние стека вызовов в следующее конкретное состояние стека вызовов. Результатом прогонки является конечное дерево возможных вычислений данного на входе прогонки параметризованного (т. е. частично неизвестного) стека вызовов функций. Ниже мы будем называть это дерево *гроздью прогонки*, чтобы отличать его от других деревьев, возникающих в процессе суперкомпиляции.

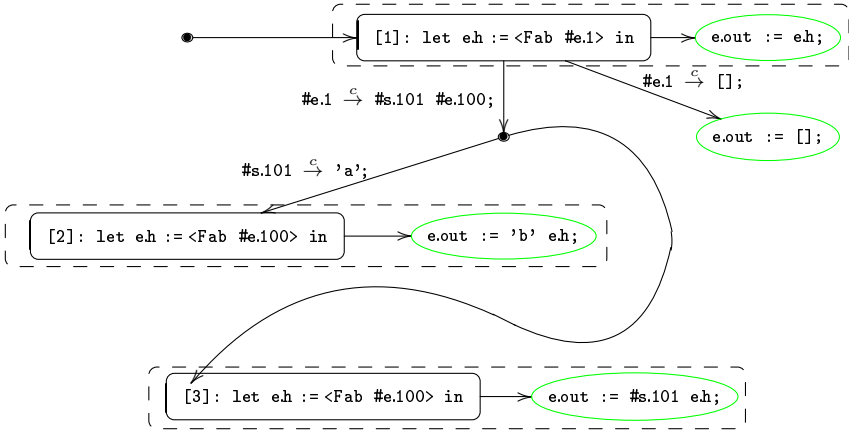


Рис. 1. Результат первого вызова прогонки.

На рис. 1 показан результат первого вызова алгоритма прогонки. Пунктирные кластеры показывают структуры параметризованных стеков в узлах и листьях грозди прогонки. Эллипсами отмечены полностью вычисленные параметризованные части стеков (т. е. в них нет вызовов функций), соответствующих пути вычисления из корня грозди в данную вершину грозди. Вершины грозди прогонки, обозначенные (жирными) точками, являются реальными точками ветвления, а не только удобной формой представления в грозди. Причины, по которым параметризованное состояние вычислительной системы не показано в вершинах-точках, будут рассмотрены в одной из последующих заметок. Стартовая параметризованная конфигурация, задача на суперкомпиляцию, являющаяся корнем метадерева возможных вычислений, выделена указанием на неё стрелки-ребра, в исходящей вершине которого нет никакой конфигурации. Ребра грозди прогонки Г помечены предикатами выбора/расщепления конкретных значений параметров (выбора конкретного шага Рефал-машины в момент исполнения преобразуемой программы).

Входными аргументами суперкомпилятора SCP4 здесь были нижеследующая Рефал-программа `p` и указанная в корне Г задача на специализацию программы `p` (т. е. `<Fab #e.x>`). В данном примере в постановке задачи нет никакой заданной информации, по которой можно было бы содержательно проспециализировать программу `p`.

```
Fab {
  'a' e.y = 'b' <Fab e.y>;
  s.x e.y = s.x <Fab e.y>;
   = ;
}
```

Условия выбора конкретных ветвей (рёбер) грозди прогонки всегда определены только в терминах параметров. Параметры являются семантическими объектами, а переменные программы `p` – синтаксическими объектами. Рёбра грозди, исходящие из конкретной вершины, упорядочены слева направо. Условие выбора (сужения параметров) конкретной ветви ρ: (1) ни одна ветвь, находящаяся слева от ρ, не выбрана; (2) параметры удовлетворяют условию, описанному в метке ребра ρ.

Метки рёбер суть последовательная композиция элементарных предикатов-сужений. Оператор композиции обозначен сиволом «точка с запятой». Например, сужение `#e.1` $\stackrel{c}{\rightarrow}$ `#s.101 #e.100;` означает, что параметр `#e.1` имеет вид `#s.101 #e.100`. Пустая метка обозначает тривиальное условие, которое всегда истинно.

Отметим, что, в отличие от интерпретации конкретного (не параметризованного) стека вызова функций, разложение синтаксической композиции вызовов функций в стек, линейную последовательность их исполнения, может быть нетривиальным и зависит от стратегии суперкомпиляции. Примерами таких стратегий могут быть «ленивое» (call-by-name) и аппликативное (call-by-value) вычисление в процессе суперкомпиляции.

Рисунок 2 показывает результат развития дерева возможных вычислений задачи на суперкомпиляцию (пары – преобразуемой программы и ее параметризованной входной точки): результат второго вызова алгоритма прогонки был подклеен вместо листа левой ветви грозди прогонки, построенной предыдущим вызовом прогонки.

В данной заметке мы вынуждены были обходить молчанием некоторые свойства грозди, построенной алгоритмом прогонки. Например, мы никак не пояснили: в чем заключается смысл выделения вершин-точек? Это и некоторые другие свойства грозди связаны с другими инструментами суперкомпиляции, которые используют данные свойства. Мы надеемся рассмотреть их в последующих заметках.

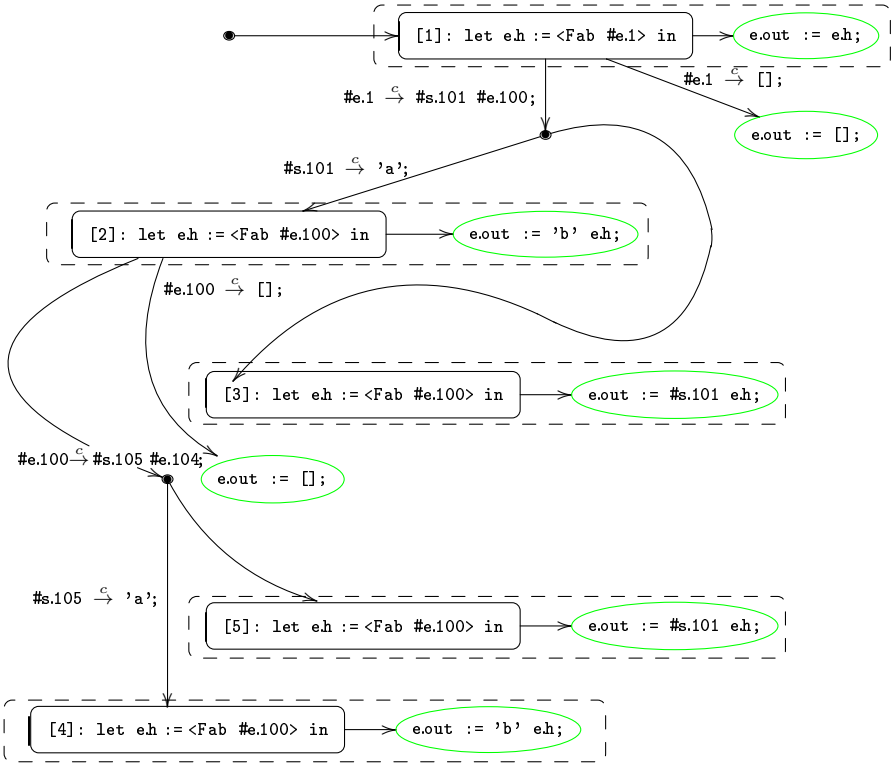


Рис. 2. Дерево возможных вычислений задачи на суперкомпиляцию после второго вызова прогонки.

§ 3. О развитии дерева возможных вычислений

В том случае когда задача на специализацию $\mathcal{Z}$ программы p вырождается в задачу на исполнение (в постановке задачи нет параметров – она полностью определяется конкретными данными рассматриваемого языка программирования $\mathcal{L}$), все шаги вычисления (интерпретации) входной точки (задачи $\mathcal{Z}$) программы p полностью определены. Результат n-ого шага интерпретации $\texttt{STEP}_n$ (см. предыдущую заметку 2) однозначно определяет выбор веток «условных операторов» интерпретации шага $\texttt{STEP}_{n+1}$. Если вычисление заканчивается за конечное число шагов N, тогда результат вычисления задачи $\mathcal{Z}$ есть результат шага $\texttt{STEP}_N$: значение конфигурации, описывающей постановку задачи $\mathcal{Z}$, или аварийная остановка. В этом случае мы имеем конечную последовательность шагов вычисления

$$\texttt{STEP}_1, \ \ldots, \ \texttt{STEP}_N$$

Если p не завершается на входных данных, указанных в постановке задачи $\mathcal{Z}$, тогда процесс вычисления представляется бесконечной последовательностью шагов:

$$\texttt{STEP}_1, \ \ldots, \ \texttt{STEP}_n, \ \ldots$$

Рассмотрим мета-аналог `tree_dev`[2] этого процесса вычислений. Как и при обычной интерпретации, мы будем допускать, что `tree_dev` может строить потенциально бесконечную структуру данных, не завершая своей работы за конечное время.

В случае общего положения, когда задача на специализацию $\mathcal{Z}$ параметризована, выбор веток «условных операторов» процесса вычисления может зависеть от неизвестных (но фиксированных) значений параметров. Ниже мы будем рассматривать только корневые ориентированные деревья (далее кор-дерево), вершины и листы которых помечены параметризованными конфигурациями[3].

Пусть $V(\mathtt{T})$, $L(\mathtt{T})$ обозначают соответственно множества вершин и листьев кор-дерева `T`.

Пусть даны два кор-дерева `K` и `M`. Скажем, что кор-дерево `K` накрывает кор-дерево `M`, если:

1. корень кор-дерева `K` совпадает с корнем кор-дерева `M`;
2. $V(\mathtt{M}) \subseteq V(\mathtt{K})$;
3. $L(\mathtt{M}) \subseteq L(\mathtt{K})$;
4. и любой путь $v_1, \ldots, v_k$ в кор-дереве `M` (здесь для всех i: $v_i \in V(\mathtt{M})$) является подпоследовательностью некоторого пути в кор-дереве `K`.

«В первом приближении» алгоритм `tree_dev` можно понимать нижеследующим образом. `tree_dev` строит кор-дерево `T`, возможно бесконечное, которое *накрывает* кор-дерево $\mathtt{S}_{\mathcal{Z}}$ всех *семантически возможных вычислений* программы `p`, начинающихся из конфигурации постановки задачи $\mathcal{Z}$, и которое *накрывается* кор-деревом Π_Z всех *синтаксически возможных процессов вычислений* программы `p`, исходящих из той же параметризованной входной точки.

Содержательно, кор-дерево $\mathtt{S}_{\mathcal{Z}}$ есть *абстрактная операционная семантика* программы `p` в контексте задачи $\mathcal{Z}$, а кор-дерево Π_Z является полной формальной синтаксической разверткой программы[4] `p`.

При этом, *абстрактная операционная семантика (как и формальная синтаксическая развертка) может зависеть от стратегий развёртки и может не совпадать с операционной семантикой конкретной модели вычислений в рассматриваемом языке программирования* $\mathcal{L}$. Мы требуем только одно: она должна сохранять значение программы `p` на области определения частичной функции, описанной программой `p`.

Следующее пояснение понятия полной синтаксической развертки показывает некоторый набор возможных стратегий абстрактной операционной семантики и необходимость более сложных структур представления результатов этой развертки и алгоритма `tree_dev`, чем простого кор-дерева, данного в *«первом приближении»*.

---

[2] От tree developer.

[3] Параметризованными состояниями стека вызовов функций специализируемых программ – см. выше.

[4] Каждая программа представляет собой текстовую кодировку ориентированного графа с выделенной вершиной – точкой входа; этот граф можно формально развернуть в ориентированное дерево с корнем в точке входа.

Рассмотрим два вложенных цикла, представленных в некотором псевдокоде. Пусть q и g – некоторые предикаты, зависящие от состояния машины в точках их вычисления.

```
while (q)  {
    A;            /* последовательность операторов */
    while (g) {
    B;            /* последовательность операторов */
    }
    C;            /* последовательность операторов */
}
D;                /* последовательность операторов */
```

Рис. 3. Фрагмент псевдокода программы.

В этом случае число итераций во время вычисления внешнего и внутреннего циклов зависит от входных данных программы и, следовательно, в общей ситуации не может быть известно во время преобразований (специализации) программы.

Если процесс синтаксической развертки данного фрагмента псевдокода следует естественному порядку работы этого псевдокода, тогда мы должны разворачивать внутренний цикл.

Результат двух итераций такой развертки схематично показан ниже.

```
while (q) {
    A;
    if (g) then {
        B;
        if (g) then {
           B;
           while (g) {
             B;
            }
        } else C;
    }
    else C;
}
D;
```

Рис. 4. Результат двух итераций развертки первого типа.

И если предположить, что у нас есть магический инструмент бесконечной развертки, тогда внешний цикл будет разворачиваться только после построения внутреннего бесконечного дерева T. В процессе развертки внешнего цикла копии дерева T будут вставляться на ветки этой развертки.

Можно рассмотреть и другой порядок формальной развертки этих двух циклов: сначала разворачивается внешний цикл, а лишь затем копии внутреннего цикла. В этом случае схема результата двух итераций развертки показана на рис. 5.

```
if (q) then {
    A;
    while (g) {
       B;
    }
    C;
    if (q) then {
        A;
        while (g) {
           B;
        }
       C;
       while (q) {
         A;
         while (g) {
            B;
         }
         C;
       }
    } else D;
} else D;
```

Рис. 5. Результат двух итераций развертки второго типа.

Стратегии «ленивых» (call-by-name) вычислений в процессе суперкомпиляции соответствует некоторая комбинация приведенных выше разверток. Подробности будут рассмотрены нами в одной из последующих заметок.

Здесь мы только заметим, что осталось еще много неопределенности, которая должна быть устранена при разработке алгоритма `tree_dev` построения кор-дерева. Например, в каком порядке разворачивать два первых цикла `while (g) { ...}` (см. рис. 5). Мы временно абстрагируемся от этих неопределенностей и стратегий.

Прогонка является основным инструментом развертки средствами суперкомпилятора – это один шаг развертки.

Корень будущего дерева развертки помечается задачей на специализацию $\mathcal{Z}$ – начальной параметризованной конфигурацией.

Развертка начинается с корневой конфигурации: строится гроздь прогонки Г, далее прогоняется один из листьев (обозначим его `a`) грозди Г[5]. Лист `a` заменяется результатом его прогонки – гроздью $\Gamma_{\mathtt{a}}$, после чего заменяется следующий лист и т. д.

Два шага этого процесса для реальных, не упрощенных конфигураций показаны на рисунках 1 и 2 (см. предыдущий п. 2).

Таким образом, структура алгоритма развития дерева возможных вычислений программы `p` в контексте задачи на специализацию $\mathcal{Z}$ средствами суперкомпиляции выглядит следующим образом:

[5]Точнее, конфигурация, помечающая этот лист `a`.

```
tree_dev(p,Z) {
  инициализация;
  while (построение кор-дерева развертки
         в контексте задачи Z не завершено)
              { drv; }
  return (результирующее кор-дерево);
}
```

Где мы опустили аргументы прогонки `drv`.

Алгоритм `tree_dev` является метарасширением интерпретатора[6] языка $\mathcal{L}$, на котором написана специализируемая программа `p` (см. заметку 2).

```
Int(p,d) {
  инициализация;
  while (вычисления не завершены)
              { STEP; }
  return (результирующее состояние);
}
```

Теперь мы готовы внести некоторые уточнения в данный выше набросок алгоритма `tree_dev`.

Пусть дана программа `p` и задача на специализацию этой программы $\mathcal{Z}$.

Для данной параметризованной конфигурации алгоритм прогонки `drv` строит гроздь Г, которая может содержать более одного листа с неполностью вычисленной конфигурацией[7], помечающей этот лист. Ниже мы будем называть такие конфигурации/листы *активными*, а полностью вычисленные конфигурации/листы – *пассивными*.

Ребра в грозди Г помечены предикатами, сужающими множества значений параметров и выбирающими конкретное вычисление. Ребра в Г также упорядочены согласно операционной семантике шага `STEP`, который проверяет (в этом порядке) логические ветвления (`case`, `if`).

На вход алгоритму `tree_dev(p,`$\mathtt{r}_{\mathcal{Z}}$`)` подается лист $\mathtt{r}_{\mathcal{Z}}$, помеченный конфигурацией постановки задачи $\mathcal{Z}$.

Пусть дано дерево $\mathtt{T}_k$ с корнем $\mathtt{r}_{\mathcal{Z}}$. Алгоритм `tree_dev(p,`$\mathtt{T}_k$`)` выбирает один из активных листов n дерева $\mathtt{T}_k$ и подаёт конфигурацию `C`, помечающую этот лист, на вход алгоритма `drv(p,C)`.

Далее, если лист n не является корнем дерева $\mathtt{T}_k$, тогда существует ребро (m, n). В этом случае $\mathtt{T}_k$ преобразуется в дерево $\mathtt{T}_{k+1}$ следующим образом: если результатом `drv(p,`n`)` является

- пустая гроздь, тогда ребро (m, n) дерева и лист n удаляются (ни одно из возможных вычислений, которое может начинаться с n, не заканчивается нормальной (*неаварийной*) остановкой с построением конкретного результата программы `p` на входных данных, указанных в постановке задачи на специализацию $\mathcal{Z}$); пусть `K` – дерево, полученное в результате удаления ребра (m, n), – далее работает алгоритм `if_transitive_node(K,`m`)`;

[6] Алгоритма интерпретации.

[7] Т. е. конфигурацией содержащей вызовы функций/подпрограмм программы `p`.

- непустая гроздь Γ, тогда лист n заменяется на гроздь Γ (т. е. корень $\mathtt{r_C}$ грозди Γ вставляется вместо листа n); пусть дерево K обозначает результат этой подстановки, – далее работает алгоритм `if_transitive_node(K,`$\mathtt{r_C}$`)`.

Если лист n есть корень дерева $\mathtt{T}_k$ (т. е. оно состоит из одного листа), тогда $\mathtt{T}_k$ преобразуется в дерево $\mathtt{T}_{k+1}$ следующим образом:

если результатом `drv(p,`n`)` является

- пустая гроздь, тогда дерево $\mathtt{T}_{k+1}$ является пустым и алгоритм `tree_dev(p,`$\mathtt{r_Z}$`)` завершает работу;
- непустая гроздь Γ, тогда строится ребро $(n, \mathtt{r_C})$, которое помечается всюду истинным предикатом (т. е. ничем); пусть дерево K обозначает результат этого преобразования, – далее работает алгоритм `if_transitive_node(K,`$\mathtt{r_C}$`)`.

Пример двух последовательных деревьев развертки показан на рисунках 1 и 2 из предыдущего п. 2: на рис. 1 дерево $\mathtt{T}_k$, на рис. 2 – дерево $\mathtt{T}_{k+1}$.

На рис. 6 мы видим фрагмент дерева развёртки нижеследующей программы:

```
F {
  'a' e.y = 'b' <G e.y>;
  s.x e.y = s.x <H 'b' e.y>;
   = ;
}
H {
  'a' e.y = 'b' <H e.y>;
  'c' e.y = <H e.y>;
   = ;
}
G {
  ...= ...;
  ...= ...;
}
```

Факт того, что результатом прогонки конфигурации является пустая гроздь, отмечен темной расцветкой листа, который помечает эта конфигурация.

На рис. 7 показан фрагмент того же дерева после удаления ребра, входящего в потемневший («отсохший») лист.

Вершину (узел) в дереве T назовём транзитной, если из неё выходит ровно одно ребро. Если вершина m дерева T является транзитной, тогда алгоритм `if_transitive_node(T,`m`)`: удаляет вершину m; преобразует («склеивает») входящее в и выходящее из m рёбра в одно ребро. Построенное ребро β исходит из вершины, из которой исходило первое (входящее в m), и которое входит в вершину, в которое входило второе ребро. Ребро β помечается композицией предикатов, которые помечали удаленные рёбра.

Рис. 8 демонстрирует результат удаления транзитной вершины, показанной на рис. 7. Входящее и исходящее в эту вершину ребра склеены в одно ребро,

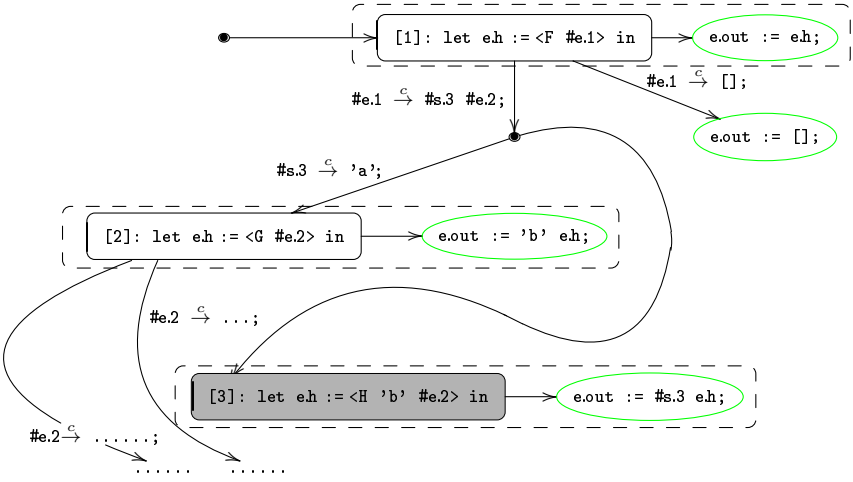


Рис. 6. Фрагмент дерева развертки: в результате прогонки конфигурации, помечающей темный лист, получилась пустая гроздь.

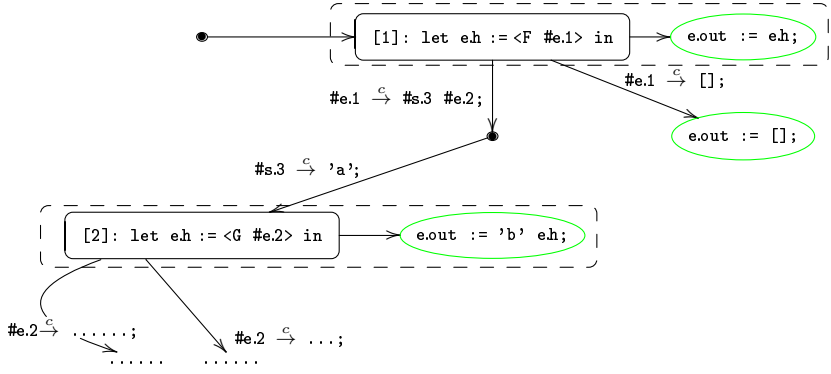


Рис. 7. Фрагмент дерева развертки: в результате удаления ветви (см. рис. 6) появилась вершина с одним исходящим ребром.

которое помечено предикатом, являющимся композицией[8] предикатов, помечавших склеенные ребра.

С абстрактной точки зрения, переход во время интерпретации программы p от одной конфигурации C_1 (состояния) к другой конфигурации C_2 требует действий построения C_2. Следовательно, корректное удаление параметризованной конфигурации, описание которой включает в себя конкретную конфигурацию C_2, из дерева развёртки программы p ведёт к ненужности выполнения этих действий в момент интерпретации. Абстрактное дерево развёртки является прототипом графа, представляющего программу – результат суперкомпиляции программы p. (Далее в наших заметках мы будем результат специализации называть остаточной программой.) Т. е. произошла оптимизация программы p:

[8] В данном случае эта композиция может быть представлена как `#e.1 c→ 'a' #e.2;`.

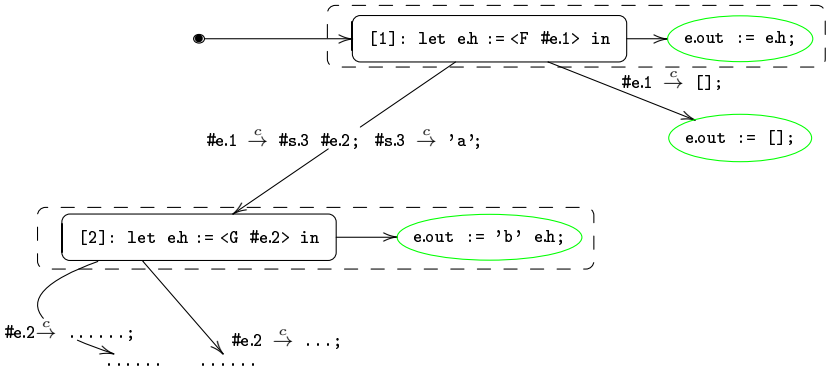


Рис. 8. Фрагмент дерева развертки: удалена вершина с одним исходящим ребром (см. рис. 7), входящее и исходящее в эту вершину ребра склеены в одно ребро.

остаточная программа будет работать быстрее, – если на её вход подать данные, во время интерпретации обеспечивающие «проход» машины через ребро β.

Под «проходом» через ребро мы имеем в виду: попадание машины в одно из конкретных состояний, которые описываются параметризованной конфигурацией, из которой исходит ребро β, и положительный результат проверки условия, описанного предикатом на ребре β.

Эта оптимизация произошла в контексте постановки задачи на специализацию $\mathcal{Z}$, которую решал суперкомпилятор.

С практической точки зрения, ситуация выглядит сложнее. Мы надеемся подробнее обсудить этот вопрос в одной из последующих заметок.

Данную заметку мы завершим указанием на то, что алгоритм `if_transitive_node(T,`m`)` запускается только тогда, когда выбранная стратегия суперкомпиляции разрешает удаление транзитных вершин.

§ 4. О вложении конфигураций

В общем положении, дерево всех возможных вычислений программы, преобразуемой суперкомпилятором в контексте данной задачи на специализацию, является бесконечным.

Бесконечный процесс построения этого дерева не всегда оказывается бессмысленным, – так же как бывают осмысленными программы, не завершающие своей работы до физического выключения вычислительной машины[9].

Однако только в том случае когда процесс суперкомпиляции завершается и остаточная программа обладает какими-то содержательными свойствами оптимальности, которыми не обладает исходная, преобразуемая программа, мы можем говорить о результате суперкомпиляции.

[9]Например: операционная система, коммуникационный протокол и т. д.

Требования завершаемости работы специализатора на всех входных программах и оптимальности[10] *полученных в результате таких завершений остаточных программ в общем случае являются противоречивыми.*

Это утверждение следует из алгоритмической неразрешимости задачи распознавания любого нетривиального семантического свойства программ, написанных на алгоритмически полных языках программирования[11].

Бесконечное (точнее – не являющееся равномерно ограниченным по значениям входных параметров) дерево возможных вычислений T можно попытаться представить конечным графом.

Процесс свертки дерева T в конечный граф является ключевым алгоритмом суперкомпилятора. По сути, только этим алгоритмом и отличаются разные суперкомпиляторы – от игрушечных-модельных до экспериментального.[12] В любом более-менее содержательном «суперкомпиляторе» этот алгоритм является очень сложным. Наиболее трудоемкий его под-алгоритм называется *обобщением*; к нему мы надеемся обратиться в наших последующих заметках – сначала описывая схематично и далее шаг за шагом вводя уточнения.

Будем смотреть на процесс свертки дерева T как на попытку доказательства следующего утверждения:

«Пусть дана программа $\mathtt{p}$ *и задача на специализацию этой программы* $\mathcal{Z}$*. Тогда дерево возможных вычислений* T*, соответствующее паре* $(\mathtt{p}, \mathcal{Z})$*, можно за конечное время свернуть (представить в виде) в некоторый конечный граф.»*

Конечно, здесь нужно уточнение. Что мы имеем в виду в выражении *«представить в виде»*? Необходимые пояснения будут даны ниже.

На содержательном же уровне сформулированное выше утверждение, очевидно, имеет место, ибо дерево T получено в результате развёртки графа программы p в контексте задачи $\mathcal{Z}$ (см. предыдущий п. 3). Именно, например, в этот граф и можно свернуть дерево T. Могут, конечно, существовать (и всегда существуют!) и другие графы свёртки дерева T.

Нас интересуют графы, обладающие (в каком-то смысле) наилучшими свойствами оптимальности. Тонкий вопрос об оптимальности результата свёртки мы пока оставим за скобками рассмотрения.

Мы встанем на точку зрения алгоритма свёртки, которому *a priori* вообще не известен ни один граф свёртки дерева T. Однако мы будем предполагать, что поставленная задача построения графа свёртки может быть алгоритмически решена за конечное время посредством механизмов, о которых мы говорим ниже.

Рассмотрим первый модельный пример. Пусть множество данных в модельном языке программирования совпадает с множеством натуральных чисел.

Функцию проекции на второй аргумент в этом языке можно неоптимально определить, например, так:

[10] В этой заметке мы никак не уточняем понятие *«оптимальности»* и обращаемся к интуиции читателя.

[11] См. классическую теорему Успенского-Райса в монографии Н. К. Верещагина, А. Шеня «Вычислимые функции» [1].

[12] Мы здесь говорим не о способе кодирования/программе, а именно об алгоритме.

```
F(0,m) = m;
F(n+1,m) = F(n,m);
```

Рис. 9. Программа `q` проекции на второй аргумент, написанная на модельном языке программирования.

Рассмотрим следующую задачу на специализацию: `F(#n,#m)`. Построим дерево (всех) возможных вычислений пары (`q`, `F(#n,#m)`):

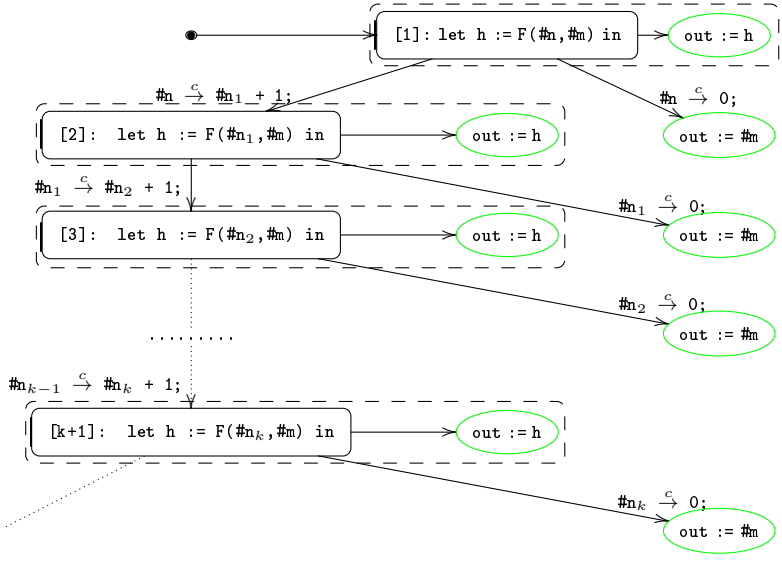


Рис. 10. Дерево всех возможных вычислений для (`q`, `F(#n,#m)`).

Предикат $\texttt{\#n} \xrightarrow{c} \texttt{\#j + 1}$ читается как «значение параметра `#n` имеет вид `#j + 1`», где справа и слева стоят *разные* параметры.

Наша цель – провести конструктивное доказательство возможности свертки этого бесконечного дерева в конечный граф. Будем регулярно обходить и анализировать это дерево от корня в глубину.

Ветвь, ведущая из корня дерева в эллипсоидную вершину, соответствует значению параметра $\texttt{\#n} = 0$. Эта ветвь конечна – эллипсоидная вершина является листом. Следовательно, подграф, висящий на этой ветви, конечен и состоит из одного узла.

Второе ребро, исходящее из корня, сводит решение задачи специализации `F(#n,#m)` программы `q` к задаче специализации $\texttt{F(\#n}_1\texttt{,\#m)}$ той же самой программы `q`.

Поскольку описания этих двух задач отличаются лишь именами параметров, то дерево развертки (дерево всех возможных вычислений) первой задачи совпадает с точности до имён параметров с деревом развёртки второй задачи.

Следовательно, решение второй задачи можно свести к решению первой задачи, – посредством замены значения параметра `#n` на значение параметра $\texttt{\#n}_1$:

$$\texttt{\#n := \#n}_1\texttt{;}$$

Или то же самое на языке графов: можно с точностью до переименования параметров отождествить вершину `[2]` с вершиной `[1]` и таким образом свернуть ветвь развёртки задачи `F(#n,#m)`, соответствующую предикату

$$\texttt{\#n} \xrightarrow{c} \texttt{\#n}_1 \texttt{ + 1;}$$

в конечный граф[13].

Результат описанной свёртки представлен на рис. 11, в котором оставлена информация о замене/подстановке параметров, позволившей свернуть основную ветвь дерева развёртки.

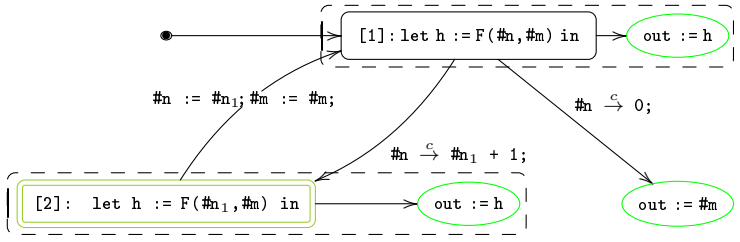


Рис. 11. Граф – результат свёртки дерева всех возможных вычислений задачи $(\texttt{q}, \texttt{F(\#n,\#m)})$.

Отметим, что подстановка, сводящая вычисления, которые могут начаться из одной вершины дерева развёртки, к вычислениям, которые могут начаться из другой вершины, может иметь более сложную структуру, чем в разобранном выше примере.

Например, для программы вычисления суммы первого и второго аргумента:

```
A(0,m) = m;
A(n+1,m) = A(n,m+1);
```

Рис. 12. Программа вычисления суммы входных аргументов.

Результат свёртки дерева возможных вычислений задачи на специализацию `A(#n,#m)` будет отличаться от графа на рис. 11 только подстановкой сведения: она в этом случае будет иметь следующий вид $\texttt{\#n := \#n}_1\texttt{; \#m := \#m+1;}$.

Подобным образом – с помощью замены параметров – сводить одну задачу специализации $\mathcal{Z}_2$ некоторой программы `p` к другой задаче специализации $\mathcal{Z}_1$ этой же программы можно и тогда, когда множество возможных вычислений, начинающихся в задаче $\mathcal{Z}_2$, необязательно совпадает с множеством возможных вычислений, начинающихся в задаче $\mathcal{Z}_1$, а является его подмножеством. Важно только, чтобы задача $\mathcal{Z}_1$ в процессе свёртки дерева возможных вычислений `T` уже рассматривалась раньше задачи $\mathcal{Z}_2$.

[13] Если бы не было оговорки «с точностью до переименования», тогда подобное отождествление в математике называлось бы факторизацией.

Здесь происходит шаг математической индукции по построению дерева T.

При этом в T может не существовать пути из вершины, в которой описана $\mathcal{Z}_1$, в вершину, в которой описана $\mathcal{Z}_2$. Такой путь существует в рассмотренном выше модельном примере (см. рис. 10).

Напомним, что задача на специализацию описывается посредством параметризованной конфигурации (см. п. 2).

Если одну параметризованную конфигурацию $\mathtt{C}_2$ в дереве T можно свести посредством подстановки параметров σ к другой параметризованной конфигурации $\mathtt{C}_1$ в T, тогда говорят, что конфигурация $\mathtt{C}_2$ *вкладывается* в конфигурацию $\mathtt{C}_1$ посредством σ.

При попытке вложения конфигурации $\mathtt{C}_v$, помечающей конкретную выбранную вершину v в дереве T, может существовать несколько вершин $u_1, \ldots, u_n$, ранее рассмотренных алгоритмом сверки, в параметризованные конфигурации из которых $\mathtt{C}_{u_1}$, ..., $\mathtt{C}_{u_n}$ вкладывается конфигурация $\mathtt{C}_v$, помечающая вершину v. Следовательно, возникает необходимость в стратегии выбора одной конфигурации из множества $\mathtt{C}_{u_1}$, ..., $\mathtt{C}_{u_n}$, к которой и будет сведена конфигурация $\mathtt{C}_v$. Мы откладываем вопрос обсуждения таких стратегий на последующие заметки.

Внимательный читатель уже отметил, что обсуждаемое в этой заметке понятие вложения одной конфигурации в другую ему также (как и понятие прогонки) давно известно из школьного курса математики.

Это замена переменных при решении алгебраических уравнений. Здесь уравнение играет роль конфигурации, а переменные – роль параметров. Смысл замены переменных состоит в том, чтобы свести процесс решения одного уравнения к решению другого уравнения, которое ученик уже умеет решать. Например, методом замены переменных биквадратные уравнения от одной переменной сводятся к квадратным уравнениям от другой переменной.

При этом школьник должен запомнить саму замену переменных, чтобы, решив получившееся квадратное уравнение, можно было найти корни исходного биквадратного уравнения.

По аналогичной причине мы оставляем в графе свёртки информацию о подстановках сведения одной конфигурации к другой (см. рис. 11).

Пусть дано дерево развертки T некоторой задачи на специализацию $(\mathtt{q}, \mathcal{Z})$. Всегда ли существуют две разные вершины v, u дерева T такие, что либо конфигурация $\mathtt{C}_u$ вкладывается в конфигурацию $\mathtt{C}_v$, либо конфигурация $\mathtt{C}_v$ вкладывается в конфигурацию $\mathtt{C}_u$?

От ответа на этот вопрос зависит, можно ли обойтись описанным выше приемом вложения одной вершины в другую, или нужны дополнительные механизмы для свёртки дерева T в конечный граф.

Ответ на поставленный вопрос отрицательный.

Построим соответствующий пример. Рассмотрим программу q, данную на рис. 12. Дерево возможных вычислений задачи на специализацию `A(#n,1)` этой программы показано на рис. 13.

Вторые аргументы всех конфигураций из вершин ветвления этого дерева являются константами, и все эти константы различны. Следовательно, ни одна из конфигураций не может быть вложена в другую.

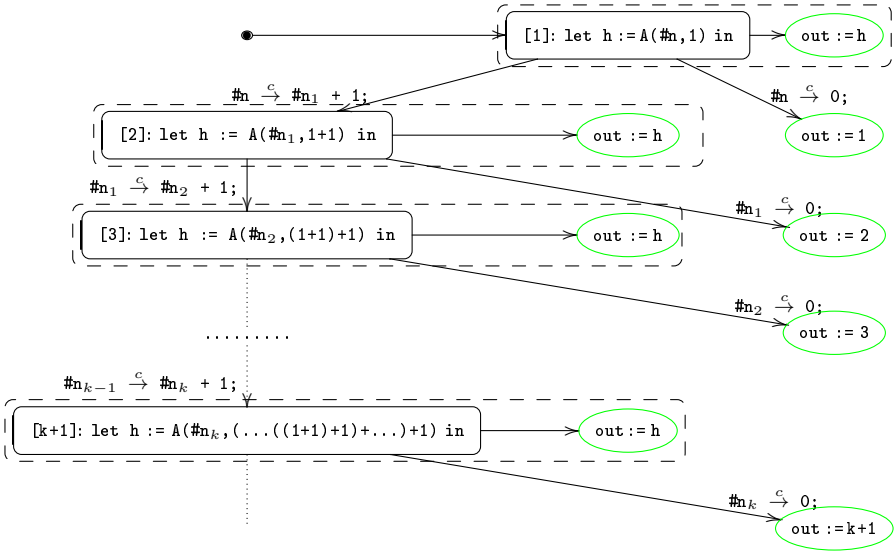


Рис. 13. Дерево всех возможных вычислений для (q, A(#n,1)).

Таким образом, чтобы свернуть произвольное дерево развёртки в конечный граф, необходимы дополнительные инструменты, которые мы надеемся обсудить в наших последующих заметках.

Данное выше неформальное описание процесса свёртки принимает на вход всё дерево возможных вычислений T программы p в контексте выбранной задачи на специализацию $\mathcal{Z}$. Как правило, T является бесконечным.

Нашим объектным языком программирования является Рефал, множество данных которого суть множество конечных последовательностей произвольных конечных деревьев. Следовательно, все алгоритмы суперкомпиляции могут работать только с элементами последовательности конечных деревьев

$$\mathtt{T}_1,\ \mathtt{T}_2,\ \ldots,\ \mathtt{T}_n,\ \ldots$$

которые аппроксимируют/приближают дерево возможных вычислений T.

Зафиксируем некоторое число k и рассмотрим дерево $\mathtt{T}_k$. Предположим, что мы свернули одну из потенциально бесконечных ветвей дерева $\mathtt{T}_k$. Дерево $\mathtt{T}_k$ превратилось в ориентированный граф $\mathtt{G}_k$ – в этом графе имеется цикл (см. рис. 11).

Начиная с этого момента входные структуры данных для большинства алгоритмов суперкомпиляции логически становятся, в общем случае, ориентированным графами, а не деревьями[14]. И, строго говоря, при необходимости последующего развития $\mathtt{G}_k$ – развёртки по другим веткам дерева $\mathtt{T}_k$, на которых алгоритм свёртки еще не работал, мы будем получать вместо рассмотренной выше последовательности деревьев последовательность ориентированных конечных графов

$$\mathtt{T}_1,\ \mathtt{T}_2,\ \ldots,\ \mathtt{G}_k,\ \ldots,\ \mathtt{G}_n,\ \ldots$$

[14]Именно логически, поскольку эти графы кодируются в Рефале деревьями.

которые также, в некотором смысле, приближают дерево возможных вычислений $\mathtt{T}$[15].

Иногда мы будем называть графы $\mathtt{G}_k$, ..., $\mathtt{G}_n$, ... частично свёрнутыми деревьями, таким образом подчёркивая, что в этих графах есть выделенная вершина – корень дерева $\mathtt{T}_k$ (мы будем называть эту вершину корнем графа) и (потенциально) бесконечные пути – несвёрнутые пути в дереве $\mathtt{T}_k$.

Пусть v некоторая вершина частично свёрнутого дерева $\mathtt{T}_k$. Обозначим через $\mathtt{reduce}(\mathtt{T}_k, v)$ некоторый алгоритм попытки свёртки пути из корня частично свёрнутого дерева $\mathtt{T}_k$, содержащего вершину v.

Алгоритм $\mathtt{reduce}(\mathtt{T}_k, v)$ пытается вложить конфигурацию $\mathtt{C}_v$ в одну из конфигураций, из некоторого подмножества S вершин w_j, уже рассмотренных предыдущими вызовами этого же алгоритма $\mathtt{reduce}(\mathtt{T}_i, w_j)$, где $i < k$.

Попытка вложения может оказаться:

- неудачной – конфигурация $\mathtt{C}_v$ не вкладывается ни в одну из конфигураций $\mathtt{C}_{w_j}$, где w_j пробегает множество S;
- либо удачной – в множестве S нашлась вершина w_m такая, что $\mathtt{C}_v$ вкладывается в $\mathtt{C}_{w_m}$; и в этом случае путь, содержащий вершину v, сворачивается в цикл.

Наша ближайшая цель – выписать первое приближение структуры алгоритма суперкомпиляции.

Ранее[16] мы подчёркивали, что алгоритмы прогонки и развития дерева возможных вычислений являются метарасширениями алгоритмов интерпретатора объектного языка программирования[17].

Для алгоритма вложения нет соответствующего подалгоритма интерпретатора, для которого вложение можно было бы рассматривать как его метарасширение.

Некоторую аналогию с вложением можно было бы указать в минимизирующем интерпретаторе[18], который следит за множеством всех вызов функций, которые уже полностью вычислены к моменту очередного вызова $\mathtt{F}(\mathtt{x}_0, \ldots, \mathtt{y}_0)$, и сохраняет значения этих предыдущих вызовов в таблице минимизации. Следовательно, такому интерпретатору нет необходимости заново вычислять значение $\mathtt{F}(\mathtt{x}_0, \ldots, \mathtt{y}_0)$ – оно уже известно – нужно просто его взять из таблицы. В данном случае имеет место вложение (посредством тождественной подстановки) вызова $\mathtt{F}(\mathtt{x}_0, \ldots, \mathtt{y}_0)$ в один из вызовов, который уже встречался в истории вычислений задачи, данной интерпретатору.

При построении первого приближения структуры алгоритма суперкомпиляции SCP мы будем отталкиваться от структуры алгоритма развития дерева возможных вычислений программы $\mathtt{p}$ в контексте задачи на специализацию $\mathcal{Z}$.[19]

Напомним его, явно указав аргумент прогонки:

---

[15] Если заново развернуть их свёрнутые пути.
[16] См. заметки 2, 3.
[17] В нашем случае – Рефала.
[18] Интерпретаторы Рефала таковыми не являются
[19] См. заметку 3.

```
tree_dev(p,𝒵) {
   инициализация;
   while (построение кор-дерева развертки
          в контексте задачи 𝒵 не завершено)
              { v := один из листов кор-дерева;
                drv(C_v);
              }
   return (результирующее кор-дерево);
}
```

Здесь, как и выше по тексту, $\mathtt{C}_v$ обозначает параметризованную конфигурацию, помечающую лист v.

Теперь мы явно укажем последовательность приближений к результирующему кор-дереву. Получим:

```
tree_dev(p,𝒵) {
   T_1 := вершину, помеченную описанием задачи 𝒵;
   i := 1;
   while (построение кор-дерева развертки
          в контексте задачи 𝒵 не завершено)
              { v := один из листов кор-дерева T_i;
                cluster := drv(C_v);
                T_(i+1) := заменить в T_i лист v гроздью
                           прогонки cluster;
                i := i+1;
              }
   return (результирующее кор-дерево);
}
```

Где корень грозди прогонки вставляется вместо листа v.

Первое приближение структуры алгоритма суперкомпиляции:

```
SCP(p,𝒵) {
   T_1 := вершину, помеченную описанием задачи 𝒵;
   i := 1;
   while (в T_i несвёрнутый (потенциально)
          бесконечный путь)
              { v := один из листов кор-дерева T_i;
                cluster := drv(C_v);
                T_(i+1) := заменить в T_i лист v гроздью
                           прогонки cluster;
                i := i+1;
                if (в cluster существует
                    нетранзитная вершина)
                  {   v := первую от корня
                           нетранзитную вершину
                           грозди прогонки cluster;
                       reduce(T_i,v);  }
              }
   return (результирующий граф);  }
```

Сделаем несколько замечаний:

- Алгоритм вложения `reduce(`$\mathtt{T}_i$`,`v`)` пытается вложить не конфигурацию $\mathtt{C}_v$ из листа v дерева, а конфигурацию из первой вершины ветвления грозди, полученной в результате прогонки конфигурации $\mathtt{C}_v$, если такая вершина существует.
  Прогонка конфигурации $\mathtt{C}_v$ позволяет определить некоторые свойства $\mathtt{C}_v$ – с точки зрения оптимизации частично свёрнутого дерева $\mathtt{T}_{i+1}$.
  Например, если конфигурация $\mathtt{C}_v$ является транзитной и может быть вложена в одну из конфигураций из множества S (см. выше), тогда выполненная перед прогонкой свёртка ветви, на которой находится лист v, оставит необходимость исполнения программой – результатом суперкомпиляции действий преобразования конфигурации $\mathtt{C}_v$ (после подстановки конкретных значений параметров этой конфигурации).
  Эти действия можно было бы выполнить во время суперкомпиляции – равномерно по значениям параметров.
  Вызов алгоритма прогонки нужен здесь, чтобы определить, является конфигурация $\mathtt{C}_v$ транзитной или не является.
- С другой стороны, игнорирование транзитных вершин в процессе свёртки может привести к незавершаемости работы суперкомпилятора.
  И решение об оформлении циклов из вершин, помеченных транзитными конфигурациям, должно приниматься одной из стратегий, что не отражено в данной нами выше схеме алгоритма суперкомпиляции.
- В структуре алгоритма SCP также явно не указаны стратегии выбора:
  1. листа v частично свёрнутого дерева, конфигурация $\mathtt{C}_v$ из которого будет прогоняться;
  2. конкретной конфигурации из множества S (см. выше), в том случае если в S существует несколько конфигураций, в которые может быть вложена $\mathtt{C}_v$;
  3. выбор самого множества конфигураций S.

Проследим работу описанной выше структуры алгоритма суперкомпиляции на примере специализации аналога программы, данной на рис. 9, в контексте соответствующей задачи на специализацию в терминах языка Рефал.

Ниже дано определение соответствующей программы.

```
F {
          (e.m)  = e.m;
  e.n '1' (e.m)  = <F (e.n) (e.m)>;
}
```

Рис. 14. Программа `q` проекции на второй аргумент, написанная на языке программирования Рефал.

Задача на специализацию, соответствующая рис. 10, в терминах Рефала определяется так: `<F #n (#m)>`. На выходе прогонки этой стартовой конфигурации имеем гроздь, показанную на рис. 15.

Эта гроздь прогонки совпадает с деревом $\mathtt{T}_2$. По техническим причинам суперкомпилятор SCP4 переименовал/заменил параметры. Безымянное ребро

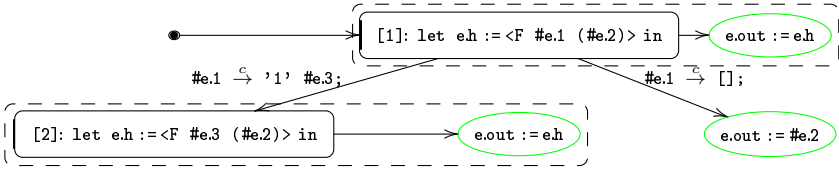


Рис. 15. Гроздь результата прогонки конфигурации `<F #n (#m)>`, описывающей задачу на специализацию программы `q`.

выходит из служебной вершины, в которой записана подстановка `#m := #e.1; #m := #e.2;` – произошедшая замена параметров. Эта подстановка не показана на рисунке и нужна для корректного описания входной точки остаточной программы.

В дереве T_2 нет ни одной активной вершины, расположенной выше вершины `[1]`, конфигурация из которой была подана на вход прогонки.

Следовательно, множество S вершин, в которые можно было бы пытаться вложить конфигурацию `<F #e.1 (#e.2)>`, является пустым множеством.

Согласно структуре алгоритма SCP, выбираем лист дерева T_2. Это дерево имеет единственный активный лист.

Подаём конфигурацию, помечающую этот лист, на вход очередного вызова алгоритма прогонки.

На рис. 16 показано дерево T_3.

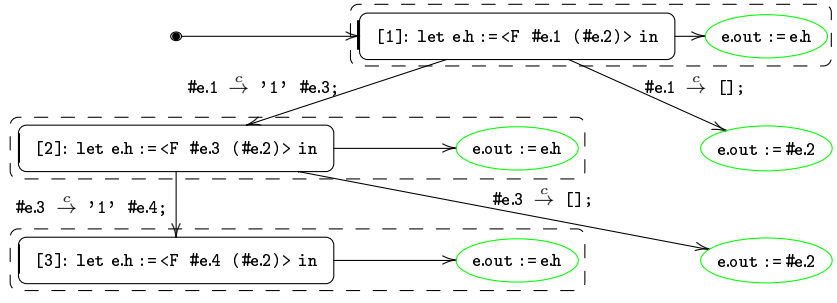


Рис. 16. Дерево T_3.

Вершина `[2]` дерева T_3 подается на вход алгоритма `reduce(`T_3`,[2])`.

В данном случае суперкомпилятор SCP4 выбирает в качестве множества S множество всех вершин, принадлежащих пути из корня T_3 в вершину `[2]`. Множество S содержит единственную вершину `[1]`.

Конфигурация из вершины `[2]` вкладывается в конфигурацию, помечающую корень.

Дерево возможных вычислений полностью свёрнуто в конечный граф (см. рис. 17).

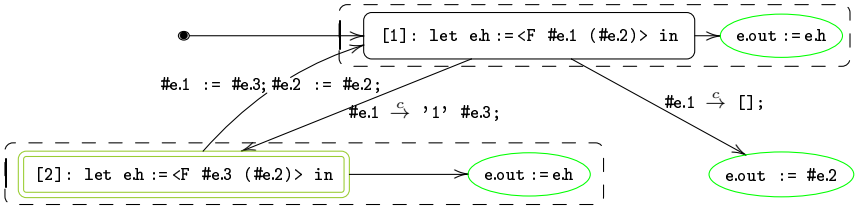


Рис. 17. Конечный граф, представляющий свёртку дерева возможных вычислений для (q, <F #n (#m)>).

§ 5. Заключение

Автор надеется продолжить заметки, начатые в данной статье.

Список литературы

[1] Н. К. Верещагин, А. Шень, *Вычислимые функции*, Издание четвёртое, исправленное, Издательство МЦНМО, Москва, 2012.

[2] К. Гао, А. П. Немытых, *CREFAL: компилятор Рефала-5 в язык сборки*, ([online]: http://www.botik.ru/pub/local/scp/refal5/), 2004.

[3] А. В. Корлюков, *Пособие по суперкомпилятору SCP4*, ([online]: http://www.refal.net/supercom.htm), 1999.

[4] А. П. Немытых, В. Ф. Турчин, *Суперкомпилятор SCP4: исходные тексты, on-line демонстрация*, ([online]: http://www.botik.ru/pub/local/scp/refal5/), 2000.

[5] А. П. Немытых, В. Ф. Турчин, *Суперкомпилятор SCP4: краткие пояснения к демонстрационным примерам*, ([online]: http://www.botik.ru/~scp/doc/dem_help.html), 2000.

[6] А. П. Немытых, В. Ф. Турчин, *Суперкомпилятор SCP4: on-line демонстрация*, ([online]: http://refal.botik.ru/scp_demo/), 2000.

[7] А. П. Немытых, *Суперкомпилятор SCP4: общая структура*, ISBN 978-5-382-00365-8, Издательство УРСС, Москва, 2007.

[8] А. П. Немытых, "О суперкомпиляции (к 80-тилетию со дня рождения В. Ф. Турчина)", *Proceedings of International Conference "Modern Problems of Mathematics, Informatics and Bioinformatics", devoted to the 100th anniversary of professor Alexei A. Lyapunov*, ISBN: 978-5-905569-03-6 (available at: http://conf.nsc.ru/files/conferences/Lyap-100/fulltext/69293/69928/nemytykh_supercompilation_Lyapunov100.pdf), Novosibirsk, Russia, 2011.

[9] В. Ф. Турчин, *Программирование на языке РЕФАЛ*, Препринт №43, ИПМ АН СССР, 1971.

[10] В. Ф. Турчин, *РЕФАЛ-5. Руководство по программированию и справочник*, ([online]: http://www.refal.org/rf5_frm.htm русский перевод *устаревшей* версии книги [17].).

[11] В. Ф. Турчин, Д. В. Турчин, А. П. Конышев, А. П. Немытых, *Рефал-5: исполняемые модули и исходные тексты*, ([online]: http://www.botik.ru/pub/local/scp/refal5/), 2000.

[12] A. P. Lisitsa, A. P. Nemytykh, "Reachability Analisys in Verification via Supercompilation", *International Journal of Foundations of Computer Science*, **19**, № 4, 2008, 953–970.

[13] A. P. Nemytykh, V. A. Pinchuk, V. F. Turchin, "A Self-Applicable Supercompiler", *Proc. of Partial Evaluation*, LNCS, **1110**, Springer-Verlag, 1996, 322-337.

[14] The GHC Team, *Haskell 2010: A Non-strict, Purely Functional Language. Haskell 2010 language report*, ([online]: http://haskell.org/definition/haskell2010.pdf), 2010.

[15] V. F. Turchin, "The concept of a supercompiler", *ACM Transactions on Programming Languages and Systems*, **8** (1986), 292–325.

[16] V. F. Turchin, "The algorithm of generalization in the supercompiler", *Partial Evaluation and Mixed Computation*, 1988, 341–353.

[17] V. F. Turchin, *REFAL-5 programming guide and reference manual*, (переработанное и расширенное издание 1999 года доступно как zipped html-файл: http://refal.botik.ru/book/refal-book-html.zip), New England Publishing Co., Holyoke, 1989.

[18] V. F. Turchin, A. P. Nemytykh, *Metavariables: their Implementation and Use in Program Transformation*, Report N. TR 2095-012, The City College of the City University of New York, 1995.

А. П. Немытых (A. P. Nemytykh)
Переславль-Залесский, ИПС РАН
E-mail: nemytykh@math.botik.ru

М. Ш. Исламов

Развитие языка программирования РЕФАЛ. Диалект Refal-D

В данной статье описывается результат работы в области совершенствования языка программирования Рефал, с целью его адаптации к актуальным задачам, путём расширения семантики концептуально непротиворечивыми, эффективными конструкциями. Итогом работы стало определение нового диалекта языка Рефал – Refal-D, интерпретатор которого находится в свободном доступе [12].

Библ. 17 наим.

Ключевые слова: сентенциальное программирование, Рефал, регулярные выражения, декларативное программирование.

Содержание

Работа выполнена в 2010 году в рамках кандидатской диссертации под научным руководством д.ф.-м.н. Н. Н. Непейвода

Введение

Рефал – сентенциальный язык программирования [6], созданный в СССР профессором В. Ф. Турчиным в 60-х годах прошлого века, обладающий высокой степенью декларативности и достаточно простой и красивой логикой вычислений.

Декларативные языки ускоряют процесс разработки программ, сокращают объём исходного кода, позволяют решать задачи более высокого уровня сложности за счёт того, что программисту не приходится полностью выстраивать алгоритм решения – от него требуется описание предметной области и постановка задачи, а поиск решения возлагается на вычислительную машину.

В последние годы наблюдается заметный рост популярности декларативных инструментов в промышленном программировании (например, активное использование лямбда-выражений в Java, C# и др.), который связан в том числе с тем, что для многих актуальных на сегодня задач современные аппаратные возможности уже обеспечивают приемлемую скорость декларативных вычислений (с учётом разницы между логической и вычислительной моделями). Стоимость вычислений на современных компьютерах часто бывает существенно ниже стоимости процесса разработки соответствующего программного обеспечения, поэтому использование декларативных инструментов там, где они уместны, ещё и экономически целесообразно.

Таким образом, развитие декларативного программирования и, в частности, сентенциального программирования - одно из актуальных направлений развития информационных технологий.

Данная работа посвящена развитию языка программирования Рефал, который может быть эффективно применим при обработке структурированных и неструктурированных текстов, в символьных вычислениях, метавычислениях, в некоторых других областях. Описываемый в статье диалект языка Refal-D является расширением фактического стандарта Рефала с целью его адаптации к решению современных промышленных задач. Оглядываясь на историю развития других языков (да и самого Рефала), можно сделать вывод о том, что расширение какого-либо языка может быть успешным только в том случае, если расширяющие конструкции и нововведения не противоречат его концептуальным основам и действительно являются необходимыми и востребованными в той области, в которой возникла такая задача. Руководствуясь вышесказанным, автор ставил принцип концептуальной непротиворечивости на первое место.

§ 1. Диалекты языка Рефал

В основе языка лежит идея алгоритмов Маркова [1], широко применяемая в инструментах прототипирования. В первоначальном виде Рефал являлся метаязыком, предназначенным для описания других алгоритмических языков [8, 9], но в результате появления достаточно эффективных реализаций для ЭВМ, он стал находить практическое использование в качестве языка символьных преобразований. На сегодняшний день существует несколько диалектов языка [10, 3, 7, 14, 13, 4], которые достаточно сильно отличаются друг от друга уже

на уровне семантики, но содержат одно общее подмножество – Базисный Рефал [10]. Фактическим стандартом среди них признан Рефал-5 [14], разработанный самим Турчиным в 1989 году[1].

Добавленные в Рефал-5 расширения Базисного Рефала были представлены автором языка как инструменты, сокращающие классическую запись программ, однако фактически они явились направляющими дальнейшего развития языка. Вспомним, что Рефал-5 отличается от базисного наличием `with`-конструкций (блоков) и `where`-конструкций (условий). Следующие версии языка (Рефал-6, Рефал+) явились углубленным развитием этих конструкций, в результате чего в диалектах появились откаты блоков и функций, предложения были заменены тропами с заборами и отсечениями[2] [4], было введено некоторое подобие объектов ООП. Последнее, скорее всего, явилось данью моде (примеры значимого применения автору статьи не известны), остальное - это инструменты управления сопоставлением. Таким образом, вместо декларативного описания постановки подзадач, пользователю предлагается программировать процесс сопоставления, что приводит к отклонению от заложенного Турчиным основополагающего принципа: язык должен быть ориентирован в первую очередь на модель человеческого мышления, а не на модель вычислительной машины [8, 9].

Поэтому в работе по дальнейшему развитию языка в сентенциальном направлении с сохранением базовых концепций автор статьи отталкивался от диалекта Рефал-5.

§ 2. Refal-D. Базис

Как и в языке-предшественнике, для представления любых данных в Refal-D используются объектные выражения, а основными операциями являются: сопоставление объектного выражения с образцом и подстановка.

От стандартного Рефала новый диалект унаследовал способ определения функции в виде списка предложений с левой и правой частью. Рефал-условия также вошли в базис нового диалекта, в отличие от рефал-блоков, которые нарушают структуру рефал-предложения и могут быть заменены другими конструкциями языка.

Программы на языке Refal-D выполняет специальная рефал-машина, принципы работы которой подробно описаны в [14].

2.1. Типы данных. Стандартными символами языка являются: текстовые символы, текстовые термы, целые числа, вещественные числа, байты. Вместе со структурными скобками они служат для образования объектных выражений языка. Для пустого объектного выражения в диалекте предусмотрено обозначение: `$empty`. Для работы с текстовыми символами используется ставший уже стандартом формат Юникод (Unicode). Поскольку юникод-символы

[1] В конце 90-х годов В. Ф. Турчин внёс небольшие изменения в синтаксис диалекта Рефал-5, которые описаны в [15]. Соответствующие изменения были внесены в компилятор этого диалекта Рефала (см. [16]). - *Прим. ред.*

[2] Здесь имеются ввиду специальные термины конструкций языка, с помощью которых разработчик определяет процесс сопоставления на более низком уровне.

не совместимы с байтами, последние были выделены в отдельный тип данных языка. Как и в языке-предшественнике, для определения текстовых термов используются двойные кавычки, которые могут опускаться, если текстовый терм начинается с латинской буквы и не содержит символов, отличных от латинских букв и цифр.

2.2. Переменные. Рефал-5 имеет три типа переменных: `s`, `t` и `e`. В описываемом диалекте присутствует ещё один тип переменных - это тип «жадная `e`-переменная» (обозначается - `E`). При инициализации `E`-переменная захватывает самое длинное допустимое подвыражение и, в случае отката, укорачивает область захвата на один терм влево. Это даёт программисту очень гибкий инструмент управления шаблонами и полностью заменяет механизм сопоставления справа налево: он аналогичен замене всех `e`-переменных левой части на переменные типа `E`.

В диалектах языка Рефал переменная может быть открытой (свободной) или закрытой (ссылающейся на открытую)[3]. В языке Refal-D для определения закрытых переменных может быть использовано специальное обозначение – символ `'@'` вместо имени типа. Данное обозначение может быть применено к любым закрытым переменным, но оно является необходимым для случаев, когда соответствующая открытая переменная не имеет имени типа (ниже будут подробнее рассмотрены такие случаи).

Чтобы избежать загромождения текста программ лишней информацией, переменные, которые используются в предложении только один раз, могут быть описаны без имени (но точка в конце обязательна). Такие переменные далее называются безымянными. Любая безымянная переменная является открытой. Из этого следует, что два идентичных объявления безымянной переменной являются объявлениями различных переменных.

```
e. 'REFAL-' s.version e. =  @.version;
```

Пример 1: Рефал-предложение с безымянными переменными и закрытой переменной `version`.

Стоит также отметить, что синтаксис описываемого диалекта Рефала запрещает использование в одном предложении программы одноимённых переменных разного типа.

2.3. Спецификаторы: группы, альтернативы и повторители. Пожалуй наиболее концептуально чистым из существующих диалектов Рефала на момент написания статьи, является Базисный Рефал. В нём был определен подход описания решения задачи через предложения (сентенции) - подход более близкий человеку, чем компьютеру. Но практическое применение языка показало его недостаточность. Решение какой-нибудь простой подзадачи требовало использования вспомогательных функций, что разрушало целостность и стройность описания алгоритма, затрудняло чтение кода программы и

[3]Автор использует стандартные термины языка Рефал - открытая и закрытая переменная, придавая им иную семантику. См. сноску на следующей странице. - *Прим. ред.*

вынуждало программиста совершать действия, не связанные напрямую с решением целевой задачи. Поэтому появление различных расширений Базисного Рефала было закономерным явлением. Рефал-условия и рефал-блоки, бесспорно, сделали Рефал-5 на порядок практичнее, однако при решении большинства стандартных задач, связанных с обработкой текста, программисты предпочитают использовать языки с более гибкими регулярными выражениями, выделяя отсутствие в Рефале аналогичных спецификаторов как недостаток языка.

Для устранения этого недостатка в Refal-D были добавлены три конструкции, расширяющие описание образцов: группы, альтернативы и повторители.

Группой называется описание переменной, тип которой определяется некоторым образцом. В конкретном синтаксисе группы обозначаются фигурными скобками, внутри которых задаётся образец. Этот образец может содержать как закрытые переменные[4] (определённые левее группы), так и открытые, но эти открытые переменные не видны в левой части рефал-предложения за пределами группы - их использование там запрещено.

```
e. { '<Tag>' e. '</Tag>' }.tag e.
```

Пример 2: Образец с группой.

После определения группы, в левой части рефал-предложения могут использоваться закрытые переменные, ссылающиеся на значение переменной этой группы. Определяются они с помощью описанного выше символа `'@'`.

Для большей выразительности программ на языке Refal-D точка у безымянных групповых переменных не указывается.

Альтернативой языка Refal-D называется группа, имеющая несколько определяющих образцов (вариантов). В конкретном синтаксисе эти образцы разделяются символом '|' и имеют приоритет согласно порядку перечисления.

```
{ '/*' e. '*/' | '//' e. '\n' | '\n*' e. '\n' }.comment E.other
```

Пример 3: Образец с альтернативой.

После описания типа переменной (например, с помощью имени типа, группы или альтернативы) может стоять пара квадратных скобок с двумя целыми числами или закрытыми `integer`-переменными, разделёнными двумя точками. Такая конструкция называется **повторителем типа** переменной Refal-D и является аналогом квантификатора повторения в регулярных выражениях языка Perl. В общем случае описание переменной с повторителем имеет следующий вид:

[4] Стандартно, термины «открытая» и «закрытая» переменная используются в Рефале только для `e`-переменных. В зависимости от контекста использования, им придаётся синтаксический или семантический смысл, которые не совпадают. Пусть дан рефал-образец `p`. В первом случае `e.v` называется *закрытой* в `p`, если на каждом уровне скобочной структуры (конструктор Рефала), в который входит `e.v`, встречается не более одного вхождения `e`-переменных. В противном случае `e.v` называется *открытой* в `p`. (См. [14, 15].) - *Прим. ред.*

<Описание_типа>[*<intA>*..*<intB>*].*<имя_переменной>*

Если значение *<intA>* больше значения *<intB>*, или одно из них отрицательное, то возникает исключительная ситуация и выполнение программы прерывается с ошибкой[5]. В противном случае, данная конструкция определяет переменную, которая сопоставима с выражением, удовлетворяющим цепочке безымянных переменных типа *<Описание_типа>* длиной от *<intA>* до *<intB>* элементов. В случае неоднозначности, приоритет отдаётся меньшему количеству элементов, однако стоит отдельно рассмотреть случай, когда в качестве описания типа выступает альтернатива.

```
{ 'X' | 'Y' | 'Z' }[2..4].variableName
```

Пример 4: Образец с повторителем.

В процессе сопоставления с образцом, в альтернативе с повторителем приоритет в первую очередь отдаётся выбору варианта, и только потом определению количества элементов. Ниже приведены возможные значения переменной из данного примера, которые указаны в порядке приоритета. Другими словами, если в определенный момент сопоставления с переменной может быть сопоставлено несколько значений из нижеприведённых, то будет выбрано самое первое из них:

'XX' 'XXX' 'XXXX' 'XXXY' 'XXXZ' 'XXY' 'XXYX' 'XXYY' ... 'XXZZ' 'XY' 'XYX' 'XYXX' 'XYXY' ... 'XZZZ' ... 'YX' 'YXX' 'YXXX' ... 'ZZZZ'

Для определения повторителей, не ограниченных сверху, используется следующая конструкция:

<Описание_типа>[*<intA>* **...**].*<имя_переменной>*

Для случая, когда *<intA>* всегда совпадает с *<intB>*, в языке предусмотрено сокращение:

<Описание_типа>[*<intA>*].*<имя_переменной>*

Как и для групп, для безымянных переменных с повторителем точка в конце не требуется.

2.4. Рефал-условия. Рефал-условия впервые появились в Рефале-4 и доказали свою состоятельность во всех последующих диалектах. В Refal-D они имеют тот же синтаксис и семантику, что и в Рефале-5:

, *<результатное_выражение>* **:** *<образец>*

[5] Корректность границ повторения не может быть проверена до начала выполнения программы потому, что в качестве *<intA>* и *<intB>* могут находиться переменные, значение которых может быть получено только во время выполнения программы. - *Прим. ред.*

Одной из выразительных и эффективных конструкций нового диалекта является отрицательное условие. Отрицательные условия определяются с помощью модификатора `$NOT`, который может стоять перед результатным выражением условия или перед образцом условия. Если левая часть имеет отрицательные условия, то она сопоставима с объектным выражением только в тех случаях, когда все эти условия, взятые без модификатора `$NOT`, не выполняются.

```
e.nodig ,
   $NOT e.nodig : e. {'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'|'0'} e.
```

Пример 5: Левая часть для выражения, не содержащего цифр.

2.5. Именованные шаблоны. Уже к появлению первой компьютерной реализации Рефала была выделена проблема гибкости описания образцов, для решения которой в диалекте Рефал-2 появились спецификации переменной и возможность определять их за пределами места применения. Однако эти спецификации позволяли программисту лишь накладывать ограничения на область допустимых значений переменной.

Более гибким и выразительным является механизм построения именованных шаблонов с помощью рефал-образцов и рефал-условий. Для определения именованного шаблона используется оператор `::=` и ключевое слово **Template**:

Template *<имя>* **::=** *<тело_шаблона>* **;**

где *<имя>* - это идентификатор (любой, кроме объявленных типов переменных), а *<тело_шаблона>* - рефал-образец, возможно, с условиями:

<тело_шаблона> ::= <образец> { ,<условие> }[0...]

Определение элемента *<тело_шаблона>* специально приведено в форме, близкой к расширенной форме Бекуса-Наура, чтобы наглядно продемонстрировать сходство именованных шаблонов Refal-D и РБНФ.

Объявление в программе именованного шаблона является объявлением нового типа переменной. С такой переменной может быть сопоставлено лишь то объектное выражение, которое соответствует образцу и условиям из тела именованного шаблона. Переменная, в качестве типа которой указано имя шаблона, далее называется *переменной пользовательского типа.* Стоит отметить, что в теле шаблона могут быть использованы любые другие именованные шаблоны в качестве типов переменных, а так же сам описываемый шаблон. Это добавляет в алгоритмический язык рекурсивных функций ещё один вид рекурсии – рекурсию описания шаблонов.

```
Template GoodBrackets ::=
                  { $empty | '(' GoodBrackets. ')' GoodBrackets. }
```

Пример 6: Шаблон для правильного списка скобок.

В приведённом выше примере определяется именованный шаблон для выражения, состоящего из литер-скобок, расставленных в правильном порядке. В программе с таким определением шаблона становится возможным использование переменных типа `GoodBrackets`:

```
Go {
  $empty,
        <Card> : GoodBrackets.input = <Prout 'Ok'>;
  $empty = <Prout 'Incorrect'>;
}
```

Для удобной работы с переменными пользовательского типа в Refal-D реализована **операция разыменования** (`::`), позволяющая получить ссылку на переменную из тела шаблона по её имени. Синтаксис операции разыменования:

<закрытая_переменная>`::`*<имя_вложенной_переменной>*

Такая ссылка является закрытой переменной, а значит и к ней применима операция разыменования, что даёт возможность строить цепочки разыменований для доступа к вложенным переменным. На следующем примере рассмотрим применение операции разыменования.

```
/* вспомогательные типы переменных */
Template Letter  ::=  {'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'};
Template Id      ::= Letter.head { Id.body |  };
Template Tag     ::= '<' Id.name '>' e.tagBody '</' Id.name '>';
Template TagList ::= { Tag.head TagList.tail |  }.list;

Template Separators ::= {' '|'\n'|'\t'}[1..255].seps;

/* описание некоторого подмножества html */
Template THtml  ::= '<html>' THead.head TBody.body '</html>';
Template THead  ::= '<head>' TTitle.title '</head>';
Template TTitle ::= '<title>' e.value '</title>';
Template TBody  ::= '<body>' TagList.tags '</body>';

$ENTRY Go {
                = <Parse <Clear <Mount 'in.txt'>>>;
}
/* нормализация html-текста: удаление разделителей между тегами */
Clear {
        e.A '>' Separators.'<' e.B   =   e.A '>' '<' <Clear e.B> ;
        e.A   =   e.A ;
}
```

```
Parse {
        THtml.html =
            <Print 'Title:       ' THtml.html::head::title::value>
            <Print 'Body-Head: ' THtml.html::body::tags::list::head>
            <Print 'Body-Tail:  ' THtml.html::body::tags::list::tail>;

        e.else = <Prout 'fail!'>;
}
```

Пример 7: Рефал-программа с использованием именованных шаблонов и операции разыменования.

В данном примере представлен исходный рефал-код программы, которая читает файл `in.txt`, и в случае, если в этом файле текст является некоторым сильно урезанным подмножеством html-языка, выводит на экран его части. В противном случае, программа выводит на экран сообщение: «`fail!`».

```
<html>
  <head>
    <title>Title of the page</title>
  </head>
  <body>
    <div>Some information</div>
    <div>Some other <span>tags</span> </div>
  </body>
</html>
```

Пример 8: Пример файла `in.txt`.

```
Title:     Title of the page
Body-Head: <div>Some information</div>
Body-Tail: <div>Some other <span>tags</span></div>
```

Пример 9: Результат выполнения программы.

Запуск данной программы начинается с вычисления функции `Go`, в которой выполняется чтение текста из файла `in.txt` функцией `Mount`, затем текст обрабатывается функцией `Clean` (`Clean` удаляет из текста несущественные разделители между тегами: пробелы, переносы строк и знаки табуляции), результат которой подаётся на вход функции `Parse`. Здесь файл `in.txt` из Примера 8 был специально подобран таким образом, что функция `Parse` успешно сопоставляет обработанный текст с переменной `THtml.html`. С помощью переменной `THtml.html` и операции разыменования, в соответствующем рефал-предложении можно получать доступ к логически значимым частям значения этой переменной. Так, в рассматриваемом примере, выражение `THtml.html::head` ссылается на значение переменной `THead.head` именованного шаблона `THtml`. Поскольку в описании шаблона `THead` присутствует переменная `TTitle.title`,

а выражение `THtml.html::head` определено шаблоном `THead` (и является закрытой переменной типа `THead`), то к выражению `THtml.html::head` можем применить операцию разыменования для получения значения его переменной `TTitle.title`: `THtml.html::head::title`. В конечном итоге, с помощью цепочки операций разыменования мы добираемся до текста

```
'Title of the page'
```

на который ссылается выражение `THtml.html::head::title::value` (соответствует переменной `e.value` шаблона `TTitle`). Функция `Print` выводит этот текст на экран первой строкой. Следом на экран выводятся значения, на которые ссылаются выражения

```
THtml.html::body::tags::list::head
THtml.html::body::tags::list::tail
```

Так работает операция разыменования переменной.

Стоит отметить, что кроме возможности писать выразительные программы, именованные шаблоны также позволяют определять неэффективные и неправильные алгоритмы. Неправильно спроектированный шаблон может привести к зацикливанию сопоставления пользовательской переменной, например, с пустым выражением. `E-` или `e`-переменная в «удачном» месте может привести к очень длительному определению того, что образец не сопоставим с входным объектным выражением, либо что пользователь допустил опечатку в описании одного из именованных шаблонов. А поскольку ошибка может быть скрыта за длинной цепочкой рекурсивных определений шаблонов, то поиск и устранение такой ошибки может оказаться сложнее решения непосредственно самой задачи на каком-нибудь стандартном языке программирования.

Для того чтобы исключить такие проблемы, на определение именованных шаблонов были наложены некоторые ограничения. Первым ограничением является запрет на использование `E`-переменных в образце тела шаблона (но не в условиях тела шаблона). Стиль программирования на Refal-D с использованием именованных шаблонов подразумевает, что зная о существовании нужного типа переменной (нужного именованного шаблона), пользователю нет необходимости углубляться в его реализацию для использования в своей программе. Одновременно с этим, проектируя образец с использованием переменной нужного типа, программисту необходимо понимать логику процесса сопоставления своего образца с объектными выражениями. Присутствие жадных `E`-переменных в именованном шаблоне означало бы, что переменная этого типа также является жадной. И если о жадности `E`-переменной мы узнаем по имени типа, то о жадности переменной пользовательского типа программист может узнать только обратившись к реализации соответствующего именованного шаблона, что не является концептуально корректным. Поэтому использование `E`-переменных в образце именованного шаблона запрещено, и все переменные пользовательских типов заранее не являются жадными. Что касается условий в теле шаблона, то они не влияют на жадность всего шаблона, поэтому использование в них `E`-переменных вполне безопасно.

Другим ограничением является запрет на использование открытых `e`-переменных в начале и конце образца тела шаблона, поскольку иначе область

допустимых значений пользовательской переменной значительно увеличивается, как и время сопоставления. Конечно, программист может обойти последнее ограничение, маскируя произвольные выражения спецификаторами, однако в этом случае подразумевается, что он знает что делает.

§ 3. Refal-D. Расширение

Базис языка Refal-D содержит в себе языковые конструкции, которые в полной мере позволяют создавать выразительные декларативные программы. Однако в пользу выразительности любого декларативного языка приходится жертвовать некоторой эффективностью вычислений, и Refal-D здесь не является исключением. Мало пользы от понятной и красивой программы, если она выполняется недопустимо долго. В данной части статьи описаны дополнительные конструкции диалекта, с помощью которых программист может повысить эффективность вычисления своих программ без потери выразительности. Функции, предназначенные для метавычислений, так же были отнесены к расширению языка.

3.1. Дополнительные типы переменных. Для того чтобы выяснить, к какому типу данных относится некоторый терм объектного выражения, в предшествующем диалекте - Рефале-5 – имеется функция `Type`, которая возвращает определённые текстовые символы для каждого типа данных. Как правило, программисту приходится писать для её использования вспомогательную функцию, либо использовать рефал-блоки.

В диалекте Refal-D данная задача решается более просто. В языке определено несколько дополнительных встроенных типов переменных для каждого вида рефал-символов:

`integer.`	–	переменная для целого числа;
`real.`	–	переменная для вещественного числа;
`number.`	–	переменная для любого числа;
`symbol.`	–	переменная для текстового символа;
`alpha.`	–	переменная для латинской буквы;
`digit.`	–	переменная для текстового символа-цифры;
`byte.`	–	переменная для байта;

Использование встроенных типов переменных значительно повышает эффективность программ, так как исключает дополнительные вызовы функций. Типы переменных для более специфических видов данных программист может определить с помощью именованных шаблонов.

3.2. Расширенные варианты. Несмотря на озвученный выше отказ от рефал-блоков, в диалекте Refal-D имеется похожий, более ограниченный механизм – альтернативы с расширенными вариантами. Если альтернатива является описанием типа переменной с именем, то любому варианту (или вариантам) этой альтернативы может быть приписана подстановка - некоторое объектное выражение. Если такой вариант окажется успешным при сопоставлении переменной-альтернативы, то в каждом результатном выражении с ссылкой на эту переменную её значение будет заменено соответствующей подстановкой.

В конкретном синтаксисе вариант и его подстановка разделяются символом '='. Для большей наглядности далее приводится пример[6].

```
UnEscape {
    '\\' { '\\' | 'n'='\n'| 't'='\t'| '"'| '\''}.sym E.next
                        = @.sym <UnEscape E.next>;
    '\\' s.sym E.next   = <MakeError>;
    s.s E.next          =  s.s <UnEscape E.next>;
}
```

Пример 10: Использование расширенного варианта.

В данном примере описана функция `UnEscape`, которая на входе принимает объектное выражение в виде последовательности символов, в которой некоторые символы экранированы обратным слешем. На выходе функция возвращает последовательность, в которой экранированные символы заменены на их реальные значения. Обратим внимание на первое предложение функции `UnEscape`. Переменная с именем `sym` определена с помощью альтернативы, состоящей из пяти вариантов: обратного слеша, текстового символа `'n'`, текстового символа `'t'`, текстового символа `'двойная кавычка'`, текстового символа `'одинарная кавычка'` (для обозначения одинарной кавычки потребовалось её экранировать с помощью обратного слеша). В правой части предложения используется значение перменной `sym`. Поскольку для текстовых символов `'n'` и `'t'` в альтернативе определены подстановки, то для этих случаев значением переменной `@.sym` будут соответственно символ `'перенос строки'` и символ `'табуляция'`.

Расширенные варианты не нарушают структуру предложения, одновременно с этим они играют такую же полезную роль, как и рефал-блоки в Рефале-5. Стоит отметить, что в роли подстановок могли бы выступать не объектные, а результатные выражения, что сделало бы язык более мощным, однако автор отказался от данной идеи в пользу выразительности программ.

3.3. Ссылки на объектные выражения. Часто при написании рефал-программ возникает ситуация, когда после распознавания некоторой достаточно сложной структуры в левой части рефал-предложения, в правой части происходит вызов функции, которой в качестве аргумента передаётся часть распознанной структуры или вся структура целиком. Поскольку для работы с этим аргументом необходимо его повторное сопоставление с похожим, а иногда таким же образцом, многие операции повторяются вновь. В случае рекурсивных вычислений количество ненужных операций растет в разы. Для устранения такой избыточности сопоставлений в языке Refal-D предусмотрены ссылки на объектные выражения.

Ссылки – это рефал-символы, которые содержат информацию о структуре объектного выражения и предоставляют быстрый и эффективный доступ к его содержимому. Структура такого объектного выражения определяется именованным шаблоном, а ссылка – переменной пользовательского типа, оснащённой

[6]В данном примере присутствует выражение, которое, как может показаться на первый взгляд, состоит из четырёх одинарных кавычек. На самом деле это двойная кавычка, обрамленная одинарными.

специальной меткой (здесь и далее такие переменные называются ссылочными). В конкретном синтаксисе этой меткой является символ '&' перед именем типа пользовательской переменной.

```
e. MyTemplate.x e. = <X &MyTemplate.x >;
```

Пример 11: Рефал-предложение с определением ссылки на объектное выражение в правой части.

Типом ссылки и *типом ссылочной переменной* будем называть шаблон, определяющий эту ссылку и ссылочную переменную.

Ссылочные переменные могут быть как открытыми, так и закрытыми. Открытые ссылочные переменные успешно сопоставляются только с ссылками того же типа. Закрытые — определяют идентичную ссылку. Две ссылки являются идентичными, если они указывают на одно и то же объектное выражение.

В приведённом выше примере продемонстрировано рефал-предложение, в левой части которого присутствует переменная определённого пользователем типа `MyTemplate`, а в правой части – определение ссылки на значение этой переменной. Когда управление программой будет передано функции `X`, её аргументом будет всего один рефал-символ – ссылка типа `MyTemplate`. Для доступа к значениям переменных шаблона `MyTemplate` в этой ссылке, программист может использовать операцию разыменования (`'::'`). Принцип её действия для ссылочных переменных аналогичен принципу действия для переменных пользовательского типа. Это становится возможным благодаря тому, что в рефал-ссылке сохраняется не только ссылка на объектное выражение, но и ссылка на результат его сопоставления.

```
X {
    &MyTemplate.x e. = &MyTemplate.x::subvar::subsubvar ;
}
```

Пример 12: Использование операции разыменования для ссылочной переменной.

Практическое применение ссылок не вызывает трудностей, однако суть использования одноимённых пользовательских и ссылочных переменных в образцах не очевидна, поэтому ниже приводятся примеры образцов с пояснениями:

Образец		Значение
&Map.v	(Map.v)	ссылка на объектное выражение типа Map и эквивалентное выражение (не ссылка) в скобках
Map.v	&Map.v	выражение и ссылка типа Map на такое же выражение
&Map.v	&Map.v	две ссылки типа Map на одно объектное выражение

Как и другие рефал-символы, ссылки могут быть сопоставлены с s-переменными.

3.4. Отсечение. Именованные шаблоны и спецификаторы предоставляют удобный и достаточно эффективный способ распознавания ожидаемой структуры выражения. Однако если на вход программе подаются некорректные данные, либо программист накладывает в своей программе дополнительные ограничения на значение пользовательской переменной, то могут появиться проблемы в плане эффективности вычисления, которые лучше продемонстрировать на конкретном примере.

Пусть пользователь определил именованный шаблон `TSomeTag` для xml-тега, который называется `'SomeTag'`. У этого тега есть параметр, который называется parameter. Шаблон определён следующим образом[7]:

```
Template TSomeTag ::=
    '<SomeTag parameter="' e.paramValue '">' e.content '</SomeTag>',
    <IsCorrectTags e.content> : true;
```

Попробуем сопоставить образец:

```
e.A TSomeTag.tg e.B
```

с таким текстом (здесь и далее номера строк текста частью текста не являются):

```
1    <html>
2    <SomeTag parameter="tag1">
3                <SomeTag parameter="subtag">
4                Value-sub-tag
5                </SomeTag>
6    </SomeTag>

...  <!-- некоторый xml-код -->

9999 </html>
```

Почти сразу сопоставление завершится успехом, переменной `e.paramValue` шаблона `TSomeTag` будет присвоено значение `'tag1'`, а переменной `e.content`, следующее значение:

```
'<SomeTag parameter="subtag">Value-sub-tag</TAG>'
```

Для корректных входных данных алгоритм отработает успешно. Рассмотрим случай, когда вместо шестой строки предыдущего текста будет такая строка:

```
6    </SomeWrongTag>
```

Очевидно, заданный шаблон не будет успешно сопоставлен с таким текстом. Но прежде чем рефал-машина это выяснит, она переберет все возможные

[7]Согласно синтаксису Рефала, последовательность текстовых символов обрамляется одинарными кавычками. При обрамлении символа «двойная кавычка» может ошибочно показаться, что в коде программы присутствуют тройные кавычки. Здесь одинарные кавычки обозначают границы текста, в то время как двойные кавычки являются частью этого текста.

значения переменных `e.content` и `e.paramValue`, причём значениями `e.paramValue` будут в том числе и те, которые содержат двойные кавычки. При этом переборе каждый раз будет вызываться функция `IsCorrectTags`. Для большого файла такая программа может работать неприемлемо долго. Выход – сократить количество ненужных переборов, используя *отсечение*.

Прежде чем дать определение операции отсечения, рассмотрим ещё одну ситуацию с нашим файлом. Пусть программиста интересуют теги с именем `SomeTag`, у которых значение атрибута `'parameter'` начинается с буквы `'s'`:

```
e.A TSomeTag.tg e.B,   @.tg::paramValue : 's' E.
```

В данном случае, после сопоставления переменной `TSomeTag.tg` со строками 1-6, значение `@.tg::paramValue` не удовлетворит рефал-условию шаблона, что приведет к откату сопоставления. Данный откат дойдет до переменной `e.content`, после чего рефал-машина продолжит её сопоставлять (очевидно, это будут лишние вычисления). Затем откат вернется к переменной `e.paramValue`, которая так же продолжит сопоставляться от предыдущего состояния. Для повышения эффективности вычисления, в этом случае мы тоже воспользуемся отсечением.

Отсечением называется метка в образце (обозначается `'!'`), которая объявляет всю левую от неё часть этого образца единожды сопоставимой. Для рефал-машины это означает, что при откате сопоставления выражения с образцом к этой метке весь текущий образец объявляется несопоставимым с выражением. Обратите внимание: отсечение, определённое в образце, действует только на этот образец, а не на всю левую часть с этим образцом! Рассмотрим назначение и логику отсечения на конкретном примере вычисления некоторой рефал-программы, часть которой представлена ниже.

На вход программе подаётся следующий текст:

```
1    <html>
2    <SomeTag parameter=”tag1”>
3                <SomeTag parameter=”subtag”>
4                Value-sub-tag
5                </SomeTag>
6    </SomeTag>
...
9999 </html>
```

Программа:

```
Template TSomeTag ::=
    '<SomeTag parameter="' e.paramValue '">' ! e.content '</SomeTag>',
    <IsCorrectTags e.content> : true ;

SomeFunction {
    e.A TSomeTag.tg e.B,    @.tg::paramValue : 's' E.  =  ... ;
    ...
}
```

В продемонстрированном примере рефал-машина не будет пытаться наращивать значение переменной `e.paramValue` после того, как она примет значение `'tag1'`, поскольку этого не допустит отсечение. Вместо этого результат сопоставления всего образца шаблона `TSomeTag` будет признан неуспешным, вычисление выражения откатится от переменной `TSomeTag.tg` к переменной `e.A`, которая расширится до следующей подстроки `'SomeTag'` текста, с которым успешно будет сопоставлена переменная `TSomeTag.tg`.

Предыдущий пример демонстрирует работу отсечения внутри образца, однако на практике такой код использовать не стоит, поскольку проблема с откатом переменной `e.content` в образце остаётся в этом случае не решённой. Более правильным решением является отсечение всей левой части в теле шаблона `TSomeTag`:

```
Template TSomeTag ::=
    '<SomeTag parameter="' e.paramValue '">' e.content '</SomeTag>',
    <IsCorrectTags e.content> : true,
    ! ;
```

В данном примере символ отсечения ставится не внутри образца, а в качестве своеобразного условия.

Стоит отметить, что необходимость в отсечениях как правило возникает при использовании `e-`, `E-` и `s[...]`- переменных[8] в шаблонах, однако синтаксис и семантика языка допускают использование отсечений в любых левых частях программы.

3.5. Динамическое доопределение программ. В самом первом диалекте языка рефал, который тогда ещё назывался Метаалгоритмическим языком, была определена возможность занесения из поля зрения в поле памяти любого набора рефал-предложений, что фактически равносильно самомодификации программы. Далеко не все компьютерные реализации языка имеют

[8] Тип переменной `s[...]` иначе можно записать как `{s.}[...]`.

такую возможность (примером реализации динамического доопределения является диалект FLAC [5]). В связи с активным применением языка в метавычислениях и задачах искусственного интеллекта, динамическое изменение рефал-программ становится все более необходимым.

В диалекте Refal-D имеется возможность переопределять функции и именованные шаблоны программы во время её выполнения. Это достигается с помощью встроенной функции `<Refal e.>`, которой в качестве аргумента передаётся текст (список текстовых символов) определения функции или шаблона. Поскольку этот текст может генерироваться во время выполнения программы — у программиста появляется возможность написания полиморфного рефал-кода.

§ 4. Заключение

Refal-D – это предложенное автором статьи направление в сторону решения проблемы отсутствия современной и концептуально непротиворечивой реализации диалекта Рефала, необходимость в которой назрела уже давно. Задачи обработки данных на уровне глубокого анализа их структур из-за отсутствия достаточно простых, но мощных инструментов, принято относить к классу экзотических и сложно реализуемых популярными инструментами. Описанный в статье язык программирования позволяет работать со сложными структурами более естественным способом[9] – с помощью конкретизации с гибкой системой шаблонов.

Одним из приоритетных направлений, на которое в первую очередь обращал внимание автор при проектировании и реализации диалекта, являются метавычисления. Рефал изначально ориентирован на работу с алгоритмическими языками, однако его практическое применение выявило ряд слабых мест, включая слабый способ определения образцов. Спецификаторы и именованные шаблоны расширяют Рефал выразительными инструментами, сохраняя основные концепции языка, а функции динамического доопределения программ дают пользователю возможность создавать более гибкие алгоритмы. Внешние шаблоны с условиями, очевидно, упрощают описание сложных инвариантов в метапрограммировании, что может оказаться полезным для суперкомпиляции.

В то же время стоит отметить, что описание языка и его первая реализация являются лишь началом процесса, в ходе которого некоторые аспекты и принципы могут быть пересмотрены для достижения его полной замкнутости и концептуальной непотиворечивости.

Результаты применения диалекта Refal-D в этой и других областях планируется изложить в следующих работах.

---

[9] См. замечание в Предисловии к данному Сборнику о возможном применении диалекта Refal-D для обработки интернет-документов.

В 2001 году в г. Переславле-Залесском была разработана начальная версия прототипа системы поддержки Рефала-5 как скрипт-языка для преобразования HTML-страниц. Публикаций об этой работе не последовало.

В 2002 г. Л. Ф. Белоус реализовал систему Refal-PHP [17], которая представляет собой интерфейс системы Рефал-5 со скрипт-языком PHP. - *Прим. ред.*

Список литературы

[1] А. А. Марков, “Теория алгорифмов”, *Труды Математического института им. В. А. Стеклова*, Изд-во АН СССР, 1954.

[2] Ан. В. Климов, С. А. Романенко, В. Ф. Турчин, *Компилятор с языка Рефал*, ИПМ АН СССР, М., 1972.

[3] Ан. В. Климов, С. А. Романенко, *Система программирования Рефал-2 для ЕС ЭВМ. Описание входного языка*, ИПМ им.М.В.Келдыша АН СССР, М., 1987.

[4] Р. Ф. Гурин, С. А. Романенко, *Язык программирования Рефал Плюс*, Интертех, М., 1991.

[5] Е. А. Гайдар, И. М. Игнатович, В. Ф. Козадой, А. П. Немытых, В. А. Пинчук, С. В. Чмутов, “Функциональный язык для алгебраических вычислений FLAC”, *Сборник трудов по функциональному языку программирования Рефал, том I*, Издательство «СБОРНИК», Переславль-Залесский, 2014, 11–42.

[6] Н. Н. Непейвода, Н. Н. Скопин, *Основания программирования*, Институт компьютерных исследований, Москва - Ижевск, 2003.

[7] С. А. Романенко, *Рефал-4 - расширение Рефала-2, обеспечивающее выразимость результатов прогонки*, ИПМ им. М. В. Келдыша АН СССР, М., препринт № 147, 1987.

[8] Турчин В.Ф., “Метаалгоритмический язык”, *Кибернетика*, 1968, 45–54.

[9] Турчин В.Ф., “Метаязык для формального описания алгоритмических языков”, *Цифровая вычислительная техника и программирование*, М., 1966, 116–124.

[10] В. Ф. Турчин, “Базисный РЕФАЛ. Описание языка и основные приемы программирования (методические рекомендации)”, *Фонд алгоритмов и программ в отрасли «Строительство»*, **5**, № 33, ЦНИПИАСС, М., 1974.

[11] В. Ф. Турчин, “Эквивалентные преобразования программ на Рефале”, *Автоматизированная система управления строительством*, ЦНИПИАСС, М., 1974, 36–68.

[12] M. Sh. Islamov, *Dialect Refal-D: Interpretator of Refal program language with user-type templates (EBNF-like form) and dynamical evalution*, ([executable module]: https://code.google.com/p/refal51/), 2010.

[13] Ark. V. Klimov, *Refal-6 http://www.refal.org/~arklimov/refal6*, 2003.

[14] V. F. Turchin, *REFAL-5, Programming Guide and Reference Manual*, New England Publishing Co., Holyoke, 1989.

Ниже следуют ссылки, добавленные редактором.

[15] V. F. Turchin, *REFAL-5 programming guide and reference manual*, (переработанное и расширенное издание доступно как zipped html-файл: http://refal.botik.ru/book/refal-book-html.zip), 1999.

[16] В. Ф. Турчин, Д. В. Турчин, А. П. Конышев, А. П. Немытых, *Рефал-5: исполняемые модули и исходные тексты*, ([online]: http://www.botik.ru/pub/local/scp/refal5/), 2000.

[17] Л. Ф. Белоус, *Система программирования Refal-PHP*, ([online]: http://www.refal.net/~belous/rphpintr.htm), 2002.

М. Ш. Исламов (M. Sh. Islamov)
г. Ижевск, УдГУ
E-mail: islamov.marat@gmail.com

А. Н. Непейвода

Верификация пинг-понг протоколов в модели префиксных грамматик с помощью суперкомпиляции

Описывается способ моделирования двух- и многосторонних пинг-понг протоколов префиксными грамматиками, позволяющий свести их верификацию к решению задачи пустого слова. Далее показано, как эта задача может быть разрешена с помощью алгоритма развертки, используемого инструментами преобразования программ. Алгоритм разрешения основан на отношении подпоследовательности и согласован с алгоритмом верификации протоколов Долева–Ивена–Карпа с помощью конечных автоматов.

Библ. 20 наим.

Ключевые слова: криптографические протоколы, верификация, пинг-понг протоколы, суперкомпиляция, префиксные грамматики, конечные автоматы.

Содержание

Исследование выполнено при финансовой поддержке РФФИ в рамках научного проекта № 14-07-00133 а.

Введение

Один из самых частых видов угроз при передаче секретных данных по открытому каналу – это угроза типа «человек в середине»: некто выдает себя за одного из участников передачи и от его имени отправляет сообщения другим так, что они не могут установить факт подмены. Таким образом можно получить доступ к зашифрованным данным, даже не пытаясь взломать сам шифр. Чтобы избежать подобного рода угроз, предлагается использовать криптографические протоколы — они устанавливают правила поведения пользователей при обмене сообщениями так, чтобы злоумышленнику было как можно труднее выдать себя за правомерного пользователя протокола. Не случайно мы пишем здесь «как можно труднее»: общая задача проверки того, имеет ли протокол уязвимости, неразрешима [3], поэтому о его надежности можно говорить лишь в узком смысле, имея в виду ограничения на тип используемых операций и поведение злоумышленника. В частности, разрешим вопрос о надежности пинг-понг протоколов в модели угрозы Долева-Яо.

Пинг-понг протокол требует, чтобы передаваемое секретное сообщение было представлено единственной строкой, изменяемой посредством конечного множества операций. Участники протокола по очереди применяют операции этого множества к сообщению и пересылают друг другу результат, как будто подавая мяч при игре в пинг-понг. Долев, Яо и Карп доказали, что если ограничить действия злоумышленника прослушиванием канала и подменой сообщений, пересылаемых по нему, то вопрос о безопасности пинг-понг протокола алгоритмически разрешим [5]. В работе [7] предложено обобщение процедуры разрешения для пинг-понг протоколов с произвольным количеством участников.

Хотя пинг-понг протоколы удобно верифицировать с помощью специализированных программ, в настоящее время известно много случаев разрешения подобных задач верификации с помощью инструментов преобразования программ общего назначения. Так, протокол Нидхэма–Шредера оказалось возможно проверить на угрозы с помощью суперкомпиляции [4]. Более того, выяснилось, что преобразователи программ способны проводить полный анализ на уязвимости некоторых известных классов моделей протоколов. В частности, суперкомпиляцией можно установить надежность протоколов когерентности кэша [13]. Поэтому хотелось бы надеяться, что можно построить и модель пинг-понг протоколов, верифицируемую с помощью суперкомпиляции. И главное затруднение здесь таково, что набор ситуаций, возникающих при передаче сообщения по протоколу, неограничен, поскольку один и тот же протокол может использоваться сколько угодно раз (именно эта черта чаще всего и влечет наличие атак типа «человек в середине»).

Данная работа описывает способ верификации пинг-понг протоколов с помощью произвольного инструмента преобразования программ, способного:

1. разворачивать дерево путей, порождаемых программной моделью,
2. обрывать слишком длинные ветви в этом дереве согласно условию гомеоморфного вложения («похожести» одной конфигурации на ветви вычислений на другую, находящуюся в вершине–предке первой [18]).

В параграфе 1 приводится определение пинг-понг протокола для многих участников и обсуждается понятие атаки на протокол – тем самым устанавливается точное описание класса протоколов, которые нужно уметь верифицировать. В параграфе 2 приводится классический алгоритм верификации протоколов интересного нам класса. В параграфе 3 мы описываем способ моделирования пинг-понг протоколов префиксными грамматиками и сравниваем выразительную силу полученных моделей с классическими моделями пинг-понг протоколов, представляемыми в виде конечного автомата. После этого предлагаем алгоритм верификации посредством развертки дерева возможных путей, порождаемых грамматикой, и обосновываем формальную корректность этого алгоритма. Наконец, оцениваем сложность алгоритма верификации и обсуждаем некоторые методы по уменьшению времени его работы.

Верификация примеров, приведенных в этой статье, была экспериментально проведена на суперкомпиляторе SCP4 [1].

§ 1. Пинг-понг протоколы

1.1. Определение пинг-понг протокола. Криптографический протокол — множество правил, определяющее поведение участников процесса обмена сообщениями в сети (например, этот обмен может быть процессом аутендификации или передачей засекреченных сведений). Ниже мы дадим его формальное определение.

Рассмотрим процесс обмена данными по общей сети между несколькими участниками. Данные представляют собой слова в конечном алфавите; множество таких слов обозначаем $\mathbb{S}$. Множество всех пользователей сети обозначим Usr. Назовем словарем A (обозначается Σ_A) множество операторов $f : \mathbb{S} \to \mathbb{S}$, доступных $A \in \mathrm{Usr}$. Пустое (тождественное) действие обозначим Λ; а композицию операторов $g_B \in \Sigma_B$ и $f_A \in \Sigma_A$ как $f_A \circ (g_B)$ (сокращенно $f_A(g_B)$ или просто $f_A g_B$, если эти операторы – функции).

Определение 1. Пусть дана функция от двух переменных $F : X \times Y \to S$ и x_0 из X. Ограничение F на множество $\{x_0\} \times Y$ называется *сечением функции* F по первому аргументу. Обозначим его F_{x_0}.

Пусть дана функция $f : \mathrm{Usr} \times \mathbb{S} \to \mathbb{S}$. Сечение этой функции $f_{x_0} : \mathbb{S} \to \mathbb{S}$, где x_0 из Usr, назовем *параметризованным оператором*. Таким образом, параметризованный оператор f_x зависит от параметра x. Множество всех параметризованных операторов[1] обозначим **VO**.

Иначе говоря, если $y \in \mathbb{S}$, $f_x(y)$ есть оператор от двух переменных $f(x, y)$, специализированный по первому аргументу $x \in \mathrm{Usr}$.

Ниже приведены часто используемые в криптографии параметризованные операторы:

- Оператор шифрования открытым ключом E_x. $E_x(Y)$ зашифровывает Y ключом пользователя x;
- Оператор расшифровки D_x. $D_x(Y)$ расшифровывает Y ключом пользователя x;

[1]Элементы этого множества будем называть (просто) операторами, если это не приводит к недоразумениям.

- Оператор приписывания имени a_x. $a_x(Y)$ приписывает имя пользователя x к Y;
- Оператор удаления имени d_x. $d_x(Y)$ стирает префикс Y, соответствующий имени пользователя x.

D_x обратен E_x: выполнено как $D_xE_x = \Lambda$, так и $E_xD_x = \Lambda$ — допустимо как расшифровывать уже наложенный шифр, так и предотвращать шифрование. d_x является левым обратным для a_x, так что выполнено $d_xa_x = \Lambda$, и в то же время $\forall F(F \in \mathbf{VO} \Rightarrow Fd_x \neq \Lambda)$[2]. E_x, a_x, d_x — открытые, доступные всем пользователям сети операторы, а D_x — секретный, формально

$$\forall x, y \in \mathrm{Usr}(E_x \in \Sigma_y \,\&\, a_x \in \Sigma_y \,\&\, d_x \in \Sigma_y \,\&\, D_x \in \Sigma_x \,\&\, (x \neq y \Rightarrow D_x \notin \Sigma_y)).$$

Множество **VO** может содержать и другие параметризованные операторы. Например, в статье [7] **VO**, помимо вышеуказанных, включает еще F_x и G_x, где F_x и G_x — взаимно обратные операторы перестановки фрагментов строки. Некоторые операторы (например, зашифровка, расшифровка, приписывание одной буквы и т. п.) являются элементарными: действия, соответствующие таким операторам, не могут быть разложены в композицию более простых. Другие элементы из Σ_x могут быть представлены как композиции операторов, причем сами эти операторы не обязательно должны принадлежать Σ_x. Например, такое может случиться, если участник переписки использует некоторую программу шифрования, не позволяющую применять шифрующий алгоритм без приписывания к сообщению персонального идентификатора. Составные операторы из Σ_x записываются как последовательности букв (при отсутствии ассоциативности слева — последовательности скобок и букв), соответствующих элементарным.

Многие часто используемые операторы обладают следующим важным свойством.

Если всякие два элемента **VO** a, b могут удовлетворять лишь равенствам вида $ab = \Lambda$, и кроме того, $\forall a, b, c(a(bc) = (ab)c)$, то такую структуру **VO**, вслед за работой [7], будем называть полусвободной алгеброй[3].

Свойство полусвободы упрощает построение модели протокола; в то же время, оно имеет практический смысл, поскольку выполнено для классических криптографических моделей с открытыми и закрытыми ключами. Если рассматривать более широкий класс операторов, включая универсальные ключи расшифровки, являющиеся обратными более, чем к одному ключу зашифровки (как в Примере 11), их алгебра уже не будет полусвободной. Однако неассоциативность операторной алгебры чаще всего не мешает опускать скобки при записи композиций ее элементов. К примеру, если администратор имеет доступ к универсальному ключу U, который способен расшифровать либо предотвратить шифрование ключами E_A и E_B, так что $U(E_B(M)) = E_A(U(M)) = M$,

[2]Строго говоря, d_x является частично определенным оператором на $\mathrm{Usr} \times \mathbb{S}$. Однако в модели не предполагается, что применение d_x к слову, начинающемуся не на a_x, обязательно приводит к ошибке: этого может и не произойти. Мы лишь оговариваем, что такое применение приведет к тому, что все операторы, примененные к строке до «ошибочного» применения d_x, станет невозможно сократить.

[3]Поскольку равенства вида $a \circ b = \Lambda$ допустимы, $\langle \mathbf{VO}, \circ \rangle$ не является свободной полугруппой; поскольку не у каждого элемента F есть обратный, свободной группой $\langle \mathbf{VO}, \circ \rangle$ также не является.

то последовательность $E_A(U(E_B(M)))$ необходимо эквивалентна $E_A(M)$, а не $E_B(M)$. Порядок применения ключей, начиная с самого внутреннего в композиции, предустановлен и не допускает альтернатив.

Используем эту особенность. Потребуем, чтобы операторы — элементы рассматриваемой алгебры могли удовлетворять только равенствам вида

$$\forall F \in \mathbf{VO}^*(a_1(a_2(\dots(a_n(F))\dots)) = F).$$

В дальнейшем, равенства такого вида будут называться *стирающими правилами*, а их применения — *сокращениями*.

Кроме того, если в алгебре имеются два различных стирающих правила вида $\forall F \in \mathbf{VO}^*(a_1(a_2(\dots(a_n(F))\dots)) = F)$ и $\forall F \in \mathbf{VO}^*(b_1(b_2(\dots(b_m(F))\dots)) = F)$, мы предполагаем, что слово $a_1a_2\dots a_n$ не является подсловом слова $b_1b_2\dots b_m$. С учетом этих ограничений опишем точный порядок применения стирающих правил к композиции операторов.

Определение 2. Композиция $a_1(a_2\dots(a_n(M))\dots))$ операторов, где оператор M нульместен (является константой), находится в *нормальной форме*, если к ней не применимо ни одно стирающее правило.

Алгоритм 1. Любую композицию $a_1(a_2\dots(a_n(M))\dots))$ можно привести к нормальной форме следующим образом.

1. Назначим $k_1 = n$, $k_2 = n$.
2. Если $k_1 > 0$, ищем стирающее правило для a_{k_1}, применимое к данной композиции.
   (a) Если нашлось стирающее правило

   $$\forall F \in \mathbf{VO}^*(a_i(\dots(a_{k_1}(F))\dots) = F),$$

   причем его длина равна L, применить его, уменьшить k_2 на L, присвоить переменной k_1 значение k_2 и вернуться к шагу (2).
   (b) Если правил вида $\forall F \in \mathbf{VO}^*(a_i(\dots(a_{k_1}(F))\dots) = F)$ не нашлось, уменьшить k_1 на 1.

Алгоритм Определения 2 однозначно определяет нормальную форму для всякой последовательности операторов в силу введенных ограничений. Скобки в последовательностях операторов всегда сгруппированы вправо, и с введением точного порядка применения стирающих правил теперь можно записывать эти последовательности в виде обычных строк, не боясь неточностей. Стирающие правила далее будем записывать как $a_1a_2\dots a_n = \Lambda$, а не как $\forall F \in \mathbf{VO}^*(a_1(a_2(\dots(a_n(F))\dots)) = F)$.

В нашей работе допущение полусвободы не считается выполненным по умолчанию, однако в примерах, использующих классическую алгебру операторов $\{E_x, D_x, a_x, d_x\}$, оно выполняется, о чем мы более не напоминаем.

Определение 3. *p-сторонний пинг-понг протокол* $P[x_1,\dots,x_p]$ — это последовательность пар $((y_1, \alpha_1[x_1,\dots,x_p]),\dots,(y_l, \alpha_l[x_1,\dots,x_p]))$, где p — число

участников, y_i — переменная из Usr и $\alpha_i[x_1, \ldots, x_p]$ — последовательность параметризованных операторов из словаря пользователя y_i, находящаяся в нормальной форме. Здесь мы пишем $\alpha_i[x_1, \ldots, x_p]$ с перечислением всех переменных из Usr, от которых зависят элементы α_i. Частные случаи α_i будем писать без квадратных скобок.

Поясним, как это определение работает. Пусть несколько участников U_1, U_2, …, U_n обмениваются данными по открытому каналу связи. Обозначим исходные (секретные) данные буквой M. Если U_1, U_2, …, U_n используют p-сторонний протокол $P[x_1, \ldots, x_p]$ (где $p \leqslant n$), то обмен сообщениями происходит нижеописанным способом.

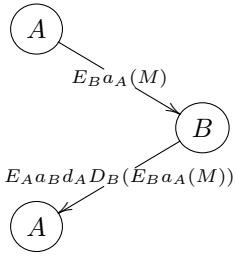
Взаимодействие начинается с пересылки пользователем U_1 пользователю U_2 сообщения $\alpha_1(M)$ (напоминаем, что $\alpha_1 \in \Sigma^*_{U_1}$). Затем U_2 посылает U_3 сообщение $\alpha_2(\alpha_1(M))$ (где опять же $\alpha_2 \in \Sigma^*_{U_2}$) и так далее вплоть до момента, когда сообщение $\alpha_{p-1}(\ldots\alpha_1(M))$ достигает U_p. После чего U_p применяет к нему последовательность α_p и пересылает результат всем пользователям U_1, U_2, …, U_n (если $n > p$, то U_{p+1}, …, U_n участвуют в протоколе лишь как получатели).

Пример 1. Опишем двухсторонний пинг-понг протокол, представляющий собой модификацию безопасного протокола из работы [6] в терминах Определения 3.

$$P_2[x_1, x_2] = ((x_1, E_{x_2}a_{x_1}), (x_2, E_{x_1}a_{x_2}d_{x_1}D_{x_2}))$$

Напомним, что $d_x a_x = \Lambda$ и $D_x E_x = E_x D_x = \Lambda$. Так что x_2 сначала отменяет все операции, совершенные x_1, и затем применяет $E_{x_1}a_{x_2}$ к исходному сообщению (такой порядок действий позволяет x_2 прочитать исходное сообщение, что, впрочем, не требуется протоколом).

Пусть x_1 — это A, а x_2 — B. Тогда $P_2[A, B]$ можно схематично представить следующей схемой. Стрелки помечены сообщениями, пересылаемыми пользователями друг другу.

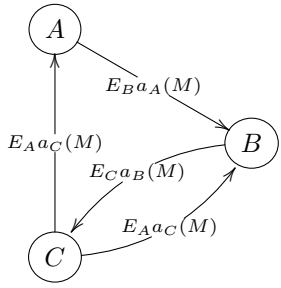


Здесь мы в целях наглядности подписали стрелки полными наборами операторов, примененных к M. В дальнейшем на стрелках надписываются нормализованные варианты строк операторов.

Пример 2. Теперь рассмотрим трехсторонний аналог протокола $P_2[x_1, x_2]$.

$$P_3[x_1, x_2, x_3] = ((x_1, E_{x_2}a_{x_1}), (x_2, E_{x_3}a_{x_2}d_{x_1}D_{x_2}), (x_3, E_{x_1}a_{x_3}d_{x_2}D_{x_3}))$$

$P_3[A, B, C]$ будет изображаться так:



Можно ли сказать, что $P_3[x_1, x_2, x_3]$ безопасен, как и $P_2[x_1, x_2]$? Чтобы поставить вопрос о безопасности этого протокола, необходимо уточнить понятие поведения злоумышленника.

1.2. Определение понятия атаки на протокол. Важно учесть, что протокол может быть применен одной и той же группой участников многократно (либо разными группами, в которые участники могут входить повторно). Так что каждый пользователь, исполняющий протокол, должен быть соотнесен с двумя сущностями: своим постоянным идентификатором и временной «ролью», назначаемой в соответствии с протоколом, чтобы ограничить действия пользователя в его рамках. Для этого и потребуются параметризованные операторы. Их обычно не используют в классическом двухстороннем случае, поскольку для каждого протокола $P[x, y]$ можно ограничиться $\mathrm{Usr} = \{A, B, I\}$ (I — злоумышленник), и все варианты применения $P[x, y]$ к этому множеству пользователей легко перечисляются (их шесть). Мы представляем двухсторонние протоколы в терминах параметризованных операторов из соображений единообразия.

Определение 4. Пусть $u = (U_1, U_2, \ldots, U_p)$ — последовательность p элементов из Usr и $P[x_1, \ldots, x_p]$ — p-сторонний пинг-понг протокол. *u-подстановка в протокол* $P[x_1, \ldots, x_p]$ (обозначаемая как $P[U_1, \ldots, U_p]$) — это результат замены переменной x_j идентификатором пользователя U_j для всех $1 \leqslant j \leqslant p$. Слово $\alpha_j[U_1, \ldots, U_p]$ назовем *u-подстановкой в слово* $\alpha_j[x_1, \ldots, x_p]$.

u-подстановку в p-сторонний пинг-понг протокол назовем *корректной*, если u состоит из p различных элементов Usr.

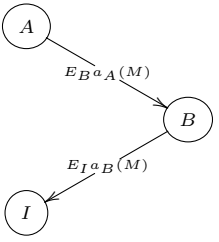
Теперь определим понятие атаки на протокол. Нижеприведенное определение совпадает с классическим; позже мы внесем в него изменения, которые будут полезны в дальнейшем.

Определение 5. Пусть пинг-понг протокол $P[x_1, \ldots, x_p]$ состоит из параметризованных слов $\alpha_1[x_1, \ldots, x_p]$, $\alpha_2[x_1, \ldots, x_p]$, …, $\alpha_l[x_1, \ldots, x_p]$. Зафиксируем некоторую $u = (U_1, U_2, \ldots, U_p)$ — последовательность p различных пользователей, и обозначим множество этих пользователей Usr'.

Пусть $J \subseteq \mathrm{Usr}$. Тогда $\Sigma_J = \bigcup_{j \in J} \Sigma_j$. Множество всех корректных подстановок в слова протокола P пользователей из множества J обозначим $\mathrm{INST}(P, J)$.

Скажем, что протокол $P[x_1, \ldots, x_p]$ *сильно ненадежен* в модели угрозы по Долеву-Яо, если существует множество S и строка операторов ξ такие, что $S \subseteq \mathrm{Usr} \setminus \mathrm{Usr}'$, $\xi \in (\Sigma_S \cup \mathrm{INST}(P, \mathrm{Usr}))^*$ и $\xi\alpha_1[U_1, \ldots, U_p] \equiv \Lambda$.

Пример 3. Протокол $P_2[x_1, x_2]$ надежен в модели угрозы по Долеву-Яо, и можно допустить, что $P_3[x_1, x_2, x_3]$ также надежен. Однако рассмотрим следующую последовательность передач (I — злоумышленник, A и B — правомерные участники протокола). Злоумышленник выдает себя за x_3, так что B зашифровывает сообщение ключом $E_{x_3} = E_I$.



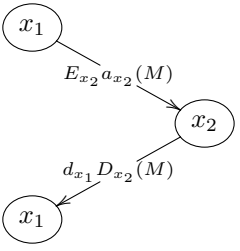
Словарь злоумышленника содержит[4] d_B и D_I, поэтому он может получить Λ из $E_I a_B$. При этом до третьего правомерного участника протокола сообщение вообще не доходит. $P_3[x_1, x_2, x_3]$ оказывается сильно ненадежным протоколом.

В статье [7] S. Even и O. Goldreich также вводят понятие слабой ненадежности. Слабо ненадежные протоколы допускают уязвимости, в которых могут быть произвольные (а не только корректные) подстановки. Уязвимость протокола в слабом смысле не влечет уязвимости в сильном — в работе [7] это показано на Примере 4.

Пример 4. Рассмотрим следующий двухсторонний протокол

$$P_{WI}[x_1, x_2] = ((x_1, E_{x_2} a_{x_2}), (x_2, d_{x_1} D_{x_2})),$$

который можно представить схемой



Этот протокол надежен в сильном смысле — выходное значение переменного оператора $d_{x_1} a_{x_2}$ никогда не будет Λ. Но в слабом смысле $P_{WI}[x_1, x_2]$ уязвим: если в переменные x_2 и x_1 в слове $d_{x_1} D_{x_2}$ подставить одного и того же пользователя U_2, то $\alpha_2[U_2, U_2]\alpha_1[U_1, U_2]$ превратится в пустое слово, и нарушителю не нужно будет прилагать никаких усилий, чтобы узнать пересылаемые данные.

[4] Согласно упомянутым свойствам приватности и публичности операторов d_x и D_x.

Рассмотрение слабых атак вместо сильных выгодно с точки зрения вычислительной сложности тем, что позволяет обойтись множеством нарушителей, состоящим из единственного элемента. Но S. Even и O. Goldreich в работе [7] подчеркивают, что такое понятие уязвимости, в противоположность уязвимости в сильном смысле, неестественно. Безусловно, Пример 4 демонстрирует как раз такую неестественную атаку: второй участник протокола (подразумевается, что этот участник легален) как будто бы посылает сообщение сам себе.

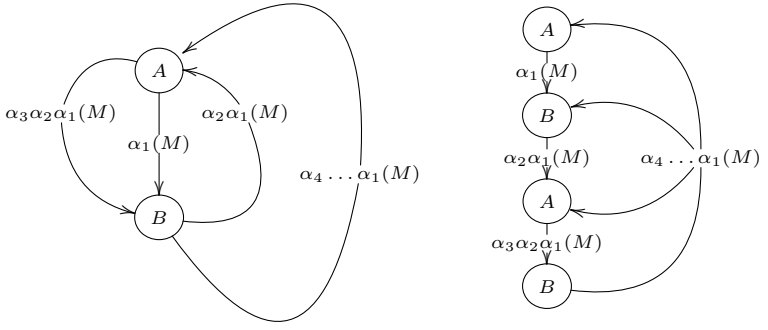
Если сузить понятие слабой атаки, чтобы оно не допускало:

1. подстановки активного легального пользователя ни в какие другие роли,
2. и подстановки никаких других пользователей в роль активного пользователя,

это понятие станет более сообразно практике. В частности, такая слабая модель атаки позволяет свести классический случай двухсторонних протоколов к частному случаю многосторонних.

Подстановки, запрещающие подмену роли активного пользователя, назовем *слабо корректными*.

Пример 5. Слева — общий вид двухстороннего пинг-понг протокола с четырьмя этапами передач. Справа его многосторонний аналог, допускающий слабо корректные подстановки. Заметим, что на последнем этапе B будто бы посылает сообщение «сам себе», но на самом деле здесь нет противоречия с определением слабо корректной подстановки, поскольку задача последнего пользователя — вернуть сообщение первому пользователю, а второй и третий пользователи получают его попутно и в подстановке в α_4 отсутствуют.



Так же, как и исходное понятие слабой атаки, сужение слабой атаки на слабо корректные подстановки позволяет обходиться одноэлементным множеством нарушителей (что уменьшает размер модели) и позволяет находить атаки, не обнаруживаемые алгоритмом поиска атаки в сильном смысле.

Пример 6. Рассмотрим многосторонний протокол $P_D[x_1, x_2, x_3, x_4, x_5]$ следующего вида. Пользователи x_4 и x_5 — только получатели. Пользователь, пересылающий первое сообщение — это администратор, так что для всякого $\alpha_1[\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{G}, \mathbf{H}]$, $D_{\mathbf{H}} \in \Sigma_{\mathbf{A}}$ и $D_{\mathbf{B}} \in \Sigma_{\mathbf{A}}$. Все прочие пользователи, как обычно, умеют расшифровывать лишь то, что зашифровано их ключом.

$\alpha_1 = E_{x_2} a_{x_3} a_{x_4} a_{x_4} D_{x_5}$

$\alpha_2 = E_{x_3} a_{x_2} a_{x_4} a_{x_5} E_{x_2} d_{x_4} d_{x_4} d_{x_3} D_{x_2}$

$\alpha_3 = E_{x_1} d_{x_5} d_{x_4} d_{x_2} D_{x_3}$

В сильной модели атаки $P_D[x_1, x_2, x_3, x_4, x_5]$ надежен. Действительно, рассмотрим $\alpha_1 = E_{\mathbf{B}} a_{\mathbf{C}} a_{\mathbf{G}} a_{\mathbf{G}} D_{\mathbf{H}}$. Единственная возможность избавиться от $E_{\mathbf{B}}$ — применение $\alpha_3[x_1, x_2, \mathbf{B}, x_4, x_5]$ к α_1 (поскольку применение $\alpha_2[x_1, \mathbf{B}, x_3, x_4, x_5]$ порождает опять $E_{\mathbf{B}}$). Поскольку $x_5 \neq x_4$, $d_{x_5} d_{x_4}$ никогда не будет обратным к $a_{\mathbf{G}} a_{\mathbf{G}}$. Однако используя модель атаки со слабо корректными подстановками, легко построим атаку: $\alpha_3[\mathbf{H}, \mathbf{C}, \mathbf{B}, \mathbf{G}, \mathbf{G}]\alpha_1[\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{G}, \mathbf{H}] = \Lambda$.

В дальнейшем будем говорить о слабых моделях атак в вышеописанном узком смысле.

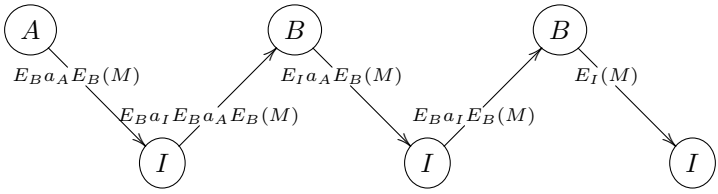
1.3. О целях задачи верификации. Заметим, что не в каждом протоколе единственной секретной информацией является начальное сообщение M и не во всех протоколах оно вообще считается секретным. Например, в протоколе Нидхэма–Шредера [12] $P_{NS}[x_1, x_2]$ факт получения злоумышленником начального случайного числа, сгенерированного x_1, не является свидетельством атаки. Зато атакой будет последовательность действий, позволяющая злоумышленнику получить случайное число, сгенерированное x_2 в качестве ответа на сообщение x_1. Учитывая это наблюдение, можно попробовать охватить чуть более широкий спектр протоколов за счет расширения множества секретных слов, которые не должны попасть к злоумышленнику (в классических пинг-понг протоколах это множество есть $\{\Lambda\}$).

Определение 6. Пусть дан протокол $P[x_1, \ldots, x_p]$ и множество секретных слов INSEC (также называемое *множеством уязвимостей*). Злоумышленника обозначим I. Назовем *обобщенной слабой атакой* на $P[x_1, \ldots, x_p]$ последовательность $\xi \in \{\Sigma_I \cup \{\alpha_i[y_1, \ldots, y_p] | \forall j (y_j = x_i \Leftrightarrow j = i)\}^*$ такую, что $\beta \in$ INSEC и $\xi\alpha_1[U_1, \ldots, U_p] \equiv \beta[U_1, \ldots, U_p]$.

Теперь, когда мы определились с понятием атаки, нужно разобраться с вопросом, что будет пониматься под верификацией протокола. В классическом случае, процедура верификации — это тест, отвечающий на вопрос о надежности протокола: он принимает правила протокола и возвращает логическое значение (один бит). Если нам интересно знать не только то, уязвим ли протокол, но и то, какие именно у него уязвимости, чтобы мочь их сразу же исправить, желательно, чтобы процедура верификации возвращала также и найденные ею атаки. В этом случае, если мы прервем процесс верификации после первой же найденной уязвимости, она может дать неверное представление о слабых местах протокола.

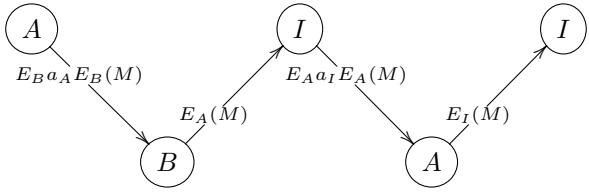
Пример 7. Обратимся к протоколу $P_{Double}[x_1, x_2]$, описанному в [5].
$P_{Double}[x_1, x_2] = ((x_1, E_{x_2} a_{x_1} E_{x_2}), (x_2, E_{x_1} D_{x_2} d_{x_1} D_{x_2}))$.

$P_{Double}[x_1, x_2]$ ненадежен. Это можно показать на примере следующей атаки, в которой участвуют правомерные пользователи A и B и злоумышленник I.



Здесь злоумышленник перехватывает $E_B a_A E_B(M)$, приписывает к нему свое имя и шифрует опять ключом E_B. B считает пересланное сообщение продолжением некоего правомерного взаимодействия с I по тому же протоколу и отсылает ему $E_I a_A E_B(M)$. I расшифровывает сообщение ключом D_I, стирает a_A и повторяет свои предыдущие действия. После ответа B злоумышленник I получает $E_I(M)$ и читает M. Подстановки в слова протокола здесь следующие: $\alpha_1[A, B]$, $\alpha_1[I, B]$ и $\alpha_2[I, B]$.

Рассмотрим другую последовательность действий A, B и I.



И опять злоумышленник получает доступ к M. Но теперь после перехвата $E_A(M)$ I начинает протокольное взаимодействие с A. Так что подстановки в слова протокола здесь $\alpha_1[A, B]$, $\alpha_1[I, A]$ и $\alpha_2[I, A]$.

Обе эти атаки несут в себе информацию об уязвимостях. Вторая схема атаки — самая короткая, но включает в себя подстановку $\alpha_2[I, A]$, возможную, только если права B и A в данной сети одинаковы. Но если, к примеру, $P_{Double}[A, B]$ регламентирует поведение клиент-серверной системы, так что B может лишь отвечать, а A — лишь инициировать обмен сообщениями, то вторая атака становится невозможной. И хотя первая атака длиннее, она рассматривает A только в роли отправителя, а B — в роли отвечающего, а значит, применима в большем количестве случаев.

Пример 8. Нам уже известно, что протокол $P_3[x_1, x_2, x_3]$ Примера 2 ненадежен. Атака Примера 3 является кратчайшей атакой на $P_3[x_1, x_2, x_3]$. Она становится возможной потому, что первый участник протокола не передает второму явной информации о третьем участнике, и второй участник не имеет возможности отличить I от C. Если исправить протокол следующим образом:

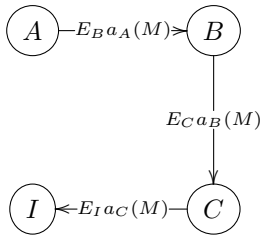
$$P'_3[x_1, x_2, x_3] = ((x_1, E_{x_2} a_{x_1} a_{x_3}), (x_2, E_{x_3} a_{x_2} d_{x_3} d_{x_1} D_{x_2}), (x_3, E_{x_1} a_{x_3} d_{x_2} D_{x_3})),$$

атака из Примера 3 исчезнет.

Однако $P'_3[x_1, x_2, x_3]$ также ненадежен, поскольку теперь второй участник протокола не передает информацию о первом участнике третьему, и третий

участник не знает, каким ключом шифровать сообщение, чтобы первый участник протокола мог его прочитать, а злоумышленник не мог.

Мы могли бы понять, что изменений, внесенных в $P_3'[x_1, x_2, x_3]$, чтобы превратить $P_3[x_1, x_2, x_3]$ в надежный протокол, недостаточно, даже не верифицируя сам $P_3'[x_1, x_2, x_3]$: существует следующая атака на $P_3[x_1, x_2, x_3]$:



Итак, чем больше различных схем атак находит алгоритм верификации, тем легче оказывается искать надежные аналоги исследуемого протокола.

Учитывая эти соображения, мы начнем с построения простого разрешающего алгоритма верификации протоколов, а затем посмотрим, будет ли он применим для получения более подробной информации об уязвимостях.

Таким образом, основные отличия предлагаемых нами в данной статье модели протокола и определения понятия угрозы на протокол от моделей, описанных в [7], проявляются в:

- **структуре словарей:** в словаре злоумышленника могут быть сложные слова;
- **выборе подстановок:** модель атаки допускает все подстановки, за исключением самоподстановок активного пользователя;
- **свойствах алгебры операторов:** допустимы неассоциативные действия; алгебра операторов не обязательно полусвободна;
- **типах атак:** атака не обязательно нацелена на получение исходного сообщения; она нацелена на получение одного из множества секретных слов INSEC, не все из которых могут быть суффиксами $\alpha_1[x_1, \ldots, x_n]$.

Кроме того, целью нашего алгоритма верификации является получение как можно более исчерпывающей информации о возможных атаках.

§ 2. Верификация пинг-понг протоколов с помощью конечных автоматов

В статье [5] предлагается алгоритм верификации пинг-понг протоколов с помощью конечных автоматов. Мы кратко опишем этот алгоритм для случая многосторонних протоколов со слабой моделью атаки (в исходной статье он применялся лишь к двусторонним протоколам) и обсудим, насколько он подходит для нашего расширения модели. Еще раз заметим, что мы используем суженное понятие слабой атаки; в случае сильных атак либо слабых атак без ограничений, алгоритм верификации [5] должен быть изменен.

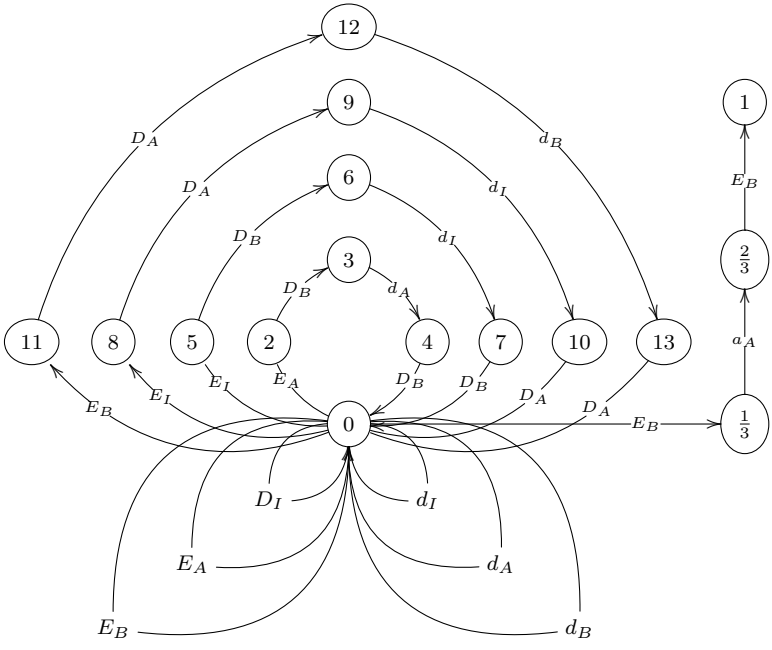
Роли правомерных участников $x_1, \ldots, x_n$ соотносятся с именами $U_1, \ldots, U_n$. Злоумышленник — пользователь I.

Пример 8 показывает, что если $\alpha_1[U_1, \ldots, U_n]$ не содержит операторов с индексом U_i, злоумышленник всегда может успешно выдать себя за пользователя U_i. Поэтому в дальнейшем мы считаем полноправными лишь тех пользователей, подстановки которых в исходное слово $\alpha_1[U_1, \ldots, U_n]$ непусты.

Алгоритм 2. Построим по n-стороннему протоколу $P[x_1, ..., x_n]$ автомат с конечным числом состояний.

1. Состояние 0 — единственное начальное состояние, 1 — единственное конечное. Входной алфавит $\Sigma = \Sigma_{U_1} \cup \cdots \cup \Sigma_{U_n} \cup \Sigma_I$.
2. Существует направленный путь из 0 в 1, ребра которого помечены операторами последовательности $\alpha_1[U_1, \ldots, U_n]$.
3. Для каждой буквы $\sigma \in \Sigma_I$ существует цикл из 0 в 0, помеченный σ.
4. Для каждой (не само-) подстановки $\alpha_i[U_{k_1}, \ldots, U_{k_n}]$, $\alpha_i[x_1, \ldots, x_n] \in P$, существует цикл из 0 в 0, ребра которого помечены операторами последовательности $\alpha_i[U_{k_1}, \ldots, U_{k_n}]$ в том же порядке, в котором они записываются в $\alpha_i[U_{k_1}, \ldots, U_{k_n}]$ (т. е. в порядке, обратном их фактическому применению).

Пример 9. Построим конечный автомат по протоколу $P_{Double}[A, B]$ Примера 7.



Нижние петли соответствуют операторам из словаря злоумышленника. Петли, соответствующей E_I, нет, поскольку никто, кроме I, не может применить D_I. Самый внутренний верхний цикл 0–2–3–4–0 соответствует подстановке $\alpha_2[A, B]$. Следующие два — подстановкам $\alpha_2[I, B]$ и $\alpha_2[I, A]$. Самый внешний верхний цикл — $\alpha_2[B, A]$. Подстановки $\alpha_2[A, I]$ и $\alpha_2[B, I]$ опущены — их можно заменить сочетанием операторов из Σ_I (корректность этого упрощения показана в [5]).

Теперь начинает работать собственно алгоритм верификации. Скажем, что путь p схлопывается, если соответствующее ему слово нормализуется в Λ. Множество всех схлопывающихся путей обозначим C. Верификация модели протокола в классическом смысле — это проверка, существует ли путь p', начинающийся в 0, завершающийся в 1 и принадлежащий C. Такая проверка осуществляется следующим образом [5].

Алгоритм 3.

1. Помещаем в C все пары (i,i). Создаем очередь Q, содержащую эти же пары в произвольном порядке.
2. Пока $Q \neq \emptyset$,
   (a) Удаляем из Q первую находящуюся в ней пару (i,j).
   (b) Если $(j,k) \in C$, $(i,k) \notin C$, помещаем (i,k) в C и в Q.
   (c) Если $(k,i) \in C$, $(k,j) \notin C$, помещаем (k,j) в C и в Q.
   (d) Если нашлось ребро $k \to i$, помеченное оператором τ, и ребро $j \to l$, помеченное σ, причем $\tau\sigma \to \Lambda$ и $(k,l) \notin C$, помещаем (k,l) в C и в Q.

Алгоритм совершает полиномиальное ($O(x^3)$) количество шагов от числа состояний автомата. Множество C, получающееся в результате его работы, содержит $(0,1)$ тогда и только тогда, когда существует схлопывающийся путь из 0 в 1.

Если попробовать перенести Алгоритм 3 на более широкий класс протоколов, условие «словарь злоумышленника может содержать составные операторы» требует лишь небольшого видоизменения действия (3) при построении автомата Алгоритмом 2. Выбор подстановок, как и говорилось, соответствует определенному нами ограниченному понятию слабой атаки. А другие два условия расширенной модели могут вызвать затруднения при переносе алгоритма. Рассмотрим их более подробно.

Алгебра: *Допустимы неассоциативные действия; алгебра операторов не обязательно полусвободна.*
Пусть алгебра операторов содержит какие-либо соотношения помимо $ab = \Lambda$. Исходный алгоритм верификации рассматривает лишь пары путей, новый же будет вынужден просматривать все цепочки путей вплоть до определенной длины, что скажется на времени его работы. Более того, в автоматной модели, построенной по Алгоритму 2, невыразимы такие операторы, которые соответствуют не действиям участников протокола, а некоторым внешним предписаниям, срабатывающим сразу же, как только возникает требуемое соотношение. Пусть, например, существует некоторая последовательность операторов σ_1, σ_2, σ_3, запускающая «триггер тревоги»: как только она появляется в слове, активный пользователь вынужден прервать сессию обмена сообщениями. Такие случаи с помощью Алгоритма 3 обработать не удастся. Эта трудность тесно связана с трудностями, возникающими при попытке применить алгоритм к неассоциативным данным. К примеру, пусть администратор располагает ключом U, умеющим расшифровывать как E_A, так и E_B, кроме того, $E_AU = E_BU = \Lambda$. Последовательность операторов $E_A(U(E_B(M)))$ схлопнется в $E_A(M)$, но не в $E_B(M)$. Однако это ограничение алгоритм не заметит.

Атаки: *Атака не обязательно нацелена на получение исходного сообщения; она нацелена на получение одного из множества секретных слов INSEC, не все из которых могут быть суффиксами слова* $\alpha_1[U_1, \ldots, U_n]$.
Алгоритм не хранит промежуточных данных о том, какие последовательности операторов можно прочитать, а помнит только пути, по которым можно прочитать пустое слово. Это позволяет уменьшить вычислительную сложность верификации. Но если INSEC $\neq \{\Lambda\}$, то отказаться от запоминания промежуточных слов уже невозможно.

Кроме прочего, Алгоритм 3 необходимо является чисто разрешающим и не дает подсказок насчет возможных атак. Сколько бы ни было схлопывающихся путей из 0 в 1, для автомата все они окажутся неразличимы.

Хотя классический алгоритм и не может использоваться в расширенной модели по причинам, указанным выше, его вычислительная сложность намного ниже, чем у предлагаемого в данной работе, и в классическом случае он будет работать лучше. То, что найденный способ верификации может быть применим к более широкому классу протоколов, отчасти оправдывает его большую вычислительную сложность (см. параграф 6).

§ 3. Построение моделей протоколов в префиксных грамматиках

В этом параграфе приводится понятие префиксной грамматики и затем — как с помощью этого понятия можно моделировать протоколы. Здесь также мы вводим понятие модели атаки и показываем, что в случае классических пинг-понг протоколов оно описывает то же множество атак, что и автоматная модель, описанная в параграфе 2.

Алфавит грамматики будем обозначать Υ. Буквы алфавита (грамматики) здесь и далее — строчные латинские буквы $a, b, c, \ldots, p, q, r$ и прописные $A, B, C, D, E, F, S, T, U$ (возможно с нижними и/или верхними индексами), переменные — x, y, z, w; а слова из Υ^* обозначаются прописными греческими буквами $\Gamma, \Delta, \Phi, \Psi, \Theta$.

Определение 7. Рассмотрим тройку $\langle \Upsilon, \mathbf{R}, \Gamma_0 \rangle$, где Υ — алфавит[5], $\Gamma_0, \Gamma_0 \in \Upsilon^*$ — начальное слово; $\mathbf{R} \subset \Upsilon^* \times \Upsilon^*$ — множество правил переписывания. Если эти правила переписывания применяются лишь к префиксам слов, иначе говоря, $R : \Phi \to \Psi$ применяется лишь к словам вида $\Phi\Theta$ и возвращает слово $\Psi\Theta$, тогда тройка $\langle \Sigma, \mathbf{R}, \Gamma_0 \rangle$ называется *префиксной грамматикой*.

Путем, порожденным префиксной грамматикой $\langle \Upsilon, \mathbf{R}, \Gamma_0 \rangle$, назовем последовательность $\{\Phi_i\}$ (конечную либо бесконечную) такую, что $\Phi_1 = \Gamma_0$ и

$$\forall i \exists R (R : R_l \to R_r \ \&\ R \in \mathbf{R} \ \&\ \Phi_i = R_l\Theta \ \&\ \Phi_{i+1} = R_r\Theta).$$

Если переписывающее правило применимо к любому слову и не меняет буквы префикса этого слова (лишь дописывает новые), такое правило пишем как $\Lambda \to \Phi$.

Пример 10. Рассмотрим $P_{3a}[x_1, x_2, x_3]$ — немного измененный трехсторонний протокол Примера 2.

[5] В отличие от словарей Σ_U параграфа 1, Υ состоит лишь из букв.

$\alpha_1[x_1, x_2, x_3] = E_{x_2} a_{x_3} a_{x_1}$
$\alpha_2[x_1, x_2, x_3] = E_{x_3} a_{x_1} d_{x_1} d_{x_3} D_{x_2}$
$\alpha_3[x_1, x_2, x_3] = E_{x_1} d_{x_1} D_{x_3}$
Множество уязвимости INSEC $= \{\Lambda\}$.

Промоделируем некоторые правила этого протокола (и поведения злоумышленника) префиксной грамматикой. В силу свойств искомого типа атак, можно обойтись только четырьмя пользователями канала связи. Пусть это будут A, B, C — легальные участники, и I — злоумышленник[6].

Выберем начальную подстановку — например, $[A, B, C]$ — и применим ее ко всем словам протокола. Тогда начальное слово будет выглядеть как $E_B a_C a_A$. Правила переписывания, соответствующие разрешенным по протоколу действиям:
$R^{[x_1,x_2,x_3]:1} : E_{x_2} a_{x_3} a_{x_1} \to E_{x_3} a_{x_1}$
$R^{[x_1,x_2,x_3]:2} : E_{x_3} a_{x_1} \to E_{x_1}$
Что подставлять в правило $R^{[x_1,x_2,x_3]:2}$, ясно сразу — это два любых различных элемента множества $\{A, B, C, I\}$. В $R^{[x_1,x_2,x_3]:1}$ же можно подставлять любые тройки из $\{A, B, C, I\}$, в которых второй элемент отличается от двух других (при этом первый и третий могут совпадать). В противном случае подстановка не будет слабо корректной.

Чтобы смоделировать поведение злоумышленника, нам также понадобятся правила, действующие с буквами из словаря Σ_I. Эти правила выглядят как $\Lambda \to x$ для $x \in \{E_I, D_I, a_I, d_I, a_A, d_A, a_B, d_B, a_C, d_C, E_A, E_B, E_C\}$ и как $x \to \Lambda$ для $x \in \{E_I, D_I, a_I, a_A, a_B, a_C\}$.

Можно ли использовать полученную модель грамматики для верификации $P_{3a}[x_1, x_2, x_3]$? Для того, чтобы это понять, необходимо разобраться, что такое модель атаки в префиксных грамматиках. Поскольку всякое действие пользователей протокола принимает в качестве входа слово и возвращает также слово, модели атак — это пути, порожденные префиксной грамматикой, содержащие слово из INSEC. Если INSEC $= \{\Lambda\}$, то интересные нам пути представляют собой последовательности действий над протоколом, позволяющие добиться получения исходного сообщения M злоумышленником.

Вообще говоря, если существует хотя бы одна такая последовательность, то их бесконечно много, но большинство из них неинтересны с точки зрения практики. Например, предположим, существует атака на $P_{3a}[x_1, x_2, x_3]$. Теперь добавим в нее n применений правила $\Lambda \to E_I$ к начальному слову $E_B a_C a_A$ и затем n применений правила $E_I \to \Lambda$ к получившемуся слову

$$\underbrace{E_I \dots E_I}_{n} E_B a_C a_A$$

В конце этих манипуляций мы получим опять $E_B a_C a_A$, а значит, еще одну модель атаки.

Однако обе атаки Примера 7 интересны с точки зрения практики, и было бы хорошо не терять их модели. Поэтому мы ввели понятие в некотором смысле

[6] В модели, подразумевающей сильную уязвимость, понадобилось бы ввести минимум семь злоумышленников.

минимальной атаки на протокол, не содержащей в себе подпоследовательности действий, также порождающей атаку.

Определение 8. *Модель атаки* на протокол **P** с множеством уязвимостей INSEC — это произвольный путь, порожденный префиксной грамматикой – моделью протокола **P**, завершающийся словом из INSEC.

Модель короткой атаки на протокол **P** с множеством уязвимостей INSEC — это путь, порожденный префиксной грамматикой – моделью протокола **P** и завершающийся словом Ф из INSEC такой, что никакая подпоследовательность[7] правил, применяемых при порождении этой атаки, сама не порождает атаку при применении ее к Γ_0.

Имея Определение 8, можно ответить на вопрос, корректен ли алгоритм построения модели протокола, представленный в Примере 10. Следующее рассуждение показывает, что это не так.

Пример 11. Пусть $T_{\langle x,y\rangle}$ — секретный ключ, известный лишь x и y, $U_{\langle x,y\rangle}$ — универсальный алгоритм расшифровки, доступный лишь x, $T_{\langle x,y\rangle}U_{\langle x,y\rangle} = \Lambda$ и $U_{\langle x,y\rangle}T_{\langle x,y\rangle} = \Lambda$, $E_xU_{\langle x,y\rangle} = \Lambda$, $U_{\langle x,y\rangle}E_x = \Lambda$.

Рассмотрим протокол $P_{SEC}[x_1, x_2]$ со следующими правилами:

$\alpha_1[x_1, x_2] = E_{x_2}T_{\langle x_2,x_1\rangle}$
$\alpha_2[x_1, x_2] = E_{x_1}a_{x_2}U_{\langle x_2,x_1\rangle}U_{\langle x_2,x_1\rangle}$
$\alpha_3[x_1, x_2] = T_{\langle x_2,x_1\rangle}d_{x_2}D_{x_1}$
INSEC $= \{\Lambda\}$.

Неформально протокол можно описать следующим образом. x_1 желает отправить некоторую засекреченную информацию M базе данных x_2 на удаленном сервере (база данных располагает алгоритмом $U_{\langle x_2,x_1\rangle}$) и посылает M, зашифровав ее публичным ключом E_{x_2} и приватным ключом $T_{\langle x_2,x_1\rangle}$. x_2 дважды применяет универсальный алгоритм расшифровки, сохраняет M и возвращает его x_1, подписав своим именем и зашифровав открытым ключом x_1. После чего от x_1 автоматически приходит уведомление об успешном завершении обмена информацией.

Попробуем построить префиксную грамматику, соответствующую правомерному взаимодействию участников протокола, и добавить в нее модель злоумышленника. Легальных участников протокола обозначим A и B, злоумышленника I.

$\Gamma_0 = E_BT_{\langle B,A\rangle}$.

Правила, напрямую соответствующие подстановкам в слова протокола:

$R^{[1]} : E_BT_{\langle B,A\rangle} \to E_Aa_B$
$R^{[2]} : E_Aa_B \to T_{\langle B,A\rangle}$
$R^{[3]} : E_AT_{\langle A,B\rangle} \to E_Ba_A$
$R^{[4]} : E_Ba_A \to T_{\langle A,B\rangle}$
$R^{[I1]} : E_BT_{\langle B,I\rangle} \to E_Ia_B$

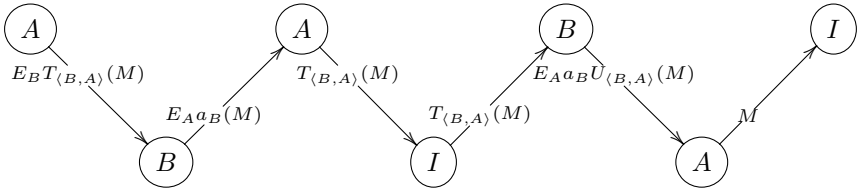
[7]Именно подпоследовательность, а не непрерывный отрезок последовательности: иначе короткими будут считаться атаки, разорванные в нескольких местах действиями с нулевым общим результатом.

$R^{[I2]} : E_A a_I \to T_{\langle I,A\rangle}$
$R^{[I3]} : E_A T_{\langle A,I\rangle} \to E_I a_A$
$R^{[I4]} : E_B a_I \to T_{\langle I,B\rangle}$

Правила, соответствующие поведению злоумышленника:

$R^{[LIA_x]} : \Lambda \to x$,
$x \in \{E_w, a_w, T_{\langle I,A\rangle}, T_{\langle A,I\rangle}, T_{\langle I,B\rangle}, T_{\langle B,I\rangle}, D_I, d_w, U_{\langle I,B\rangle}, U_{\langle I,A\rangle}\}, w \in \{A, B, I\}$
$R^{[LID_x]} : x \to \Lambda$,
$x \in \{E_I, a_A, a_B, a_I, T_{\langle I,A\rangle}, T_{\langle I,B\rangle}, D_I, U_{\langle I,B\rangle}, U_{\langle I,A\rangle}\}$

Эта грамматика не порождает слова Λ. Но протокол оказывается уязвим:



Злоумышленник перехватывает $T_{\langle B,A\rangle}(M)$ и пересылает базе данных. Та применяет к $T_{\langle B,A\rangle}(M)$ правило протокола $\alpha_2[A, B]$, сохраняет $U_{\langle B,A\rangle}(M)$ и пересылает подтверждение A, после чего от A приходит автоматическое уведомление о получении подтверждения, состоящее из исходного M.

Теперь опишем, как строить корректные модели протоколов в префиксных грамматиках. Оговорим, что словарь модельной префиксной грамматики составляют все сужения параметризованных операторов протокола на множество Usr.

Определение 9. Скажем, что префиксная грамматика $\mathbf{G}$ — *корректная модель протокола* $P[x_1, \ldots, x_n]$, если $\mathbf{G}$ порождает то же множество последовательностей операторов (слов), приведенных в нормальную форму, что и конечный автомат, построенный с помощью Алгоритма 3, работа которого начинается в состоянии 0 и заканчивается в состоянии 1.

Заметим, что в случае протоколов с полусвободной алгеброй операторов, мы можем немедленно свести задачу верификации к проблеме существования слов из INSEC в языке, порожденном корректной префиксной моделью протокола. Если такая грамматика порождает слово из INSEC, то оно порождается и автоматом, и наоборот. При переходе к более широкому классу алгебр так напрямую действовать уже нельзя, по причинам, указанным в параграфе 2. Однако если запретить промежуточные сокращения при порождении слов конечным автоматом, а оставить их на самый конец и совершать их согласно алгоритму Определения 2, то и в этом случае можно пользоваться Определением 9.

3.1. Моделирование действий протокола правилами префиксной грамматики. Прежде чем перейти к построению корректных префиксных моделей, обратим внимание, что применение последовательности операторов $c_1 c_2 c_3 \ldots c_n$ из $\mathrm{INST}(P, \mathrm{Usr}) \cup \Sigma_I$ можно смоделировать применением правила

$\Lambda \to c_1c_2c_3\dots c_n$ с учетом всех возможных стираний, вызываемых применением этой последовательности операторов (напомним, что $c_1c_2c_3\dots c_n$ приведен в нормальную форму).

Попробуем промоделировать действие стирающих правил префиксной грамматикой. Пусть последовательность операторов $c_1c_2c_3\dots c_n$ применяется к некоторой последовательности операторов β. При этом каждый из c_i имеет правый обратный: $c_ic_i^{-1} = \Lambda$. После применения $c_1c_2c_3\dots c_n$ к β может произойти от 0 до n сокращений, в зависимости от того, каким префиксом β начинается. Поэтому приписывание слова $c_1c_2c_3\dots c_n$ к слову — модели последовательности β в префиксной грамматике может осуществляться одним из следующих правил:

$$
\begin{array}{lcl}
R^{[1]} : \Lambda & \to & c_1c_2\dots c_n \\
R^{[2]} : c_n^{-1} & \to & c_1c_2\dots c_{n-1} \\
\dots & & \dots \\
R^{[n-1]} : c_n^{-1}\dots c_2^{-1} & \to & c_1 \\
R^{[n]} : c_n^{-1}c_{n-1}^{-1}\dots c_1^{-1} & \to & \Lambda
\end{array}
$$

Какое именно правило лучше всего применить в каждом конкретном случае — зависит от вида β. Если $\beta = c_n^{-1}c_{n-1}^{-1}\dots c_2^{-1}$, то из всего набора правил $R^{[1]} - R^{[n]}$, которые могут быть применены в этом случае, наиболее адекватно правило $R^{[n-1]}$: после его применения получится нормализованное слово c_1; правила же $R^{[1]} - R^{[n-2]}$ породят ненормализованные слова $c_1c_2\dots c_nc_n^{-1}\dots c_2^{-1}$, $\dots$, $c_1c_2c_2^{-1}$ соответственно.

Если оператор c_i не имеет правого обратного c_i^{-1} такого, что $c_ic_i^{-1} = \Lambda$, множество правил грамматики, соответствующих применению последовательности операторов $c_1c_2c_3\dots c_n$, можно сократить до $\{R^{[1]}, \dots, R^{[i-1]}\}$. Если некоторое подслово $c_1c_2\dots c_n$, скажем $c_i\dots c_{i+j}$, таково, что $c_{i+j}^{-1}\dots c_i^{-1} = \Lambda$, множество правил переписывания, которые могут соответствовать применению этой строки операторов, сокращается до $\{R^{[1]}, \dots, R^{[i+j-1]}\}$.

Хотя вышеуказанные правила переписывания порождают все нормализованные слова, порождаемые соответствующим автоматом, они могут порождать также слова, не находящиеся в нормальной форме. Поэтому в случае многосторонних пинг-понг протоколов описанный алгоритм впрямую не применим. Более того, если не выполнено допущение полусвободы, наличие ненормализованных слов в пути может вовсе привести к тому, что модель станет некорректной.

Пример 12. Пусть даны правила переписывания:

$$
\begin{array}{lcl@{\qquad}lcl}
R^{[1]} : \Lambda & \to & U_{\langle A,B\rangle} & R^{[6]} : \Lambda & \to & E_A \\
R^{[2]} : E_A & \to & \Lambda & R^{[7]} : D_A & \to & \Lambda \\
R^{[3]} : \Lambda & \to & D_AE_B & R^{[8]} : \Lambda & \to & D_A \\
R^{[4]} : U_{\langle A,B\rangle} & \to & D_A & R^{[9]} : E_A & \to & \Lambda \\
R^{[5]} : U_{\langle A,B\rangle}E_A & \to & \Lambda & & &
\end{array}
$$

Правила переписывания $R^{[1]} - R^{[2]}$ моделируют применение оператора $U_{\langle A,B\rangle}$ к последовательности операторов с учетом стирающего правила $U_{\langle A,B\rangle}E_A = \Lambda$; $R^{[3]} - R^{[5]}$ моделируют применение D_AE_B с учетом стирающих правил

$E_B U_{\langle A,B\rangle} = \Lambda$, $D_A E_A = \Lambda$. Правила $R^{[6]} - R^{[7]}$ моделируют применение E_A к строке операторов, с учетом стирающего правила $E_A D_A = \Lambda$, и $R^{[8]} - R^{[9]}$ моделируют применение D_A.

Пусть дан оператор $E_A E_A$, и к нему применяется $U_{\langle A,B\rangle}$, затем $D_A E_B$, а затем E_A и дважды D_A.

Применяя к $E_A E_A$ правила переписывания, указанные выше, в определенном порядке, имеем

$$E_A E_A \xrightarrow{R^{[1]}} U_{\langle A,B\rangle} E_A E_A \xrightarrow{R^{[4]}} D_A E_A E_A \xrightarrow{R^{[7]}} E_A E_A \xrightarrow{R^{[9]}} E_A \xrightarrow{R^{[9]}} \Lambda.$$

Если применить эти правила к E_A в другом порядке, получим

$$E_A E_A \xrightarrow{R^{[2]}} E_A \xrightarrow{R^{[3]}} D_A E_B E_A \xrightarrow{R^{[7]}} E_B E_A \xrightarrow{R^{[8]}} D_A E_B E_A \xrightarrow{R^{[8]}} D_A D_A E_B E_A.$$

И строка Λ, и строка $D_A D_A E_B E_A$ имеют нормальную форму. Но только одна из них — вторая — является результатом описанного процесса применения операторов к $E_A E_A$.

В случаях, подобных описанным в Примере 12, ошибочных атак можно избежать несколькими способами, часть из которых применяется уже в программной модели. Но кое-что можно улучшить и на уровне построения (недетерминированной) грамматики. Если алгебра параметризованных операторов содержит операторы без левого обратного (например, оператор d_x), причем эти операторы не входят в INSEC, то в любой модели короткой атаки применения таких операторов будут соответствовать сокращениям. Пример: если $\alpha_k = a_1 d_2 D_3$ и в INSEC нет слов с инфиксом d_2, то в модели короткой атаки α_k может появиться лишь в применении к словам с префиксом $E_3 a_2$.

После этого в случае неассоциативных данных всё равно могут появиться недопустимые пути – с ними мы разберемся уже при построении программной модели.

3.2. Некоторые способы сокращения размера модели протокола в префиксной грамматике. Если учесть указанное наблюдение, можно существенно уменьшить количество правил в модели, и не только потому, что будут удалены правила, порождающие слова с несокращенными парами букв, но и потому, что после удаления таких правил некоторые буквы могут оказаться отсутствующими в правых частях оставшихся правил (и в Γ_0). Если нет способа их породить, то нет смысла рассматривать правила их стирания. Обратно, если некоторой буквы теперь не встречается ни в INSEC, ни в каких левых частях правил, то правила, порождающие ее, нет смысла применять — после этого модели атаки никогда не получится. Этот процесс исключения избыточных правил может повториться несколько раз.

Пример 13. Пусть имеется строка операторов $E_{x_1} d_{x_1} D_{x_2}$, которую необходимо смоделировать в грамматике, описывающей протокол Примера 1, с начальным словом $\alpha_1[A,B] = E_B a_A$.
Напомним, что протокол выглядит следующим образом

$$P_2[x_1, x_2] = ((x_1, E_{x_2} a_{x_1}), (x_2, E_{x_1} a_{x_2} d_{x_1} D_{x_2})).$$

Пользователь I — злоумышленник. Множество уязвимостей $\mathrm{INSEC} = \{\Lambda\}$.

Применение последовательности операторов $E_{x_1} d_{x_1} D_{x_2}$, соответствующей второму действию по протоколу $P_2[x_1, x_2]$, к какой-либо другой последовательности операторов, так, чтобы в дальнейшем это применение привело к атаке, может проходить лишь двумя возможными способами: с сокращением фрагмента $d_{x_1} D_{x_2}$ или с сокращением всех операторов $E_{x_1} d_{x_1} D_{x_2}$. Запишем этот факт в виде правил грамматики; кроме того, запишем правилами грамматики действия злоумышленника, связанные с применением им операторов из Σ_I.

$$\begin{array}{lcl@{\qquad}lcl}
R^{[1]x_2:x_1} : E_{x_1} a_{x_2} & \to & E_{x_2} & & & \\
R^{[2]x_2:x_1} : E_{x_1} a_{x_2} D_{x_2} & \to & \Lambda & & & \\
R^{[3]x} : \Lambda & \to & E_x & R^{[5]x} : \Lambda & \to & a_x \\
R^{[4]x} : D_x & \to & \Lambda & R^{[6]x} : \Lambda & \to & d_x \\
R^{[7]x} : a_x & \to & \Lambda & R^{[8]} : E_I & \to & \Lambda \\
R^{[9]} : \Lambda & \to & D_I & & &
\end{array}$$

Видно, что нет никакой возможности породить D_A или D_B, поэтому правила переписывания $R^{[2]A:x}$, $R^{[2]B:x}$, $R^{[4]A}$, $R^{[5]A}$ излишни. Нет никакой возможности стереть d_x после порождения, поэтому излишни правила $R^{[6]x}$.

Также иногда можно обращаться к способу сокращения модели, используемому при применении Алгоритма 3 в Примере 9. А именно, если наша задача – лишь проверить протокол на надежность, а не найти все типы атак на него, можно избавиться от правил переписывания вида $\Lambda \to \mathrm{INST}(P, \mathrm{Usr})$, где $\mathrm{INST}(P, \mathrm{Usr}) \in \Sigma_I^*$ (и тех правил, которые получаются при применении строки операторов $\mathrm{INST}(P, \mathrm{Usr}) \in \Sigma_I^*$ с учетом возможных сокращений). Это преобразование не изменяет множества слов, порождаемых грамматикой, но может изменить множество путей, завершающихся словом из INSEC, так что часть коротких атак окажется потеряна.

Покажем, как наше наблюдение оптимизирует модель для $P_{3a}[x_1, x_2, x_3]$ из Примера 10.

Пример 14. Начальная неоптимизированная грамматика $P_{3a}[x_1, x_2, x_3]$:
$\Gamma_0 = E_B a_C a_A$

Правила $R^{[1-i]}$ соответстсвуют приписываниям слов, являющихся подстановками в $\alpha_2[x_1, x_2, x_3]$ (с учетом запрета на самоподстановки активного пользователя, таких подстановок может быть 36). Правила $R^{[2-i]}$ – $R^{[4-i]}$ есть приписывания этих подстановок с учетом возможных сокращений. Поскольку у a_x нет правых обратных, таких сокращений может быть, самое большее, три.

Правила $R^{[5-i]}$ соответствуют приписываниям подстановок в $\alpha_3[x_1, x_2, x_3]$. Это операторное слово не зависит от переменной x_2, поэтому количество слабо корректных подстановок в него равно 12. Правила $R^{[6-i]}$ – $R^{[8-i]}$ — применениям подстановок с учетом возможных сокращений. Правила $R^{[9-i]}$ и $R^{[12-i]}$ соответствуют применениям операторов E_x злоумышленником. Правила $R^{[10-i]}$ — применениям злоумышленником операторов a_x, и $R^{[11-i]}$ и $R^{[13-i]}$ — применениям d_x. $R^{[14]}$ и $R^{[15]}$ соответствуют применению злоумышленником D_I.

$R^{[1-1]} : \Lambda \to E_C a_A d_A d_C D_B$
$R^{[2-1]} : E_B \to E_C a_A d_A d_C$
$R^{[3-1]} : E_B a_C \to E_C a_A d_A$
$R^{[4-1]} : E_B a_C a_A \to E_C a_A$
$R^{[5-1]} : \Lambda \to E_A d_A D_C$
$R^{[6-1]} : E_C \to E_A d_A$
$R^{[7-1]} : E_C a_A \to E_A$
$R^{[8-1]} : E_C a_A D_A \to \Lambda$
$R^{[9-1]} : \Lambda \to E_A$
$R^{[10-1]} : \Lambda \to a_A$
$R^{[11-1]} : \Lambda \to d_A$
$R^{[12-1]} : D_A \to \Lambda$
$R^{[13-1]} : a_A \to \Lambda$
$R^{[14]} : \Lambda \to D_I$

$R^{[1-2]} : \Lambda \to E_B a_C d_C d_B D_A$
$R^{[2-2]} : E_A \to E_B a_C d_C d_B$
$R^{[3-2]} : E_A a_B \to E_B a_C d_C$
$R^{[4-2]} : E_A a_B a_C \to E_B a_C$
$R^{[5-2]} : \Lambda \to E_C d_C D_B$
$R^{[6-2]} : E_B \to E_C d_C$
$R^{[7-2]} : E_B a_C \to E_C$
$R^{[8-2]} : E_B a_C D_C \to \Lambda$
$R^{[9-2]} : \Lambda \to E_B$
$R^{[10-2]} : \Lambda \to a_B$
$R^{[11-2]} : \Lambda \to d_B$
$R^{[12-2]} : D_B \to \Lambda$
$R^{[13-2]} : a_B \to \Lambda$
$R^{[15]} : E_I \to \Lambda$

$\dots$
$\dots$
$\dots$
$\dots$
$\dots$
$\dots$
$\dots$
$\dots$
$\dots$
$\dots$
$\dots$
$\dots$
$\dots$

$R^{[1-36]} : \Lambda \to E_I a_I d_I d_I D_C$
$R^{[2-36]} : E_C \to E_I a_I d_I d_I$
$R^{[3-36]} : E_C a_I \to E_I a_I d_I$
$R^{[4-36]} : E_C a_I a_I \to E_I a_I$
$R^{[5-12]} : \Lambda \to E_I d_I D_A$
$R^{[6-12]} : E_A \to E_I d_I$
$R^{[7-12]} : E_A a_I \to E_I$
$R^{[8-12]} : E_A a_I D_I \to \Lambda$
$R^{[9-4]} : \Lambda \to E_I$
$R^{[10-4]} : \Lambda \to a_I$
$R^{[11-4]} : \Lambda \to d_I$
$R^{[12-4]} : D_I \to \Lambda$
$R^{[13-4]} : a_I \to \Lambda$

Учтем отстутствие левых обратных к d_x — исчезают правила $R^{[1-1]} - R^{[1-36]}$, $R^{[2-1]} - R^{[2-36]}$, $R^{[3-1]} - R^{[3-36]}$, $R^{[5-1]} - R^{[5-12]}$, $R^{[6-1]} - R^{[6-12]}$ и $R^{[11-1]} - R^{[11-4]}$. Все правила блоков $R^{[4-1]} - R^{[4-36]}$, $R^{[7-1]} - R^{[7-12]}$, $R^{[8-1]} - R^{[8-12]}$ с E_I в левой части либо D_I в правой части также избыточны[8], поскольку являются композициями правил блоков $R^{[9]}$–$R^{[13]}$ и $R^{[14]}$–$R^{[15]}$.

Для наглядности сократим грамматику еще больше, чтобы избавиться от правил, симметричных друг другу[9]. Пусть участники A, B, C не равноправны: если A может лишь посылать первое сообщение, а B и C не могут этого делать, то модель для $P_{3a}[x_1, x_2, x_3]$ становится такой:

$\Gamma_0 = E_B a_C a_A$

$R^{[1a]} : E_B a_C a_A \to E_C a_A$
$R^{[1d]} : E_B a_A a_I \to E_A a_I$
$R^{[1g]} : E_C a_B a_A \to E_B a_A$
$R^{[1j]} : E_C a_A a_I \to E_A a_I$
$R^{[1m]} : E_B a_A a_A \to E_A a_A$
$R^{[1p]} : E_C a_A a_A \to E_A a_A$

$R^{[1b]} : E_B a_A a_C \to E_A a_C$
$R^{[1e]} : E_B a_C a_I \to E_C a_I$
$R^{[1h]} : E_C a_A a_B \to E_A a_B$
$R^{[1k]} : E_C a_I a_B \to E_I a_B$
$R^{[1n]} : E_B a_C a_C \to E_C a_C$
$R^{[1q]} : E_C a_B a_B \to E_B a_B$

$R^{[1c]} : E_B a_I a_A \to E_I a_A$
$R^{[1f]} : E_B a_I a_C \to E_I a_C$
$R^{[1i]} : E_C a_I a_A \to E_I a_A$
$R^{[1l]} : E_C a_B a_I \to E_B a_I$
$R^{[1o]} : E_B a_I a_I \to E_I a_I$
$R^{[1r]} : E_C a_I a_I \to E_I a_I$

$R^{[2a]} : E_C a_A D_A \to \Lambda$
$R^{[2d]} : E_B a_A D_A \to \Lambda$

$R^{[2b]} : E_C a_I D_I \to \Lambda$
$R^{[2e]} : E_B a_C D_C \to \Lambda$

$R^{[2c]} : E_C a_B D_B \to \Lambda$
$R^{[2f]} : E_B a_I D_I \to \Lambda$

$R^{[3a]} : E_C a_A \to E_A$
$R^{[3d]} : E_B a_A \to E_A$

$R^{[3b]} : E_C a_I \to E_I$
$R^{[3e]} : E_B a_C \to E_C$

$R^{[3c]} : E_C a_B \to E_B$
$R^{[3f]} : E_B a_I \to E_I$

$R^{[4a]} : \Lambda \to E_A$
$R^{[4d]} : \Lambda \to E_I$

$R^{[4b]} : \Lambda \to E_B$

$R^{[4c]} : \Lambda \to E_C$

$R^{[5a]} : \Lambda \to a_A$
$R^{[5d]} : \Lambda \to a_I$

$R^{[5b]} : \Lambda \to a_B$

$R^{[5c]} : \Lambda \to a_C$

$R^{[6a]} : D_A \to \Lambda$
$R^{[6d]} : D_I \to \Lambda$

$R^{[6b]} : D_B \to \Lambda$

$R^{[6c]} : D_C \to \Lambda$

$R^{[7a]} : a_A \to \Lambda$
$R^{[7d]} : a_I \to \Lambda$

$R^{[7b]} : a_B \to \Lambda$

$R^{[7c]} : a_C \to \Lambda$

$R^{[8]} : \Lambda \to D_I$

$R^{[9]} : E_I \to \Lambda$

Здесь блок $R^{[1i]}$ соответствует блоку $R^{[4-i]}$ начальной грамматики, блок $R^{[2i]}$ — блоку $R^{[8-i]}$, блок $R^{[3i]}$ соответствует блоку $R^{[7-i]}$. Блоки $R^{[4i]}$ и $R^{[6i]}$ — это

[8] С точки зрения проверки модели на уязвимости, а не поиска самих уязвимостей! По смыслу эти действия соответствуют правомерным актам использования $P_{3a}[x_1, x_2, x_3]$ злоумышленником, и хотя в данном случае исключение соответствующих им правил из модели не влияет на результат анализа протокола, в более сложных случаях это может оказаться не так.

[9] То, что с такими преобразованиями также следует быть осторожными, показано в Примере 7.

прежние $R^{[9-i]}$ и $R^{[12-i]}$ соответственно. Блок $R^{[5i]}$ — прежний $R^{[1-i]}$, $R^{[7-i]}$ — прежний $R^{[13-i]}$, правила относительно применения D_I тоже переносятся без изменений.

§ 4. Верификация моделей протоколов и проблема слов для префиксных грамматик

Для верификации модели **G** достаточно выяснить, принадлежит ли хотя бы одно слово INSEC языку, порожденному **G**.

Если INSEC конечно, таким же будет и множество моделей коротких атак (это следствие Предложения 2). Мы хотим получить все это множество. Алгоритм, который мы применяем, является частью процесса суперкомпиляции [1] и основан на следующих наблюдениях.

Рассмотрим алгоритм, разворачивающий все возможные пути, порожденные **G**. Он является полуразрешающим для задачи поиска атак на модель протокола. Инструменты преобразования программ (например, суперкомпиляция), обращающиеся к развертке дерева вычислений, используют для ее завершения некоторые заранее определенные условия. Для нашей задачи они не обязательно подойдут, поскольку не приспособлены к работе именно с префиксными грамматиками, а значит, могут обрывать некоторые ветви дерева путей, порождаемых префиксной грамматикой, слишком рано. В таком случае придется видоизменять условия обрыва ветви так, чтобы они подошли для разрешения проблемы существовавния слова в языке, порождаемом префиксной грамматикой. Чтобы понять, насколько это необходимо, рассмотрим один из наиболее широко используемых в преобразовании программ условий обрыва ветви — гомеоморфное вложение термов [18]. Здесь мы интересуемся этим вложением лишь на строковых типах данных — для них оно совпадает с отношением подпоследовательности.

Определение 10. Пусть даны два слова $\Phi = a_1 a_2 \dots a_m$, $\Psi = b_1 b_2 \dots b_n$. Φ *гомеоморфно вкладывается* в Ψ(пишем $\Phi \trianglelefteq \Psi$), если последовательность букв Φ есть подпоследовательность последовательности букв Ψ.

Проверим, работает ли это отношение при разрешении проблемы слов для префиксных грамматик.

Пример 15. Рассмотрим протокол P_{RR} Примера 7 с асимметричными ролями (где A может только отправлять сообщения, B может только отвечать). Префиксная грамматика для такого сужения P_{RR}:

$R^{[1a]} : E_A a_B E_A \to E_B$ $R^{[1b]} : E_B a_A E_B \to E_A$ $R^{[1c]} : E_A a_I E_A \to E_I$
$R^{[1d]} : E_B a_I E_B \to E_I$

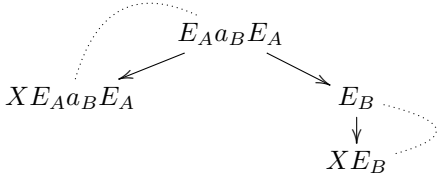
$R^{[2a]} : E_A a_B \to E_B D_A$ $R^{[2b]} : E_B a_A \to E_A D_B$ $R^{[2c]} : E_A a_I \to E_I D_A$
$R^{[2d]} : E_B a_I \to E_I D_B$

$R^{[3a]} : E_A a_B E_A D_B \to \Lambda$ $R^{[3b]} : E_B a_A E_B D_A \to \Lambda$ $R^{[3c]} : E_A a_I E_A D_I \to \Lambda$
$R^{[3d]} : E_B a_I E_B D_I \to \Lambda$

$R^{[4a]} : \Lambda \to E_A$ $R^{[4b]} : \Lambda \to E_B$ $R^{[4c]} : \Lambda \to E_I$
$R^{[5a]} : \Lambda \to a_A$ $R^{[5b]} : \Lambda \to a_B$ $R^{[5c]} : \Lambda \to a_I$
$R^{[6a]} : D_A \to \Lambda$ $R^{[6b]} : D_B \to \Lambda$ $R^{[6c]} : D_I \to \Lambda$
$R^{[7a]} : a_A \to \Lambda$ $R^{[7b]} : a_B \to \Lambda$ $R^{[7c]} : a_I \to \Lambda$
$R^{[8]} : \Lambda \to D_I$ $R^{[9]} : E_I \to \Lambda$

Здесь правила блока $R^{[1i]}$ соответствуют применению подстановок в α_2, подразумеваемому протоколом (см. диаграммы взаимодействий из Примера 7). Правила блока $R^{[2i]}$ — это тоже применения подстановок в α_2, но к словам, имеющим вид не $E_x a_y E_x \Phi$, а $E_x a_y \Phi$, так что сокращения второго оператора D_x не происходит[10]. Правила блока $R^{[3i]}$ — все то же применение подстановок в α_2, но уже в случае, когда оно сокращает «больше, чем надо» операторов (это происходит на словах с префиксом $E_x a_y E_x D_y$). Все прочие правила соответствуют применению злоумышленником операторов из своего словаря.

Пусть начальное слово $\Gamma_0 = E_A a_B E_A$. Начнем развертку всех путей, порожденных этой префиксной грамматикой, пока в пути не найдутся два слова Γ и Δ таких, что $\Gamma \trianglelefteq \Delta$:



$X \in \{E_A, E_B, E_I, a_A, a_B, a_I, D_I\}$.

Пунктирные дуги показывают вложения по $\trianglelefteq$ слов в вершинах–предках дерева в слова в вершинах–потомках.

Последовательности, содержащей Λ, в развертке не находится. Хотя мы знаем, что она есть.

Пример 15 показывает, что отношение подпоследовательности действительно не работает как достаточное условие обрыва ветви при решении проблемы существования слова в языке префиксной грамматики. Внесем в него изменения так, чтобы получилось корректное достаточное условие.

Пусть $\mathbf{G} = \langle \Upsilon, \mathbf{R}, \Gamma_0 \rangle$ — префиксная грамматика. Множество только левых частей R_l правил $R_l \to R_r \in \mathbf{R}$ обозначим $\mathbf{R_l}$. Аналогично, множество правых частей R_r правил $R_l \to R_r \in \mathbf{R}$ обозначим $\mathbf{R_r}$.

Определение 11. Префиксная грамматика $\mathbf{G} = \langle \Upsilon, \mathbf{R}, \Gamma_0 \rangle$ называется *обогащенной*, если всякие два правила $R_l^{[1]} \to R_r^{[1]}$ и $R_l^{[2]} \to R_r^{[2]}$ либо имеют

[10] Протоколом такие случаи применения операторов из α_2 не предусмотрены, но если в ход передачи данных вмешивается нарушитель — скажем, применяет к начальному слову $E_A a_B E_A$ оператор a_I, а затем E_B, — они становятся возможными.

одинаковые правые части ($R_r^{[1]} = R_r^{[2]}$), либо не имеют в них общих букв ($\forall i, j(R_r^{[1]}[i] \neq R_r^{[2]}[j])$).

Определение 12. Дана грамматика $\mathbf{G} = \langle \Upsilon, \mathbf{R}, \Gamma_0 \rangle$ с конечным числом правил. Скажем, что слово $\Gamma \in \Upsilon^+$ *избыточно*, если для какой-нибудь буквы $a \in \Upsilon$ число вхождений a в Γ больше, чем число вхождений a во все слова $\mathbf{R_l}$. Назовем это совокупное число вхождений в левые части[11] *пределом стирания* a (и обозначим его как $\mathrm{EL}(a)$).

Пример 16. Рассмотрим $\mathbf{G_{2EXP}} = \langle \{a, b, c, A, B, C, e\}, \mathbf{R_{2EXP}}, e \rangle$ со следующими правилами переписывания $\mathbf{R_{2EXP}}$:

$R^{[1]} : e \to aA$ $R^{[5]} : AA \to \Lambda$ $R^{[9]} : Ba \to bB$
$R^{[2]} : \Lambda \to aA$ $R^{[6]} : BB \to \Lambda$ $R^{[10]} : Cb \to cC$
$R^{[3]} : \Lambda \to bB$ $R^{[7]} : CC \to \Lambda$
$R^{[4]} : \Lambda \to cC$ $R^{[8]} : c \to \Lambda$

$\mathbf{G_{2EXP}}$ является обогащенной. $\mathrm{EL}(e) = 1$; также $\mathrm{EL}(a) = \mathrm{EL}(b) = \mathrm{EL}(c) = 1$, поэтому слова $aAaA$ и $cCcC$ избыточны. Слово $cCCC$ не избыточно, в силу $\mathrm{EL}(C) = 3$.

Предложение 1. Пусть $\mathbf{G} = \langle \Upsilon, \mathbf{R}, \Gamma_0 \rangle$ — обогащенная префиксная грамматика с конечным числом правил. Всякий бесконечный путь, порожденный $\mathbf{G}$, содержит либо пару равных слов, либо избыточное слово.

Доказательство. Во всякой бесконечной последовательности слов их длины либо не превышают некоторого N, либо неограниченно растут. В первом случае найдутся два одинаковых слова; во втором рано или поздно найдется слово Γ, длина которого окажется не меньше величины $|\Upsilon| * M + 1$, где M — наибольшее число вхождений буквы из Γ в какое-либо слово из $\mathbf{R_l}$. В слове Γ число вхождений какой-нибудь буквы обязательно превысит ее предел стирания. □

Предложение 2. Пусть $\mathbf{G} = \langle \Upsilon, \mathbf{R}, \Gamma_0 \rangle$ — обогащенная префиксная грамматика с конечным числом правил. Ни одна короткая модель атаки, порожденная $\mathbf{G}$, не содержит пар слов Γ и Δ таких, что $\Gamma = \Delta$ или одно из них избыточно.

Доказательство. Случай двух равных слов очевиден.

Пусть модель короткой атаки содержит избыточное слово Γ. Тогда найдется a — буква, которая должна быть стерта дважды правилами с одной и той же левой частью. Рассмотрим слова, в которых производятся эти стирания. Они имеют вид $\hat{R}_l a \Psi a \Theta_0$ и $\hat{R}_l a \Theta_0$, где $\hat{R}_l$ — префикс левой части некоторых двух правил переписывания[12] $R^{[1]} : \hat{R}_l a \overline{R_l} \to R_r^{[1]}$, $R^{[2]} : \hat{R}_l a \overline{R_l} \to R_r^{[2]}$, $R^{[1]} \in \mathbf{R}$, $R^{[2]} \in \mathbf{R}$. Рассмотрим, в каких словах порождались указанные вхождения букв a. Они имеют вид $\hat{R}'_r a \Theta_0$ и $\hat{R}'_r a \Psi a \Theta_0$ соответственно (см. рисунок ниже), причем $\hat{R}'_r$ — префикс правой части некоторых двух правил $R^{[3]} : R_l^{[3]} \to \hat{R}'_r a \overline{R'_r}$, $R^{[4]} : R_l^{[4]} \to \hat{R}'_r a \overline{R'_r}$, $R^{[3]} \in \mathbf{R}$, $R^{[4]} \in \mathbf{R}$.

[11] Поскольку $\mathbf{R_l}$ — множество, то каждая левая часть, в сколько бы правил она ни входила, входит в $\mathbf{R_l}$ лишь однажды.

[12] Эти правила могут и совпадать.

Все эти слова совместно входят в следующую последовательность. σ_i здесь — ряд правил, применяемый на выделенном отрезке пути.

$$\left.\begin{array}{r}\Gamma_0\\ \ldots\\ \hat{R}'_r a\Theta_0\end{array}\right\}\sigma_1$$
$$\left.\begin{array}{r}\ldots\end{array}\right\}\sigma_2$$
$$\left.\begin{array}{r}\hat{R}'_r a\Psi a\Theta_0\\ \ldots\\ \hat{R}_l a\Psi a\Theta_0\end{array}\right\}\sigma_3$$
$$\left.\begin{array}{r}\ldots\end{array}\right\}\sigma_4$$
$$\left.\begin{array}{r}\hat{R}_l a\Theta_0\\ \ldots\\ \Lambda\end{array}\right\}\sigma_5$$

Известно, что на отрезке последовательности, где применялись правила σ_3, суффикс $a\Psi a\Theta_0$ неизменен[13].

Теперь применим последовательность правил σ_1 к слову Γ_0, получим $\hat{R}'_r a\Theta_0$, затем применим к нему ряд правил σ_3, получим $\hat{R}_l a\Theta_0$, применим к нему цепочку σ_5 и получим Λ. Мы построили модель атаки, являющуюся подпоследовательностью исходной атаки, значит, исходная атака не принадлежит множеству коротких. □

Предложение 2 влечет множество полезных следствий.

Первое из них касается решения задачи верификации моделей протоколов с множеством уязвимости, содержащим что-то помимо пустого слова. Рассмотрим конечное множество слов INSEC и модель протокола $\mathbf{G}$. Добавим к пределу стирания буквы алфавита (обозначим a) максимальное число ее вхождений в слово из INSEC. Полученную величину обозначим $\mathrm{EL}_{\mathrm{INSEC}}(a)$. Слово, содержащее некоторую букву большее число раз, чем ее $\mathrm{EL}_{\mathrm{INSEC}}$, назовем *избыточным относительно INSEC*.

Проиллюстрируем введенное определение. Пусть грамматика $\mathbf{G_{2EXP}}$ рассматривается вместе с множеством $\mathrm{INSEC} = \{\Lambda, aa, abc\}$. Тогда $\mathrm{EL}_{\mathrm{INSEC}}(a) = \mathrm{EL}(a) + 2 = 3$, поскольку наибольшее число вхождений буквы a в слово из INSEC равно 2.

Следствие 1. Никакая модель короткой атаки на INSEC не содержит избыточных относительно INSEC слов.

Следствие 2. Если INSEC конечно, то количество моделей коротких атак конечно.

Следствие 3. Если обогащенная грамматика $\mathbf{G} = \langle \Upsilon, \mathbf{R}, \Gamma_0 \rangle$ такова, что все ее правила переписывания имеют левую часть длины 1 (такие грамматики также называются алфавитными), и $\mathrm{INSEC} = \{\Lambda\}$, то ни одна модель короткой

[13] Ни одна его буква не была стерта и порождена заново.

атаки, порожденная $\mathbf{G}$, не содержит слов Γ и Δ таких, что Γ предшествует Δ и $\Gamma \trianglelefteq \Delta$. В этом случае для решения задачи верификации достаточно использовать условие обрыва ветви по гомеоморфному вложению.

Доказательство. Пользуемся теоремой об отношении Турчина [2; Теорема 5]: первая пара Γ и Δ таких, что Γ предшествует Δ и $\Gamma \trianglelefteq \Delta$ в пути, порожденном обогащенной префиксной грамматикой, имеет вид $\Gamma = R_r\Theta_0$, $\Delta = R_r\Psi\Theta_0$, $R_r \in \mathbf{R_r}$, причем на всем отрезке пути от Γ до Δ суффикс Θ_0 неизменен.

Если Ψ содержит хотя бы одну букву из R_r, то Δ избыточно. В противном случае путь от начального слова до Λ имеет следующий вид (σ_i здесь — ряд правил, применяемый на выделенном отрезке пути; a — буква, совпадающая с последней буквой слова R_r):

$$\left.\begin{matrix}\Gamma_0\\ \dots\\ a\Theta_0\end{matrix}\right\}\sigma_1$$
$$\left.\begin{matrix}\dots\end{matrix}\right\}\sigma_2$$
$$\left.\begin{matrix}a\Psi\Theta_0\\ \dots\\ \Psi\Theta_0\end{matrix}\right\}\sigma_3$$
$$\left.\begin{matrix}\dots\end{matrix}\right\}\sigma_4$$
$$\left.\begin{matrix}\Theta_0\\ \dots\\ \Lambda\end{matrix}\right\}\sigma_5$$

Применяем σ_1 к Γ_0, затем к получившемуся слову $a\Theta_0$ применяем σ_3, получаем Θ_0 и из него уже Λ. Описанная модель атаки короче исходной. □

Что более важно, Предложение 2 совместно с Предложением 1 представляет собой достаточное условие обрыва пути в задаче поиска всех коротких моделей атак. Чтобы Предложение 2 оказалось применимым, одни и те же буквы, порожденные различными правилами, следует считать различными. В Примере 14, однако, буква a_C исходного слова Γ_0 совпадает с буквой a_C, порожденной $R^{[5c]}$, так что представленная грамматика не является обогащенной. Обогащение грамматики (и представление этих букв как различных) произведено в соответствующей программной модели и описано в параграфе 5.

Опираясь на определение модели короткой атаки, можно предложить еще один способ уменьшить размер модельной грамматики для протокола. А именно, можно выбросить из грамматики правила, соответствующие шифрованию ключом злоумышленника с секретным алгоритмом расшифровки, поскольку никто, кроме злоумышленника, расшифровать этот ключ не может. Теперь этот способ уменьшения размера модели может получить более строгое обоснование.

Предложение 3. Пусть $\mathbf{G}$ — модельная грамматика некоторого протокола, INSEC — множество его уязвимостей. Ни одна модель короткой атаки на

INSEC, порожденная $\mathbf{G}$, не порождается с помощью правил грамматики вида $\Lambda \to a$, которые имеют следующие свойства:

1. правило переписывания, содержащее a в левой части, в грамматике $\mathbf{G}$ единственно, и оно имеет вид $a \to \Lambda$;
2. a не встречается ни в одном слове из INSEC.

Доказательство. Действительно, пусть правило указанного вида применяется при порождении модели короткой атаки. Скажем, $\Lambda \to a$ применяется к Θ_0 и получается слово $a\Theta_0$. Поскольку рассматриваемый путь является моделью атаки, рано или поздно буква a должна исчезнуть из слов этого пути — она не встречается в словах из INSEC. Единственный способ ее стереть — применить правило $a \to \Lambda$ к слову $a\Theta_0$. Значит, в пути необходимо должно найтись второе вхождение слова Θ_0, которого не может быть в модели короткой атаки. □

§ 5. Построение программы по модельной префиксной грамматике

5.1. Описание модельного языка. В этом параграфе программы написаны на следующем простом функциональном языке.

В качестве конструкторов в языке допускаются конструктор кортежа фиксированного размера, для краткости записываемый как `(_,...,_)`, и конструктор списка `cons`, первым аргументом которого является голова списка, а вторым — хвост.

Имена определяемых функций в языке — это строки, начинающиеся с латинской буквы, не совпадающие со строкой `cons`. Считаем, что имена переменных в языке всегда начинаются с буквы `x`; все остальные имена, отличные от имен определяемых функций, являются идентификаторами констант. Язык также допускает в качестве типов данных и натуральные числа.

Определение функции `f(x1,...,xn)` есть последовательность предложений вида `f(T1,...,Tn)=T0;` либо `f(T1,...,Tn)=g(S1,...,Sm);`.

`Ti` может быть константой (нульместным конструктором), переменной, кортежем, содержащим константы и переменные; либо выражением вида `cons(Qi, Tj)`, где `Qi` — константа, переменная либо пара из констант либо переменных; `Si` — выражения, в которых могут быть функциональные вызовы, но не может быть переменных, которых нет в `Ti`.

Примеры ошибочного кода на модельном языке:

1. `f(x1,f(x2))=x1;` (вызов функции в образце);
2. `f(cons(cons(3,1), Nil))=0;` (наличие в образце вложенного списка);
3. `cons((A,1),xx)=x;` (недопустимое имя функции и неопределенная переменная в правой части).

При вызове функции `f(x1,...,xn)` предложения ее определения просматриваются сверху вниз, и первое же предложение, сопоставление с левой частью которого вызова `f(x1,...,xn)` по всем аргументам успешно, выполняется (как в языках Рефал, Haskell или Prolog).

Пример 17. Пусть определение функции `f(x1,x2)` выглядит так:

```
f(cons(x1,x2), x3)      = f(x2,x1);
f(cons((A,x1),x2), xx) = A;
f(Nil, x)               = x;
```

Вызов этой функции `f(x, Nil)` возвращает последний элемент списка `x`. При этом второе предложение определения функции, хотя синтаксически построено верно, никогда не используется: если первый аргумент `f` успешно сопоставляется с `cons((A,x1),x2)`, то он успешно сопоставляется и с `cons(x1,x2)`.

5.2. Построение программной модели по грамматике. Программная модель протокола строится по грамматике $\mathbf{G} = \langle \Upsilon, \mathbf{R}, \Gamma_0 \rangle$, моделирующей протокол, следующим образом.

Напомним, что $\mathbf{R_r} = \{\Phi | \exists R \in \mathbf{R} | R : R_l \to \Phi\}$. Каждая правая часть[14] $R_r^{[i]} \in \mathbf{R_r}$ получает уникальный идентификатор i. Всякую букву c, входящую в $R_r^{[i]}$, представим парой $\langle c, i \rangle$. Буквы, входящие в начальное слово, оснащаются идентификатором 0.

Обозначим $\Phi[last]$ последнюю букву слова Φ. Предложение программы, соответствующее применению правила $R_l^{[i]} \to R_r^{[i]}$, есть

$$\texttt{f}(\texttt{cons}((R_l^{[i]}[1], x_1), \texttt{cons}((R_l^{[i]}[2], x_2), \dots \texttt{cons}((R_l^{[i]}[last], x_{[last]}), \Psi) \dots), \texttt{cons}(i, \texttt{xHist}), EL_1) =$$
$$\texttt{f}(\texttt{cons}((R_r^{[i]}[1], x_1), \texttt{cons}((R_r^{[i]}[2], x_2), \dots \texttt{cons}((R_r^{[i]}[last], x_{[last]}), \Psi) \dots), \texttt{xHist}, EL_2);$$

Первый аргумент функции `f` — это слово, порождаемое грамматикой $\mathbf{G}$, имеющее форму списка, каждая буква которого оснащена идентификатором породившего ее правила. Второй аргумент `f`, `xHist` — история порождения пути, записанная как список номеров применяемых в нем правил грамматики $\mathbf{G}$. Мы добавили в аргументы историю вычислений, чтобы сделать программную модель детерминированной — этот прием также использовался при верификации протоколов когеренции кэша в [13].

Третий аргумент $\texttt{f}(\Phi, \texttt{xHist}, \text{EL})$, EL — список количеств вхождений всех букв, принадлежащих $\Upsilon \cup (\mathbf{R_r} \cap \{\Gamma_0\})$, в первый аргумент Φ. Этот аргумент добавлен, чтобы был возможен обрыв ветви вычислений согласно Предложению 2. Если некоторый счетчик в списке EL достигает значения предела стирания соответствующей буквы, то ветвь вычисления, на которой это произошло, обрывается независимо от значения `xHist`.

Пример 18. Построим модельную префиксную грамматику и соответствующую программную модель для протокола Примера 1

$$P_2[x_1, x_2] = ((x_1, E_{x_2} a_{x_1}), (x_2, E_{x_1} a_{x_2} d_{x_1} D_{x_2})).$$

Модельная префиксная грамматика $\mathbf{G_2}$ может быть записана следующим образом.

$\Gamma_0 = E_B a_A$.

[14] Заметим, что в силу того, что $\mathbf{R_r}$ — множество, при оснащении правых частей правил идентификаторами левые части правил различать не обязательно. То есть, если в грамматике имеется два правила вида $R_l^{[i]} \to R_r$ и $R_l^{[j]} \to R_r$, их (совпадающие) правые части могут получить равные идентификаторы.

$R^{[1]} : E_B a_A \to E_A a_B$ $R^{[2]} : E_A a_B \to E_B a_A$ $R^{[3]} : E_B a_I \to E_I a_B$ $R^{[4]} : E_A a_I \to E_I a_A$

$R^{[5]} : \Lambda \to E_A$ $R^{[6]} : \Lambda \to E_B$
$R^{[7]} : \Lambda \to a_A$ $R^{[8]} : \Lambda \to a_B$ $R^{[9]} : \Lambda \to a_I$

$R^{[10]} : a_A \to \Lambda$ $R^{[11]} : a_B \to \Lambda$ $R^{[12]} : a_I \to \Lambda$ $R^{[13]} : E_I \to \Lambda$

Пределы стирания: $\mathrm{EL}(E_A) = \mathrm{EL}(E_B) = \mathrm{EL}(a_A) = \mathrm{EL}(a_B) = 2$; $\mathrm{EL}(E_I) = 1$; $\mathrm{EL}(a_I) = 3$.

Перепишем эту грамматику с учетом оснащения букв в правых частях номерами соответствующих правил. Буквы x_i соответствуют произвольным числам.

$R^{[1]} : (E_B, x_1)(a_A, x_2) \to (E_A, 1)(a_B, 1)$ $R^{[2]} : (E_A, x_1)(a_B, x_2) \to (E_B, 2)(a_A, 2)$
$R^{[3]} : (E_B, x_1)(a_I, x_2) \to (E_I, 3)(a_B, 3)$ $R^{[4]} : (E_A, x_1)(a_I, x_2) \to (E_I, 4)(a_A, 4)$

$R^{[5]} : \Lambda \to (E_A, 5)$ $R^{[6]} : \Lambda \to (E_B, 6)$
$R^{[7]} : \Lambda \to (a_A, 7)$ $R^{[8]} : \Lambda \to (a_B, 8)$
$R^{[9]} : \Lambda \to (a_I, 9)$

$R^{[10]} : (a_A, x) \to \Lambda$ $R^{[11]} : (a_B, x) \to \Lambda$
$R^{[12]} : (a_I, x) \to \Lambda$ $R^{[13]} : (E_I, x) \to \Lambda$

Теперь мы готовы строить программу. `xL_occ` — количество вхождений буквы L в первый аргумент функции `f`. Константа `a` обозначает E_A, `b` обозначает E_B, `i` обозначает E_I; `A` обозначает a_A, `B` обозначает a_B и `I` обозначает a_I.

Пример 19. Программная модель для $P_2[A, B]$

```
f(Nil, xHist, (xa_occ, xb_occ, xi_occ, xA_occ, xB_occ, xI_occ))   = An attack found;

f(xWord, xHist, (3, xb_occ, xi_occ, xA_occ, xB_occ, xI_occ))      = No short attack;
f(xWord, xHist, (xa_occ, 3, xi_occ, xA_occ, xB_occ, xI_occ))      = No short attack;
f(xWord, xHist, (xa_occ, xb_occ, xi_occ, 3, xB_occ, xI_occ))      = No short attack;
f(xWord, xHist, (xa_occ, xb_occ, xi_occ, xA_occ, 3, xI_occ))      = No short attack;
f(xWord, xHist, (xa_occ, xb_occ, 2, xA_occ, xB_occ, xI_occ))      = No short attack;
f(xWord, xHist, (xa_occ, xb_occ, xi_occ, xA_occ, xB_occ, 4))      = No short attack;

f(cons((b, x1),cons((A,x2), xWord))),
     cons(R1, xHist), (xa_occ, xb_occ, xi_occ, xA_occ, xB_occ, xI_occ))
             = f(cons((a,1),cons(B,1), xWord)),
                     xHist, (xa_occ+1, xb_occ-1, xi_occ, xA_occ-1, xB_occ+1, xI_occ));

f(cons((a,x1), cons((B,x2), xWord))),
     cons(R2, xHist), (xa_occ, xb_occ, xi_occ, xA_occ, xB_occ, xI_occ))
             = f(cons((b,2),cons((A,2), xWord)),
                     xHist, (xa_occ-1, xb_occ+1, xi_occ, xA_occ+1, xB_occ-1, xI_occ));

f(cons((b,x1), cons((I,x2), xWord))),
     cons(R3, xHist), (xa_occ, xb_occ, xi_occ, xA_occ, xB_occ, xI_occ))
             = f(cons(((i,3),cons((B,3), xWord)),
                     xHist, (xa_occ, xb_occ-1, xi_occ+1, xA_occ, xB_occ+1, xI_occ-1));

f(cons((a,x1), cons((I,x2), xWord))),
     cons(R4, xHist), (xa_occ, xb_occ, xi_occ, xA_occ, xB_occ, xI_occ))
             = f(cons((i,4),cons((A,4), xWord)),
                     xHist, (xa_occ-1, xb_occ, xi_occ+1, xA_occ+1, xB_occ, xI_occ-1));

f(xWord, cons(R5, xHist), (xa_occ, xb_occ, xi_occ, xA_occ, xB_occ, xI_occ))
             = f(cons((a,5), xWord),
```

```
                    xHist, (xa_occ+1, xb_occ, xi_occ, xA_occ, xB_occ, xI_occ));

f(xWord, cons(R6, xHist), (xa_occ, xb_occ, xi_occ, xA_occ, xB_occ, xI_occ))
            = f(cons((b,6), xWord),
                    xHist, (xa_occ, xb_occ+1, xi_occ, xA_occ, xB_occ, xI_occ));

f(xWord, cons(R7, xHist), (xa_occ, xb_occ, xi_occ, xA_occ, xB_occ, xI_occ))
            = f(cons((A,7), xWord),
                    xHist, (xa_occ, xb_occ, xi_occ, xA_occ+1, xB_occ, xI_occ));

f(xWord, cons(R8, xHist), (xa_occ, xb_occ, xi_occ, xA_occ, xB_occ, xI_occ))
            = f(cons(B,8), xWord),
                    xHist, (xa_occ, xb_occ, xi_occ, xA_occ, xB_occ+1, xI_occ));

f(xWord, cons(R9, xHist), (xa_occ, xb_occ, xi_occ, xA_occ, xB_occ, xI_occ))
            = f (cons(I,9), xWord),
                    xHist, (xa_occ, xb_occ, xi_occ, xA_occ, xB_occ, xI_occ+1));

f(cons((A,x1), xWord),
     cons(R10, xHist), (xa_occ, xb_occ, xi_occ, xA_occ, xB_occ, xI_occ))
            = f (xWord, xHist, (xa_occ, xb_occ, xi_occ, xA_occ-1, xB_occ, xI_occ));

f(cons((B,x1), xWord),
     cons(R11, xHist), xa_occ, xb_occ, xi_occ, xA_occ, xB_occ, xI_occ))
            = f (xWord, xHist, (xa_occ, xb_occ, xi_occ, xA_occ, xB_occ-1, xI_occ));

f(cons((I,x1),xWord),
     cons(R12, xHist), xa_occ, xb_occ, xi_occ, xA_occ, xB_occ, xI_occ))
            = f (xWord, xHist, (xa_occ, xb_occ, xi_occ, xA_occ, xB_occ, xI_occ-1));

f(cons((i,x1),xWord),
     cons(R13, xHist), xa_occ, xb_occ, xi_occ, xA_occ, xB_occ, xI_occ))
            = f(xWord, xHist, (xa_occ, xb_occ, xi_occ-1, xA_occ, xB_occ, xI_occ));
```

Обратим внимание, что отдельные счетчики стирания на буквы, порождаемые различными правилами, в Примере 19 не заводятся. Так, количество вхождений букв `b` в первый аргумент `f` хранится в единственной переменной `xb_occ`, и по ее значению невозможно понять, буквы это из пары `(b,2)` или `(b,6)`.

Чтобы выяснить, порождает ли программная модель Λ, решим задачу суперкомпиляции полученной программы на параметризованной входной точке `f (cons((B,0), cons((a,0),Nil))), xHist, (1,0,0,0,1,0))`.

Суперкомпилятор начнет процесс развертки дерева вычислений по параметру `xHist`. Если в построенном им дереве возможных вычислений окажется лист, содержащий строку `An attack found`, то протокол ненадежен; в противном случае на модель протокола, скорее всего, атак не существует, но чтобы узнать наверняка, нам следует переписать программу, введя различные счетчики вхождений для букв, образующих пары с разными числами (например, для `(b,2)` — счетчик `xb2_occ`, а для `(b,6)` — счетчик `xb6_occ`). При этом длина кортежа в третьем аргументе `f` станет равной сумме длин всех правых частей правил исходной грамматики. Для букв с идентификатором 0 (букв начального слова) счетчики можно не заводить: известно, что их количество в слове пути может только уменьшиться.

Программа, полученная суперкомпиляцией примера с различными счетчиками стирания, так же, как и программа, полученная суперкомпиляцией модельной программы для $P_{3a}[x_1, x_2, x_3]$ с различными счетчиками стирания, не порождают моделей атак.

5.3. Некоторые свойства программных моделей грамматик. Укажем на особенность программной модели Примера 18, обеспечивающую, в том числе, корректность построения таких моделей для протоколов с неассоциативными операторами.

Как уже было сказано, предложения программы просматриваются интерпретатором сверху вниз, и для исполнения им выбирается первое же подходящее к конфигурации предложение. В модели Примера 19 это существенно, главным образом, для сокращения размера кода, поскольку соответствующая грамматика порождает лишь нормализованные слова. Для грамматики Примера 15 это уже неверно: правила из блоков $R^{[1i]}$, примененные вместо правил $R^{[3i]}$, породят пары несокращенных операторов. Для того, чтобы избежать этого, сопоставим правилам $R^{[1i]}$ и $R^{[3i]}$ один и тот же идентификатор и поместим в модельной программе предложение, соответствующее правилу $R^{[3i]}$, выше предложения, соответствующего правилу $R^{[1i]}$. Точно так же поступим с блоками правил $R^{[6i]}$ и $R^{[4i]}$ (первый приоритетнее) и правилами $R^{[9]}$ и $R^{[8]}$.

Пример ниже показывает, что отсутствие отдельных счетчиков для букв, порождаемых разными правилами, может привести к неудаче в поиске атаки.

Пример 20. Рассмотрим протокол с тремя участниками: пользователь **C**, администратор **A** и база данных **B**. Помимо стандартных операций, имеются:

1. операторы шифрования секретным именным ключом O_x;
2. операторы приписывания анонимного идентификатора пользователя i_x;
3. шифрование закрытым ключом администратора S_A;
4. простое приписывание 1 бита информации A.

К каждому этому оператору есть односторонние обратные: $N_x O_x = \Lambda$, $f_x i_x = \Lambda$, $U_x S_x = \Lambda$, $DA = \Lambda$.

$\Sigma_A = \{a_x, E_x, d_x, S_A, U_A, O_x, N_x, i_x, f_x, D_x, D\}$, $\Sigma_B = \Sigma_A$.

$\Sigma_C = \{a_x, E_x, D_C, d_x, f_I, A, D, N_C\}$,

Основное протокольное взаимодействие $P_{Vote}[x_1, x_2, x_3]$ выглядит так:

$\alpha_1[x_1, x_2, x_3] = (x_1, E_{x_2} a_{x_1})$
$\alpha_2[x_1, x_2, x_3] = (x_2, E_{x_1} S_{x_2} O_{x_1} a_{x_1} d_{x_1} D_{x_2})$
$\alpha_3[x_1, x_2, x_3] = (x_1, E_{x_2} A N_{x_1} D_{x_1})$
$\alpha_4[x_1, x_2, x_3] = (x_3, O_{x_1} i_{x_1} S_{x_2} R_{x_1} d_{x_1} N_{x_1} U_{x_2} O_{x_1} D D_{x_2})$
$\alpha_5[x_1, x_2, x_3] = (x_2, E_{x_1} i_{x_1} U_{x_2} f_{x_1} N_{x_1})$
$\mathrm{INSEC} = \{\Lambda\}$.

Этот протокол представляет собой некоторый процесс голосования. Пользователь, знающий N_{x_1}, высылает свой запрос о голосовании администратору, подтвердив его своим именем и той информацией, которую может знать только он. Администратор возвращает ему зашифрованный своим ключом именной

шифр голосующего (так, что узнать их x_1 пока не может). x_1 начинает голосование: расшифровывает сообщение своими открытым и секретным ключом и подписывает к нему свой голос A, после чего зашифровывает полученный токен открытым шифром администратора E_{x_2} и посылает базе данных. База данных проверяет, что токен действительно зашифрован секретным ключом администратора — это значит, что пользователь, который его отправил, тоже знает свой секретный ключ, — и размещает в открытом доступе анонимный идентификатор голосовавшего и его голос. После чего посылает анонимный идентификатор администратору под именным ключом, а администратор пересылает его пользователю, чтобы он мог проверить, правильно ли учтен его голос.

Возможное распределение ролей в протоколе с учетом словарей пользователей: $x_2 \in \{\mathbf{A}, \mathbf{B}\}$, $x_3 \in \{\mathbf{A}, \mathbf{B}\}$.

Также имеются два вспомогательных протокола с теми же ключами:

$P_{RemU}[x_1, x_2] = ((x_1, E_{x_2}E_{x_1}), (x_2, O_{x_1}i_{x_1}D_{x_1}D_{x_2}))$,
Здесь x_1 — произвольный пользователь, $x_2 = \mathbf{A}$ или $x_2 = \mathbf{B}$. Протокол осуществляет запрос полноправного пользователя x, располагающего ключом N_x, о напоминании анонимного идентификатора x.

$P_{RemA}[x_1, x_2, x_3] = ((x_1, O_{x_3}a_{x_3}), (x_2, O_{x_3}i_{x_3}f_{x_3}d_{x_3}))$.

Здесь x_1 и x_2 могут быть как $\mathbf{A}$, так и $\mathbf{B}$. Протокол осуществляет запрос $\mathbf{A}$ к $\mathbf{B}$ (или наоборот) о напоминании анонимного идентификатора i_x некоторого пользователя x.

Неполноправный пользователь $\mathbf{I}$ узнал пароль от открытого ключа $\mathbf{C}$ и хочет взломать секретный ключ администратора. $\Sigma_I = \{a_x, E_A, E_C, D_C, d_x, f_x, A, D\}$. Задача осложнена тем, что $\mathbf{I}$ не знает секретного ключа N_C.

$\mathbf{I}$ ждет, пока $\mathbf{C}$ проведет легальную сессию голосования с $\mathbf{A}$ и $\mathbf{B}$. Затем перехватывает некоторое сообщение $S_A(M)$, приписывает к нему a_C, шифрует результат с помощью E_A и пересылает все это $\mathbf{A}$ под видом запроса $\mathbf{C}$ о переголосовании. $\mathbf{A}$ пересылает ему $E_C S_A O_C a_C S_A(M)$. $\mathbf{I}$ шифрует это сообщение с помощью E_A и пересылает $\mathbf{A}$ в виде запроса о напоминании анонимного идентификатора $\mathbf{C}$ по протоколу $P_{RemU}[\mathbf{C}, \mathbf{A}]$. Получает $O_C i_C S_A O_C a_C S_A(M)$ и пересылает опять же $\mathbf{A}$ от имени $\mathbf{B}$ в качестве $\alpha_5[C, A, B]$ протокола $P_{Vote}[C, A, B]$. Полученный после этого $E_C i_C O_C a_C S_A(M)$ превращает в $O_C a_C S_A(M)$, затем пересылает его $\mathbf{B}$ от имени $\mathbf{A}$ по протоколу $P_{RemA}[\mathbf{A}, \mathbf{B}]$, получает $O_C i_C S_A(M)$, и под видом повторного запроса $\alpha_5[C, A, B]$ посылает его $\mathbf{A}$. Ответная реплика $E_C i_C(M)$ легко позволяет злоумышленнику получить начальное секретное сообщение M.

Если выписать эту атаку в виде пути, получим:
$\Gamma_0 = S_A(M)$
$\Gamma_1 = E_A a_C S_A(M)$
$\Gamma_2 = E_C S_A O_C a_C S_A(M)$
$\Gamma_3 = E_A E_C S_A O_C a_C S_A(M)$
$\Gamma_4 = O_C i_C S_A O_C a_C S_A(M)$
$\Gamma_5 = E_C i_C O_C a_C S_A(M)$

$\Gamma_6 = O_C a_C S_A(M)$
$\Gamma_7 = O_C i_C S_A(M)$
$\Gamma_8 = E_C i_C(M)$

В словах Γ_4 и Γ_7 одна и та же буква S_A (сама порожденная разными правилами) дважды стирается одним и тем же правилом $O_C i_C S_A \to E_C i_C$, и обрыв ветви с общими счетчиками стирания не позволит отыскать атаку.

Понять важность разделения счетчиков можно и без Примера 20. Можно заметить, что программа и грамматика Примера 18 могут быть построены в обратном порядке. Назначим начальным словом Λ (в программе — `Nil`), а множеством INSEC — $\{E_B a_A\}$ (в программе — `cons((b,x1), cons((A,x2),Nil))`). Правила вида $R_l \to R_r$ перепишем в правила $R_r \to R_l$ (похожий способ переворачивания моделей описан в [15]). Обратный путь из Λ в $E_B a_A$ также удовлетворяет Предложению 2. Во многих случаях развертка дерева возможных вычислений обратной программы работает быстрее, чем развертка для исходной программы. Причина в том, что словарь злоумышленника содержит больше операций приписывания (соответствующих правилам вида $\Lambda \to x$), чем стирающих операций (в обращенном виде соответствующих правилам вида $\Lambda \to x$), поэтому дерево возможных вычислений обратной программы имеет меньше ветвлений.

Пример обращенной модели для $P_{Double}[A, B]$ приведен в приложении [20]. Она имеет ту особенность, что, за исключением блока $R^{[2i]}$, соответствует алфавитной грамматике, поэтому счетчики стирания нужны лишь для букв, входящих в слово INSEC.

§ 6. О сложности процесса верификации

В этом параграфе оценим возможную длину атаки на модель протокола. Чтобы найти грубую верхнюю оценку на наибольшую длину атаки, можно воспользоваться идеями работы [17] и Предложением 2.

Предложение 4. Пусть $\mathbf{G} = \langle \Upsilon, \mathbf{R}, \Gamma_0 \rangle$ — произвольная обогащенная префиксная грамматика с конечным числом правил. Наибольшая длина модели короткой атаки ограничена сверху экспонентой от $card(\mathbf{R})$.

Доказательство. Возьмем произвольную букву b, принадлежащую слову из $\mathbf{R_r}$. Слово Γ, принадлежащее пути, являющемуся моделью короткой атаки на $\mathbf{G}$, содержит в себе не больше вхождений букв b, чем предел стирания $\mathrm{EL}(a)$ такой буквы a, что $\exists i, \Phi(\Phi \in \mathbf{R_r} \,\&\, b = \Phi[i] \,\&\, a = \Phi[last])$. Каждое такое a может присутствовать в слове этого пути лишь вместе с каким-нибудь суффиксом (возможно пустым) слова $\Phi \in \mathbf{R_r}$, последней буквой которого a является.

Посчитаем количество S_Γ всех возможных распределений всех возможных суффиксов слов из $\mathbf{R_r}$ в слове Γ, не содержащем букв слова Γ_0. Имеем

$$S_\Gamma \leqslant \prod_{\Phi \in \mathbf{R_r}} |\Phi|^{\mathrm{EL}(\Phi[last])}.$$

С учетом того, что начальное слово тоже порождает $|\Gamma_0|$ возможных своих суффиксов, количество S всех возможных распределений суффиксов в слове, входящем в короткую атаку, оценивается формулой $S \leqslant |\Gamma_0| * \prod\limits_{\Phi \in \mathbf{R_r}} |\Phi|^{\mathrm{EL}(\Phi[last])}$.

Если длина пути — модели атаки превысит величину S, то в нем найдутся либо два равных слова, либо избыточное слово. □

Может ли случиться так, что длина самой короткой модели атаки, порождаемой грамматикой, приблизилась к величине S? Попробуем найти такую модель атаки, пользуясь таким же построением «лестничных пролетов», как и в [17].

Предложение 5. Существует грамматика $\mathbf{G} = \langle \Upsilon, \mathbf{R}, \Gamma_0 \rangle$ такая, что длина самой короткой модели атаки, ей порождаемой, экспоненциальна от $card(\mathbf{R})$.

Доказательство. Рассмотрим семейство грамматик $\mathbf{G_{\langle n \rangle EXP}}$ с начальным словом $a_1 A_1$.

$\mathbf{G_{\langle n \rangle EXP}}$:

$R_1^W : \Lambda \to a_1 A_1$	$R_1^T : A_2 a_1 \to a_2 A_2$	$R_1^D : A_1 \to \Lambda$
.....	.....	.....
$R_i^W : \Lambda \to a_i A_i$	$R_i^T : A_{i+1} a_i \to a_{i+1} A_{i+1}$	$R_i^D : A_i \to \Lambda$
.....	.....	.....
$R_n^W : \Lambda \to a_n A_n$	$R_n^T : a_n \to \Lambda$	$R_n^D : A_n \to \Lambda$

Для грамматики $\mathbf{G_{\langle 1 \rangle EXP}}$ кратчайшая модель атаки — кратчайший путь, завершающийся Λ, — выглядит как цепочка $a_1 A_1 \xrightarrow{R_1^T} A_1 \xrightarrow{R_1^D} \Lambda$ и имеет длину 3.

Получим общую формулу длины кратчайшей модели атаки на $\mathbf{G_{\langle n \rangle EXP}}$ с помощью математической индукции. Обозначим длину кратчайшей модели атаки, порождаемой грамматикой $\mathbf{G_{\langle n \rangle EXP}}$, как $F(n)$. Теперь обратимся к грамматике $\mathbf{G_{\langle n+1 \rangle EXP}}$. Чтобы стереть a_1 из исходной $a_1 A_1$, необходимо получить слово $A_2 a_1 A_1$. A_2 в этом слове может порождаться либо правилом R_1^T, либо правилом R_2^W. В первом случае перед применением правила R_1^T получилось бы избыточное слово $a_1 A_2 a_1 A_1$; во втором случае рассматриваемый путь необходимо содержит слово $a_2 A_2 a_1 A_1$. Поскольку на отрезке пути от $a_2 A_2 a_1 A_1$ до $A_2 a_1 A_1$ правила R_1^W, R_1^T и R_1^D не используются (иначе на этом отрезке появилось бы избыточное слово, содержащее второе вхождение a_1), можно рассматривать этот отрезок как модель атаки на $\mathbf{G_{\langle n \rangle EXP}}$, в которой индексы всех букв увеличены на 1 и отсутствует самый последний шаг (стирание A_1 правилом R_1^D; в нашем отрезке — стирание A_2 правилом R_2^D). Длина этого отрезка, таким образом, не меньше, чем $F(n) - 1$.

К слову $A_2 a_1 A_1$ применим R_1^T и получим слово $a_2 A_2 A_1$. Повторяем атаку на $\mathbf{G_{\langle n \rangle EXP}}$ (с точностью до индексов букв), чтобы стереть $a_2 A_2$, и наконец стираем A_1 с помощью R_1^D. Общее число слов в построенной атаке не меньше, чем $2 * F(n) + 1$.

Так что длина самой короткой модели атаки на $\mathbf{G_{\langle n \rangle EXP}}$ есть $2^{n+1} - 1$. □

Может ли подобная грамматика соответствовать протоколу? Рассмотрим следующий аналог $\mathbf{G_{\langle n \rangle EXP}}$.

Пример 21. Пусть $\mathbf{G_{LA}} = \langle \{a, b, c, A, B, C\}, \mathbf{R_{LA}}, aA \rangle$, а множество правил переписывания $\mathbf{R_{LA}}$ выглядит следующим образом:

$$\begin{array}{llll} R^{[1]} : \Lambda \to aA & R^{[1a]} : A \to a & R^{[4]} : Ba \to bB & R^{[4a]} : BaB \to b \\ R^{[2]} : \Lambda \to bB & R^{[2a]} : B \to b & R^{[5]} : Cb \to cC & R^{[5a]} : CbC \to c \\ R^{[3]} : \Lambda \to cC & R^{[3a]} : C \to c & & R^{[6]} : c \to \Lambda \end{array}$$

$\mathbf{G_{LA}}$ очень похожа на грамматику $\mathbf{G_{2EXP}}$ Примера 16, но имеет то отличие, что правила $AA \to \Lambda$, $BB \to \Lambda$ и $CC \to \Lambda$ для нее являются не внутренними, а внешними, и применяются сразу же, как только это становится возможно после применений правил $R^{[1]}$–$R^{[5]}$. Из-за этого (а также из-за того, что начальное слово aA совпадает с правой частью правила $R^{[1]}$) $\mathbf{G_{LA}}$ формально не является обогащенной, но если присвоить правилам $R^{[ia]}$ и правилам $R^{[i]}$ один и тот же номер, после чего в программной модели первые всегда делать приоритетнее, получится корректная программная модель для обогащенной грамматики $\mathbf{G_{2EXP}}$.

Предположим, что на некотором сайте администраторы ввели собственную систему шифрования: прежде зашифровки обычным открытым ключом, она применяет к сообщению некоторый самообратный шифр перестановки, являющийся подтверждением того, что адресант является легальным пользователем сайта. Самообратный шифр, дважды подряд примененный, дает исходное сообщение.

Поставим в соответствие буквам A, B, C самообратные шифры перестановки $F_{\mathbf{A}}$, $F_{\mathbf{B}}$, $F_{\mathbf{C}}$, а строчным a, b, c — операции шифрования $E_{\mathbf{A}}$, $E_{\mathbf{B}}$, $E_{\mathbf{C}}$. По условию $F_x F_x = \Lambda$ — правило стирания для F_x. Чтобы послать администратору $\mathbf{B}$ запрос о смене приватных данных, пользователь $\mathbf{C}$ должен зашифровать его обычным ключом администратора $E_{\mathbf{B}}$, а затем собственным ключом перестановки $F_{\mathbf{C}}$ (что можно сделать, применив открытую систему шифрования сайта, выражающуюся композицией $E_{\mathbf{C}}F_{\mathbf{C}}$, а затем дешифровав собственный открытый ключ, тем самым подтверждая, что сообщение пересылается именно от того пользователя, который в нем заявлен). Администратор, зная, кому принадлежит какой ключ перестановки, расшифровывает $F_{\mathbf{C}}E_{\mathbf{B}}$ и возвращает пользователю его исходное сообщение, зашифрованное открытой системой шифрования сайта (то есть опять применяет композицию $E_{\mathbf{C}}F_{\mathbf{C}}$). Имея $D_{\mathbf{C}}$ и зная, что перестановка $F_{\mathbf{C}}$ самообратна, пользователь $\mathbf{C}$ может убедиться, что ему пришло то сообщение, которое он посылал.

Представим себе, что помимо администратора, на сайте есть еще и программист $\mathbf{A}$. С обычными пользователями он не общается, а с администратором взаимодействует по вышеописанной схеме, причем ключи друг друга им известны. В некоторый момент пользователю $\mathbf{C}$ удалось перехватить сообщение $E_{\mathbf{A}}F_{\mathbf{A}}F_{\mathbf{B}}(M)$, соответствующее реплике $\mathbf{B}$ на сообщение $\mathbf{A}$ вида $F_{\mathbf{A}}E_{\mathbf{B}}F_{\mathbf{B}}(M)$, в котором предположительно содержится база данных паролей. Сумеет ли $\mathbf{C}$ взломать это сообщение?

Ответ заключен в грамматике $\mathbf{G_{LA}}$ (с той лишь разницей, что начальное слово здесь aAB, а не aA). При отсутствии бдительности со стороны администрации $\mathbf{C}$ может узнать M, но для этого ему потребуется 34 операции!

1	aAB	$cCbBAB$	$cCAB$	BaB
2	$bBaAB$	$CbBAB$	CAB	b
3	$cCbBaAB$	$cCBAB$	cAB	cCb
4	$CbBaAB$	$CBAB$	AB	Cb
5	$cCBaAB$	$cBAB$	aB	cC
6	$CBaAB$	BAB	$bBaB$	C
7	$cBaAB$	bAB	$cCBaB$	c
8	$BaAB$	$cCbAB$	$CBaB$	Λ
9	$bBAB$	$CbAB$	$cBaB$	

Хотя этот пример и модельный, он указывает на ненадежность переборного поиска атак вплоть до некоторой (небольшой) длины.

§ 7. Обсуждение

Предложенный способ верификации протоколов неоптимален во многих отношениях. Во-первых, он имеет высокую вычислительную сложность по сравнению с автоматным алгоритмом Долева–Ивена–Карпа [5], хотя и возвращает больше информации о возможных атаках (если они существуют). Во-вторых, описанный нами алгоритм моделирования протокола префиксной грамматикой может быть улучшен почти во всех частных случаях.

Однако в нашей попытке верификации наиболее интересен не собственно алгоритм. Как уже говорилось, в последние годы инструменты преобразования программ общего назначения (вроде суперкомпиляторов) хорошо себя показали при решении различных задач верификации, таких как верификация протоколов когерентности кэша [13] или поиск атак на некоторые криптографические протоколы [4]. Эти задачи могут быть сложны (например, задача о миссионерах и каннибалах, разрешенная суперкомпиляцией в общем виде [14]) и нести практический смысл (например, как в работах [11] или [8]). Поэтому хотелось бы понять, насколько возможно распространить успех решения одной такой задачи на целый класс сходных.

На этот вопрос трудно ответить из-за того, что чаще всего предмет верификации теоретически изучен гораздо лучше, чем инструмент преобразования программ, ее осуществляющий. Поэтому может быть интересно сначала изучить возможности некоторого класса инструментов преобразования программ с точки зрения теории, а затем найти задачи, которые можно разрешить в рамках этих возможностей. Этот способ рассуждений оправдал себя в нашем случае: после изучения свойств гомеоморфного вложения на путях, порожденных префиксными грамматиками [17] стало возможно описать такие префиксные модели пинг-понг протоколов, про которые доказуем факт корректности их верификации суперкомпиляцией. Удалось не только построить формально верифицируемую модель пинг-понг протоколов, но и расширить класс протоколов, для которых описанный алгоритм работает корректно. Но, хотя мы доказали, что наш алгоритм формально корректен для пинг-понг протоколов и заметили, что он может быть применен в более широком контексте, все-таки не смогли установить точных границ его применимости.

Вторая интересная особенность нашего исследования касается условия обрыва ветви дерева возможных путей вычислений. Первая попытка найти такое условие в случае верификации протоколов, представленная на VPT [16], использовала видоизмененное отношение, предложенное В. Ф. Турчиным для обрыва ветви дерева вычислений при суперкомпиляции [19]. Это отношение использовало понятие времени, что приводило к дополнительным затратам памяти [1]. В нашей работе удалось избавиться от привязки к временным меткам, чтобы условие обрыва могло быть использовано теми инструментами преобразования программ, которые не используют отношение Турчина. Но в доказательствах Предложения 2 и Предложения 3 понятие времени опять неявно возникает, когда идет речь о неизменности суффикса слова в отрезке последовательности. В этих доказательствах мы работаем не только со словами, а с историями порождения этих слов, что перекликается с попыткой использовать историю в приложении к частичным вычислениям в логических программах [9].

Благодарности

Искренне благодарю редактора сборника Андрея Петровича Немытых за поддержку автора при работе над содержательной частью статьи и помощь в улучшении ее изложения.

Список литературы

[1] А. П. Немытых, *Суперкомпилятор SCP4: общая структура*, УРСС, М., 2007.

[2] А. Н. Непейвода, “Отношение Турчина и аппроксимация циклов при анализе программ”, *Сборник трудов по функциональному языку программирования Рефал*, **1** (2014), 170–192.

[3] M. Abadi, A. D. Gordon, “A bisimulation method for cryptographic protocols”, *Nordic Journal of Computing*, **5** (1998), 267–303.

[4] A. Ahmed, A. Lisitsa, A. Nemytykh, “Cryptographic Protocol Verification via Supercompilation (A Case Study)”, *Proceedings of VPT 2013, First International Workshop on Verification and Program Transformation*, EPiC Series, **16**, EasyChair, 2013, 16–29.

[5] D. Dolev, S. Even, R. M. Karp, “On the security of ping-pong protocols”, *Information and Control*, **55** (1982), 57–68.

[6] D. Dolev, A. C. Yao, “On the security of public key protocols”, *Transactions on Information Theory*, **29** (1983), 198–208.

[7] S. Even, O. Goldreich, “On the security of multi-party ping-pong protocols”, Technical Report, 1985.

[8] G. W. Hamilton, N. D. Jones, “Distillation with labelled transition systems”, *Proceedings of the ACM SIGPLAN 2012 workshop on Partial evaluation and program manipulation*, 2012, 15–24.

[9] J. Gallagher, L. Lafave, “Regular Approximation of Computation Paths in Logic and Functional Languages”, *Lecture Notes in Computer Science*, **1110**, 1996, 115–136.

[10] A. Klimov, “Solving Coverability Problem for Monotonic Counter Systems by Supercompilation”, *Lecture Notes in Computer Science*, **7162**, 2012, 193–209.

[11] I. Klyuchnikov,, “Nullness Analysis of Java Bytecode via Supercompilation over Abstract Values”, *Fourth International Valentin Turchin Workshop on Metacomputation*, 2014, 161–176.

[12] R. Needham, M. Schroeder, "Using encryption for authentication in large networks of computers", *Communications of ACM*, **21**:12 (1978), 993–999.

[13] A. Lisitsa, A. P. Nemytykh, "Reachability Analysis in Verification via Supercompilation", *International Journal of Foundations of Computer Science*, **19**:4 (2008), 953–970.

[14] A. Lisitsa, A. Nemytykh, "A Note on Program Specialization. What Syntactical Properties of Residual Programs Can Reveal?", *Proceedings of VPT 2014, Second International Workshop on Verification and Program Transformation*, EPiC Series, **16**, EasyChair, 2014, 52–65.

[15] A. P. Lisitsa, A. P. Nemytykh, "On one application of computations with oracle", *Programming and Computer Software*, **36**:3 (2010), 157–165.

[16] A. Nepeivoda, "Ping-Pong protocols as prefix grammars and Turchin's relation", *Proceedings of VPT 2013, First International Workshop on Verification and Program Transformation*, EPiC Series, **16**, EasyChair, 2013, 74–87.

[17] A. Nepeivoda, "Turchin's Relation and Subsequence Relation in Loop Approximation", *Proceedings of PSI 2014, Ershov Informatics Conference*, EPiC Series, **23**, EasyChair, 2014, 30–42.

[18] M. H. Sørensen, R. Glück, "An Algorithm of Generalization in Positive Supercompilation", *Proceedings of ILPS'95, the International Logic Programming Symposium*, 1995, 465–479.

[19] V. F. Turchin, "The algorithm of generalization in the supercompiler", *Partial Evaluation and Mixed Computation*, 1988, 341–353.

[20] "Программная модель протокола двойной зашифровки", http://refal.botik.ru/preprints/Antonina_Nepeivoda-Double_Proto-14032015v1.ref.

А. Н. Непейвода (A. N. Nepeivoda)
Переславль-Залесский, ИПС РАН
E-mail: a_nevod@mail.ru

Научное издание

Сборник трудов по программированию

Сборник трудов по функциональному языку программирования Рефал, том II, 2015 г.

Под редакцией А. П. Немытых.

Для научных работников, аспирантов и студентов.

Издательство «**Сборник**»,
152025 г. Переславль-Залесский, ул. Строителей 41.
Электронный адрес: **sbornik.pz@gmail.com**

Гарнитура **Computer Modern**. Формат **70×100/16**.
Дизайн обложки: *Н. А. Федотова*. Уч. изд. л. **8.09**.
Усл. печ. л. **12.68**. Подписано к печати **20.08.2015**.
Ответственная за выпуск: *Н. А. Федотова*.

СБОРНИК
Издание трудов научных конференций

Отпечатано в **ООО "Регион"**.
Печать **цифровая**. Бумага **мелованная**. Тираж **100 экз**. Заказ 2.
152025 Ярославская область г. Переславль-Залесский ул. Строителей, д. 41