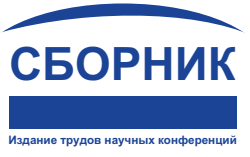
РОССИЙСКАЯ АКАДЕМИЯ НАУК
ИНСТИТУТ ПРОГРАММНЫХ СИСТЕМ
ЛАБОРАТОРИЯ АВТОМАТИЗАЦИИ ПРОГРАММИРОВАНИЯ

СБОРНИК ТРУДОВ
по функциональному языку программирования Рефал

I



ПЕРЕСЛАВЛЬ-ЗАЛЕССКИЙ
2014

УДК 004.42(063)
ББК 22.18

В874

Сборник трудов по функциональному языку программирования Рефал, том I // *Под редакцией А. П. Немытых.* — Переславль-Залесский: Издательство «СБОРНИК», 2014, **194** с. — ISBN 978-5-9905410-1-6

Книга является сборником статей и лекций, посвященных различным сторонам функционального языка программирования Рефал (автор В. Ф. Турчин) и его реализаций. Рассматриваются два диалекта Рефала: Рефал-5 (последний из диалектов, который был разработан В. Ф. Турчиным); FLAC (Functional Language for Algebraic Calculation), ориентированный на алгебраические вычисления (автор В. Л. Кистлеров).
Сборник логически замкнут и 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 978-5-9905410-1-6

Предисловие редактора

Три из предлагаемых читателю статей были написаны для сборника Института программных систем, который так и не был напечатан по «политическим мотивам» конца двадцатого века, как некоторый отчёт о текущем состоянии языка и системы программирования для алгебраических вычислений FLAC, реализация которой была осуществлена Лабораторией автоматизации программирования. На момент написания этих текстов, на персональных вычислительных машинах использовалась операционная система MS DOS. Вычислительные ресурсы самих вычислительных машин были сравнимы с возможностями современных сотовых телефонов – последние даже превосходят первые, что, например, показано в одной из статьей данного сборника. Те же экономические причины остановили и поддержку FLACa.

На мой взгляд, эти три статьи, вместе с открывающей сборник заметкой безвременно ушедшего автора языка FLAC – Виктора Л. Кистлерова, представляют единое целое, описывая язык (являющийся диалектом функционального языка программирования Рефал), принципы его реализации и пример использования. Тексты по принципам реализации Рефала давно не переиздавались и являются библиографической редкостью.

Во второй части открытого читателем сборника представлены тексты, относящиеся к различным сторонам Рефала-5, который является последним диалектом функционального языка программирования Рефал, предложенным непосредственно автором Рефала – Валентином Федоровичем Турчиным: от лекций по языку Рефал до отношения Турчина, заданного на элементах последовательности параметризованных стеков функций. Отношение Турчина связано с метавычислительными возможностями Рефала, который изначально разрабатывался автором как мета-язык программирования.

Данный сборник можно рассматривать как учебное пособие по функциональному программированию. Исходные тексты обсуждаемых реализаций, кроме реализации Рефала-5 на Windows Mobile 5.0, доступны по интернету (см. ссылки в списках литературы, данные в статьях). Перенос FLACa на современные операционные системы, на мой взгляд, является хорошим упражнением для студента, который пожелает закрепить прочитанный материал.

Статьи в сборнике расположены в порядке времени произведения исследований, которые они описывают; исключение сделано лишь для заметки Виктора Л. Кистлерова.

Редактор искренне благодарит Антонину Непейвода за помощь в подготовке сборника.

Андрей П. Немытых

Содержание

Victor L. Kistlerov

The Language FLAC: Computational Model and Modularity

The development of a language for algebraic computation actually needs a specific computation model that supports typical behavior of formula manipulation. For this purpose an "intentional" model with suspended computations was developed, and language FLAC with specific modularity is an implementation of the model.

The development of a language for algebraic computation actually needs a specific computational model that supports typical essential features of algebraic computation.

The base of the model is the *concept of suspended computations*, that regards all functions as partial ones, and result of computation of $f(x_0)$, that is a call of function f at point x_0, specifically depends on the domain of f. It is fundamentally, that if $x_0 \notin Dom(f)$ then the function call $f(x_0)$, in spite of conventional approach, is suspended, i.e. function f is extended at the point x_0 by so called "intensional", that is a ground term $f(x_0)$.The intensional is appended to the set of values, and the computation continues on the extended set of values.

A reason for the model is substantiated by examples of regular means for imitation of constructors for numbers and algebraic expressions. For example, number -1 is regarded as an intesional, arisen as a suspension of function `subtract` for naturals in expression `subtract(0,1)`. Similarly $1/2$ is the same for natural function `divide`: `divide(1,2)`; the imaginary number I is intentional, arisen for real function `sqrt` at the point -1. And finally, the polynomial $x+1$ is the intensional of the function `add`.

The FLAC language [1] (is an abbreviation for Functional Language for Algebraic Computation) is a functional language much similar to Refal [2] with conventional elements of the languages like terms, variables, pattern matching, alternation and recursion. Any function defined in a program is regarded as partial one, and programming in FLAC is extending the functions.

Статья написана в 2008 г.

Here is a syntax of a simple version of the language.

```
      Program  =  Sentence   |   {Sentence";"}

Snt:  Sentence =  Term  "="  Expression

T:    Term   =   Simple-Term   |   Compound-Term
St:     Simple-Term   =   Simple-Ground-Term   |   Variable
SGt:    Simple-Ground-Term = Identifier  |  Number  |  Literal

Ct:   Compound-Term   =   Head  "("  List  ")"
H:      Head   =   Name   |   Term-Variable
Name:   Name   =   Identifier
L:    List   =   Term   |   Term  {","  List}

Gt:  Ground-Term  =  Simple-Ground-Term | Compound-Ground-Term
CGt:    Compound-Ground-Term   =  Name  "(" Ground-Term-List ")"
Gl:     Ground-Term-List   =  Ground-Term {"," Ground-Term-List}


V:   Variable   =   Term-Variable   |   List-Variable
Vt:    Term-Variable   =   "&" Identifier
Vl:    List-Variable   =   "#" Identifier

Id:      Identifier
Num:     Number
Liter:   Literal
```

The following is the famous factorial function written in FLAC:

```
fac(0) = 1;
fac(&n) = &n * fac(&n-1);
```

The most specific feature of the language is to support suspended computations. Any ground term is regarded as functional call and is trying to be converted. If a ground term t calls a function f outside its domain then the term t is suspended and converted to a ground term t' which denotes the result of suspension of the term t. For the sake of usability the t' is represented literally by the ground term t itself. As a result of the suspension the term t' is appending to the set of values for further computations. Actually, we have to consider every resulted ground term as an intensional of suspended computation.

Program is a sequence of definitions. The sentences of a function description give alternative patterns, that are tried one by one. To convert a term `f(a)` pattern matching process starts with the first sentence of description of the function `f`. Each Term-Variable can take only one Ground-Term, and List-Variable can take Ground-Term-List, when matching from left to right. If it is impossible to match the current sentence, the process restarts from the beginning of the next sentence.

For example, let function `apply` applies first argument to each element of the list of the second argument, that is the compound ground term with the name `Terms`:

```
apply(&f,Terms(&x,#l))  =   &f(&x), apply(&f,Terms(#l));
apply(&f,Terms(#l))     =   #l;
```

Compound ground terms are also used for data type representation: the name of a compound term is used as the name of type. For example, matrix

$$A = \begin{pmatrix} cos(\varphi) & -sin(\varphi) \\ sin(\varphi) & cos(\varphi) \end{pmatrix}$$

may be represented as:

```
A = Mat(2,2, Row(cos(fi), -sin(fi)), Row(sin(fi), cos(fi)));
```

Each sentence of description of function f defines also the domain D_i^f, and so the function has the domain $D^f = \bigcup_i d_i^f$.

Usually, programming systems have modularity. Conventional tradition allows make definition of one function only in one of composed modules. However, it is not convenient for algebraic computations, because mathematical tradition dictates necessity of extension of earlier defined operations for new mathematical objects.

FLAC has modularity that allows a once defined function to be extended in other modules. So, if function f has definitions in modules $m_1, \ldots, m_n$ that are composed into a single-module M, then the domain of the function f is $D_m^f = \bigcup_k D_{m_k}^f$.

For instance, if functions `+` and `*` were before only numeric ones, and we have made new module with defined matrix operations named `addmat` and `mulmat`, then we can just add extra definitions:

```
Mat(#l1) + Mat(#l2)  =  addmat(Mat(#l1), Mat(#l2));
Mat(#l1) * Mat(#l2)  =  mulmat(Mat(#l1), Mat(#l2));
```

and now use it in algebraic manner:

$$B = A * A + M;$$

Moreover, on modules $m_1, \ldots, m_n$ may be defined a partial order with corresponding semantics of the function f on each branch of the tree.

Though resulted domain of f does not depend on order of modules in M, but result of computation may essentially depend on it.

Список литературы

[1] V. Kistlerov, "The principals of development of the Computer Algebra Language", Preprint of Institute of Control Sciences (In Russian), Moscow, 1987.

[2] V. F. Turchin, *REFAL-5 programming guide and reference manual*, New England Publishing Co., Holyoke, 1989.

[3] Victor Kistlerov, "An operational semantics of suspended computations", Complex Systems: Control And Modeling Problems, Samara, June 1999.

Victor L. Kistlerov (Victor L. Kistlerov)
Institute of Information Transmission Problems
Russian Academy of Sciences,
Moscow

Е.А.Гайдар, И.М.Игнатович, В.Ф.Козадой, А.П.Немытых, В.А.Пинчук, С.В.Чмутов

Функциональный язык для алгебраических вычислений FLAC

В статье приводится описание языка и системы FLAC, реализованной в ИПС АН СССР. Библ. 10 наим.

Содержание

Работа выполнена в 1988 г.

§ 1. Введение

Язык FLAC (Functional Language for Algebraic Calculation) был разработан в ИПУ АН СССР В. Л. Кистлеровым [1]. Им же был написан на языке РЕФАЛ первый интерпретатор FLACa. В ИПС АН СССР язык FLAC реализован на ЭВМ «Электроника 100-25» (операционная система МНОС РЛ 1.2), «Электроника 79» (операционная система ДЕМОС 2.0), IBM PC XT (MS DOS)[1]. Язык реализации – «Си». В данной статье мы описываем версию языка и системы FLAC, реализованную в ИПС АН СССР на IBM PC XT. Описание реализации см. в [2].

FLAC – функциональный язык программирования, ориентированный на решение задач символьных аналитических вычислений. Одна из задач, стоявших перед В. Л. Кистлеровым при разработке языка, – адекватное отражение предметной области аналитических вычислений. На наш взгляд, в языке FLAC эта задача успешно решена.

Наиболее близким к FLACу языком программирования является РЕФАЛ [3],[4]. Ниже приведено время работы двух тестов[2] в реализации языков FLAC, РЕФАЛ и в системе символьных аналитических преобразований μSimp. Первый тест – это вычисление числа 300!. Результат изложен в таблице, где приведены два числа. Верхнее – это время вычисления 300!, нижнее – время до выдачи результата. Нижнее число больше верхнего, т. к. после вычисления 300! результат нужно перевести в десятичную систему счисления. Второй тест – это проход по линейному списку длиной 5000 элементов.

	FLAC	РЕФАЛ Веденов	РЕФАЛ Хорошевский	РЕФАЛ Турчин	μSimp
1 тест					
2 тест					

Из прикладных программ в ИПС АН СССР на языке FLAC реализованы следующие[3]:

- программа для решения систем линейных уравнений над произвольным евклидовым кольцом (Н. А. Чмутова [6]);
- программа для нахождения базиса резонансных соотношений (В. А. Швачко);
- полиномиальный калькулятор (В. С. Титов);
- программа нахождения базиса Грёбнера полиномиального идеала (Е. А. Гайдар).

[1] Позже реализация была перенесена на операционную систему UNIX.

[2] В сохранившейся у редактора версии этой статьи времена работы тестов отсутствуют. Я не помню относительных результатов второго теста; относительные результаты первого теста соответствовали порядку систем программирования, который указан в таблице, – то есть FLAC на данном тесте работал наиболее эффективно. Веденов, Хорошевский, Турчин – авторы соответствующих реализаций РЕФАЛа. μSimp – актуальная на момент написания статьи система компьютерной алгебры.

[3] Позже С.Д. Мешвелиани реализовал на FLACе большие пакеты программ компьютерной алгебры. Я добавил ссылки на его работы в список литературы [9],[10].

Мы искренне благодарны В. Л. Кистлерову, от которого узнали язык FLAC до публикации [1]. Мы благодарны также Н. В. Кондратьеву и С. М. Абрамову за обсуждение многочисленных проблем реализации FLACа.

Ниже (до конца введения) мы кратко (и неформально) опишем язык FLAC, чтобы читатель смог сориентироваться, что это за язык и нужно ли ему читать эту статью.

Язык FLAC (как и РЕФАЛ) – язык работы со списками термов. Программа на FLACе состоит из описания функций. Описание функций состоит из предложений. Каждое предложение имеет вид

```
<левая часть> = <правая часть>;
```

где `<левая часть> ::= <имя функции>(<список аргументов>)`. Работа программы начинается с вызова некоторой функции. При этом по имени вызываемой функции находится первое предложение из описания этой функции, и делается попытка отождествления аргументов вызова функции с аргументами левой части предложения (образцом). При отождествлении переменные предложения принимают некоторые значения. Если отождествление возможно, то исходный вызов функции заменяется на правую часть предложения (с подставленными значениями переменных), и вызываются функции, указанные в правой части. Если отождествление невозможно, то делается попытка отождествить аргументы вызова с аргументами левой части следующего предложения.

Если таким образом оказываются перебраны все предложения описания данной функции, и отождествление невозможно, то результатом вызова данной функции является сам вызов, т. е. терм

```
<имя функции>(<список аргументов>)
```

Произошла задержка функции.

Переменные в программах бывают трех типов:

1 `&X`, `&Y`, ..., значением которых может быть только один терм;

2 `#X`, `#Y`, ..., значением которых может быть список термов (в том числе и пустой список);

3 `_X`, `_Y`, ..., значением которых может быть только число.

Пример. Реверсирование списка.

Программа состоит из описания одной функции `rev` с двумя предложениями:

```
rev() = ;
rev( &X #Y ) = rev( #Y ) &X;
```

Правая часть первого предложения состоит из пустого списка термов. Первое предложение означает, что для реверсирования пустого списка ничего не нужно делать. Второе предложение означает, что для реверсирования списка нужно реверсировать его «хвост» (переменная `#Y`) и дописать сзади первый элемент (переменная `&X`).

Предположим, что мы вызываем функцию `rev` с аргументами `a` и `b`: `rev(a b)`. Тогда отождествление аргументов вызова функции с аргументами левой части первого предложения неудачно. Отождествление с аргументами левой части второго предложения удачно. При этом `&X` принимает значение `a`, `#Y` – `b`. Следовательно, исходный вызов заменится на `rev( b ) a`. Далее

следует вызов `rev( b )`. Аналогично предыдущему шагу отождествление происходит с левой частью второго предложения. Теперь `&X` принимает значение – `b`, а `#Y` – пустого списка. Следовательно, вызов `rev( b )` заменится на `rev() b`. Т. е. исходный вызов `rev( a b )` заменится на `rev() b a`.

При вызове `rev()` происходит отождествление с левой частью первого предложения, и поэтому вызов `rev()` исчезает (заменяется на пустой список). Т.е. исходный вызов заменяется на список `b a`, что и является результатом вызова функции `rev( a b )`.

Отметим, что в этом примере первое и второе предложения можно поменять местами. Тогда получим более оптимальную программу.

§ 2. Язык FLAC

2.1. Данные языка FLAC.

Наиболее общей структурой данных в языке FLAC является список термов.

Термы бывают простые и аппликативные.

Простой терм – это либо атом, либо число.

Аппликативный терм – это список термов, заключенный в круглые скобки.

Список термов – это несколько последовательно записанных термов. Список может быть пустым.

2.1.1. *Синтаксис данных.*

Для описания синтаксиса данных достаточно привести синтаксические определения простых термов.

Простые термы (атомы и числа) синтаксически представляются последовательностями символов.

Число – это последовательность из цифр `0 1 2 3 4 5 6 7 8 9`, ограниченная символом, отличным от цифры.

Замечания.

Незначащие нули перед числом можно игнорировать. Например, две синтаксические конструкции `012` и `12` изображают один и тот же простой терм.

Числа во FLAC только целые неотрицательные. Система счисления – десятичная. Количество разрядов числа произвольно.

Отрицательные числа представляются аппликативными термами. Например, число -5 представляется аппликативным термом `(-5)`.

Атом – это последовательность символов (не начинающаяся с цифры), ограниченная специальным символом.

Специальный символ – это один из символов:

```
( ) ' " ; / <EOF> <пробел> <табуляция>
<обратная табуляция> [ ] | , = ^ * ! - + .
```

Кроме определенных выше атомов, бывают еще специальные атомы.

Специальный атом представляется либо последовательностью любых символов, заключенных в двойные кавычки (причем сами двойные кавычки в этой последовательности изображаются двумя символами ""), либо одним из следующих символов:

```
/ <EOF> ' [ ] | = ^ * ! - + .
```

Объясним назначение тех специальных символов, которые не являются атомами. Эти символы участвуют в записи данных FLACa, но им самим не соответствует никакой элемент данных языка.

`()` используются для изображения аппликативных термов.

`;` – конец списка.

`"` – для изображения специальных атомов.

`<пробел>` `<табуляция>` `<обратная табуляция>` – разделители термов.

Текстом атома будем называть представляющую атом последовательность символов. Если атом не является специальным, заключённым в двойные кавычки, то символы `<возврат каретки>` и `<перевод строки>` игнорируются.

Два атома с синтаксически одинаковым текстом изображают равные простые термы. Например, `A` и `"A"` – один и тот же терм.

Если список, находящийся в скобках аппликативного терма, не пуст, то его первый элемент называется именем аппликативного терма, а остальные элементы – аргументами.

Важное замечание. Если имя аппликативного терма отлично от специального атома, представленного специальным символом, то его можно выносить за скобки.

2.1.2. *Примеры данных.*

Примеры чисел:

```
5
2
198819891990
```

Примеры атомов:

```
A
"1B"
#
"a b &"
```

`""` – это единственный атом без текста.

Примеры термов:

```
A
123
(B)
(A B())
()
```

Примеры списков термов:

`A B()` – список из двух термов – простого и аппликативного;

`1 5 (A(A(A)))` – список из трех термов: первые два – простые, третий – аппликативный;

`12x`[4] – список из двух термов – числа и атома;

`- 5` – список из двух термов: специального атома и числа.

Разные синтаксические записи одного и того же аппликативного терма:

`A()` и `(A)`;

`A( B() )` , `(A (B) )` , `(A B() )` и `A( (B) )`;

`( () )` и `()()`.

[4]Пробела между `12` и `x` нет.

Следующие списки термов различны:
`A ()` и `A()` ;
`A( B() )` и `A( B () )` ;
`-(5)` и `(-5)` ;
`() ()` и `( () )` .

В последнем примере именем аппликативного терма `(())` является аппликативный терм `()`.

О внутреннем представлении данных во FLAC-системе см. [2].

2.2. Синтаксис программы.

Программа состоит из предложений вида

`<левая часть> = <правая часть>;`

`<левая часть>` – аппликативный терм, имя которого – атом.
`<правая часть>` – список термов. Набор предложений с одинаковым именем левой части называется функцией. Имя левой части называется именем функции. Как правило, предложения программы содержат переменные. Переменные бывают трех типов:

переменная типа терма – это атом, текст которого начинается с символа `&`;

переменная типа списка термов – это атом, текст которого начинается с символа `#`;

переменная типа число – это атом, текст которого начинается с символа `_`.

Текст переменной должен состоять по крайней мере из двух символов. На использование переменных имеется одно ограничение:

левая часть предложения не может содержать более одной переменной типа списка термов на одном уровне скобочной структуры. Т.е. никакой аппликативный терм левой части не содержит двух `#` переменных.

Имеется также ограничение на использование специальных атомов `[` и `]` : на каждом уровне скобочной структуры предложения квадратные скобки `[ ]` образуют правильную скобочную структуру.

Замечания.

Запись списка термов в квадратных скобках эквивалентна записи каждого терма списка в квадратных скобках.

Запись `[<терм>]` эквивалентна `HOLD(<терм>)`.

Пример (программа факториала) P.

```
fact(0) = 1;
fact(&x) = MULT(&x fact(SUB(&x 1)));
```

Другим примером программы служит программа реверсирования списка из введения. Много примеров программ приведено в [6].

2.3. Семантика.

Семантику программ опишем с помощью абстрактной FLAC-машины. Конкретная реализация этой абстрактной машины описана в [2].

Абстрактная FLAC-машина состоит из следующих элементов:

память – область для хранения программ;
результат – список для хранения промежуточных результатов;
стек активных аппликативных термов результата;

исходный вызов – аппликативный терм, подлежащий вычислению;

текущий вызов – указатель на аппликативный терм в результате, подлежащий промежуточному вычислению;

текущее предложение – указатель на предложение программы, с левой частью которого будет производиться отождествление.

Перед началом работы FLAC-машины мы предполагаем, что задан исходный вызов, и в память машины загружена некоторая программа. Эти действия обеспечивает система FLAC, описанная в п.3.

Работа абстрактной FLAC-машины.

1. Инициализация Исходный вызов помещается в результат. В стек помещается указатель на исходный вызов.

2. Проверка конца и начала шага

ЕСЛИ стек пуст,
ТО работа FLAC-машины окончена. В результате хранится вычисленный список термов.

ИНАЧЕ из стека извлекается указатель на аппликативный терм.
ЕСЛИ имя этого аппликативного терма не атом,
ТО переход к действию 2 (функция задержана),
ИНАЧЕ текущий вызов устанавливается на этот аппликативный терм.

3. Проверка на встроенную функцию

ЕСЛИ текущий вызов указывает на аппликативный терм с именем встроенной функции,
ТО происходит вызов встроенной функции. Результат работы функции подставляется в результат машины на место текущего вызова. Переход к действию 2.

4. Шаг В соответствии с правилом выбора предложения (п. 2.3.1) по имени текущего вызова устанавливается текущее предложение.

ЕСЛИ все предложения просмотрены,
ТО переход к действию 2 (функция задержалась),
ИНАЧЕ происходит попытка отождествления (п. 2.3.3) текущего вызова с левой частью текущего предложения.
ЕСЛИ отождествление невозможно,
ТО переходим к действию 4,
ИНАЧЕ аппликативные термы правой части текущего предложения помечаются как активные – по правилу активизации (п. 2.3.2); происходит подстановка значений переменных в правую часть; полученный список помещается в результат на место текущего вызова; указатель на помеченные аппликативные термы помещаются в стек в том порядке, в котором помечались термы; переход к действию 2.

2.3.1. *Правило выбора предложения.*

Рассмотрим функцию с именем текущего вызова. Т.е. набор предложений, имя левой части которых совпадает с именем текущего вызова. Этот набор предложений упорядочен так же, как предложения в нашей программе.

ЕСЛИ таких предложений нет,
ТО все предложения просмотрены,
ИНАЧЕ текущее предложение устанавливается на очередное предложение нашего набора.

К этому правилу имеется дополнение, связанное с модульностью (см. п. 3.2.1).

2.3.2. *Правило активизации.*

Аппликативные термы правой части предложения активизируются в следующем порядке.

Будем просматривать запись правой части предложения справа налево. Если при просмотре встречается круглая правая скобка, то соответствующий аппликативный терм помечается как активный.

Имеются три исключения:

- аппликативный терм с именем `HOLD` и все аппликативные термы внутри его как активные не помечаются;
- аппликативные термы, находящиеся внутри аппликативного терма с именем `QUOTE`, как активные не помечаются;
- первый аргумент аппликативного терма с именем `|` как активный не помечается.

2.3.3. *Отождествление.*

Текущий вызов и левая часть текущего предложения отождествляются успешно, если существуют такие значения переменных, при подстановке которых в левую часть текущего предложения, получим аппликативный терм, равный текущему вызову. Заметим, что значением `&`-переменной может быть только один терм, значением `_`-переменной – число, в том числе и отрицательное, а значением `#`-переменной – список.

Существенное дополнение к определению равенства атомов приведено в п.3.2.1. Это дополнение связано с модульной структурой программ.

2.3.4. *Пример.*

Рассмотрим программу факториала:

```
fact(0) = 1;
fact(_x) = MULT(_x fact(SUB(_x 1)));
```

(Эта программа отличается от программы п.2.2 только заменой `&`-переменной на `_`-переменную).

Предположим, что эта программа загружена в память абстрактной FLAC-машины. Исходный вызов – это аппликативный терм `fact(2)`.

Действие 1. Результат – `fact(2)`. Указатель на этот аппликативный терм помещается в стеке.

Действие 2. Стек не пуст. Извлекаем из него указатель на аппликативный терм `fact(2)`. Имя `fact` этого аппликативного терма – атом. Текущий вызов устанавливается на `fact(2)`.

Действие 3. `fact` не является именем встроенной функции.

Действие 4. Текущее предложение устанавливается на предложение

```
fact(0) = 1;
```

Происходит попытка отождествления текущего вызова `(fact 2)` с левой частью текущего предложения `(fact 0)`. Отождествление невозможно.

Действие 4. Текущее предложение устанавливается на предложение

```
fact(_x) = MULT(_x fact(SUB(_x  1)));
```

Отождествление `(fact 2)` и `(fact _x)` возможно. Переменная `_x` принимает значение 2. Аппликативные термы правой части помечаются как активные в следующем порядке: первым помечается аппликативный терм с именем `MULT`, вторым – `fact`, третьим – `SUB`. 2 подставляется в правую часть.

```
MULT(2 fact(SUB(2 1)))
 1      2    3
```

Цифры внизу означают порядковый номер, которым помечался соответствующий аппликативный терм как активный. Построенный аппликативный терм помещается в результат. В стек помещаются указатели на аппликативные термы с именами `MULT`, `fact`, `SUB`.

Действие 2. Из стека извлекается указатель на последний помеченный аппликативный терм – `SUB(2 1)`. Текущий вызов устанавливается на этот терм.

Действие 3. `SUB` – имя встроенной функции. Эта функция осуществляет вычитание своих аргументов (см. п.4). Результат работы функции равен 1. Результат FLAC-машины:

```
MULT( 2 fact(1) )
 1       2
```

Действие 2. Из стека извлекается указатель на терм `fact(1)`. Текущий вызов устанавливается на этот же терм.

Действие 3. `fact` – не имя встроенной функции.

Действие 4. Текущее предложение устанавливается на

```
fact(0) = 1;
```

Отождествление невозможно.

Действие 4. Текущее предложение устанавливается на

```
fact(_x) =  MULT(_x  fact(SUB(_x  1)));
```

Отождествление возможно. Переменная `_x` принимает значение 1. Результат FLAC-машины:

```
MULT( 2 MULT( 1 fact(SUB(1 1))))
 1        2      3    4
```

Стек определяется цифрами внизу.

Действие 2. Из стека извлекается указатель на `SUB(1 1)`. Текущий вызов устанавливается на этот же терм.

Действие 3. `SUB` – имя встроенной функции. Ее результат `0`. Результат FLAC-машины:

```
MULT(2 MULT(1 fact(0)))
 1      2      3
```

Действие 2. Текущий вызов – `fact(0)`.

Действие 3. `fact` – не имя встроенной функции.

Действие 4. Текущее предложение:

```
fact(0) = 1;
```

Отождествление возможно. Результат FLAC-машины:

```
MULT( 2 MULT(1 1))
 1       2
```

Действие 2. Текущий вызов – `MULT(1 1)`.

Действие 3. `MULT` – имя встроенной функции. Эта функция умножает свои аргументы. Ее результат равен 1. Результат FLAC-машины: `MULT(2 1)`.

Действие 2. Текущий вызов – `MULT(2 1)`.

Действие 3. `MULT` – имя встроенной функции. Результат FLAC-машины: 2.

Действие 2. Стек пуст. FLAC-машина оканчивает работу. Результат работы программы 2.

§ 3. Система FLAC

Язык FLAC реализован в ИПС АН СССР как составная неотъемлемая часть системы FLAC. Основные компоненты системы (за исключением языка): диалог, администратор, инфиксные операции, компилятор. Диалог осуществляет интерфейс с пользователем. Таким образом, FLAC – диалоговая система. Администратор обеспечивает модульную структуру программ, возможность динамически загружать и удалять модули. Инфиксные операции обеспечивают возможность использования в программах инфиксной записи арифметических операций и рациональной арифметики.

3.1. Диалог.

Диалог представляет собой бесконечный цикл. На каждом «витке» цикла система выдает сообщение в виде `._` . После чего система ожидает ввода либо команд, либо вызовов функций. Одна из команд `BYE;`. Это команда выхода из системы. Другие команды переключают режим работы системы. По умолчанию система работает в следующем режиме: если на приглашение вводится список аппликативных термов, то этот список помещается в результат FLAC-машины. Указатели на аппликативные термы вводимого списка помещаются в стек (справа налево). После чего FLAC-машина начинает работу. По окончании работы результат печатается в следующем виде:

```
@: <результат>
```

Затем система опять печатает приглашение и начинает новый «виток» цикла. Ниже приведен пример работы системы в этом режиме.

```
._ PRINT( MULT(2 3) = ) MULT(2 3);

MULT(2 3) =
@: 6
._
```

При вводе очередных вызовов функций можно использовать атом `@:` . В этом случае перед выполнением программы система подставит вместо `@` результат работы системы на предыдущем «витке». Таким образом, результат работы системы на одном из «витков» можно использовать в последующих «витках».

Имеется еще два независимых режима работы системы: режим печати данных и режим полного выполнения. Система может работать одновременно в обоих этих режимах. Режим печати данных отличается тем, что в «витке» имена всех аппликативных термов печатаются внутри скобок. Этот режим работы включается командой `PRINT ON;`, выключается – `PRINT OFF`. Режим полного вычисления отличается от описанного режима тем, что при вводе очередных вызовов функций активизируются (по праву активизации п.2.3.2) все вводимые аппликативные термы (а не только аппликативные термы верхнего уровня вводимого списка). Этот режим включается командой `EVAL ON;`. Выключается – `EVAL OFF`.

Ниже приведены примеры.

```
._ PRINTD ON;

@:
._ MULT(a b) = MULT(2 3);
@: (MULT a b) = 6;
._ EVAL ON;

@:

._ PRINT( MULT(2 3) = ) MULT(2 3);
6 =
@: 6
._
```

Другие возможности интерфейса с пользователем осуществляются вызовами встроенных функций (см. п. 4). В частности, функцией `SYSTEM`.

Вызов `SYSTEM()` приводит к временному выходу из FLAC-системы в DOS. Обратный вход во FLAC-систему осуществляется вводом в DOSe команды `EXIT`. Вызов `SYSTEM(<атом>)` приводит к выполнению команды DOS, которая задается текстом `<атом>`, не выходя из FLAC-системы. Другие встроенные функции, наиболее часто вызываемые из диалога: `TRACE`, `LOAD`, `KILL`. К интерфейсу с пользователем относятся также возможности обработки прерывания `^C`.

3.2. Администратор системы.

Администратор системы – это две встроенные функции `LOAD` и `KILL`.

3.2.1. *Функция `LOAD`.*

Функция `LOAD` обеспечивает модульность программ. Вызов

LOAD(<имя файла>)

загружает в систему все модули, содержащиеся в файле

<имя файла>.cod

Все программы на языке FLAC должны быть оформлены в виде одного или нескольких модулей. Каждый модуль имеет вид:

```
module <имя модуля>;
PORT(<список доступных атомов>);

<программа>

end;
```

Модульность программ означает в частности, что множество всех атомов в системе разбивается на части. Два атома с совпадающими текстами, но принадлежащие разным частям, считаются различными. Два атома равны тогда и только тогда, когда они принадлежат одной и той же части и их тексты совпадают. Это замечание существенно дополняет определение отождествления из п.2.3.3.

Определим теперь эти части. Одна часть – это доступные атомы. В нее входят:

- все атомы, вводимые в систему в диалоге;
- все атомы, указанные в списках PORT всех модулей, загруженных в систему;
- все имена всех модулей, загруженных в систему;
- список общедоступных атомов:
  / EOF ' [ ] | = ^ * ! - + PUSH POP TOP RETOP NIL FIRST REST NL SPACE nl space BELL OPEN CLOSE PRINT PRINTD FPRINT FOUT READ FREAD GETB PUTB GETBYTE ADD SUB MULT DIV LESS RANDOM SYSTEM SYNTAX RUNEND RECLAIM TIME TRACE TYPE PRESS EXPLOD HOOD CFLAC LOAD KILL LIST HOLD QUOTE EVAL module end infix @ PORT ERR TRUE FALSE N OFF ON BYE .

Кроме доступных атомов с каждым модулем связывается своя часть атомов – атомы, закрытые в модуле. В эту часть входят все атомы, имеющиеся в данном модуле, за исключением: имени модуля, атомов, указанных в списке PORT данного модуля, общедоступных атомов.

Таким образом, связь модуля с внешней средой (по отношению к данному модулю) осуществляется через список PORT и общедоступные атомы.

Еще одно важное свойство модульности – это возможность описывать одну функцию в разных модулях (имя функции должно быть указано в списках PORT). В этом случае необходимо уточнить, как происходит выбор предложения (см. п.2.3.1) для отождествления.

Все модули в системе упорядочены по загрузке. Если функция описана в нескольких модулях, то вначале перебираются все предложения данной функции, находящиеся в первом модуле (по загрузке), затем во втором, и т.д.

Пример использования модулей см. в [2]. Много примеров модулей приведено в [6].

3.2.2. *Функция KILL.*

Формат вызова этой функции – `KILL(<имя модуля>)`. Функция удаляет указанный модуль из системы. О реализации администратора см. [2].

3.3. Инфиксные операции и рациональная арифметика.

Для работы с рациональными числами в системе FLAC имеются функции сложения, вычитания, умножения, сравнения и деления. Их имена `+ - * LESSR /` . Естественная математическая форма записи этих функций – инфиксная. Для работы (компиляция и выполнение программы) с функциями в инфиксной записи и с функциями рациональной арифметики необходимо загрузить в систему файл `infix`, т.е выполнить вызов `LOAD(infix)`.

3.3.1. *Инфиксные операции.*

В этом пункте мы опишем программу для обработки инфиксных операций. На вход этой программе поступает список. Список может содержать имена инфиксных операций. В этом случае программа преобразует инфиксную форму операций в функциональную форму. Т.е. имя операции и ее аргументы заключаются в скобки (имя операции на первом месте).

Файл `infix.fl` содержит описание следующих операций:

`+` – операция унарного и бинарного сложения;
`-` – операция унарного и бинарного вычитания;
`*` – бинарная операция умножения;
`/` – бинарная операция деления;
`^` – бинарная операция возведения в степень;
`!` – унарная операция вычисления факториала;
`'` – унарная операция задержки в модуле (см. описание встроенной функции `QUOTE`);
`|` – унарная операция глобальной задержки;
`,` – операция разделения термов.

Файл `infix.fl` доступен пользователю. Таким образом, пользователь может определить свои новые инфиксные операции, удалить имеющиеся, изменить приоритеты имеющихся операций, доопределить операции на других объектах. Например, доопределить операции сложения, вычитания и умножения на элементах какого-нибудь кольца (см. [6])

Пример.

При загруженном файле `infix` список `3! * -2` после ввода преобразуется в список `( * (! 3) (- 2) )`.
Ниже приведен фрагмент экрана дисплея.

```
._  LOAD(infix);

module infix ......

module operation .....

@ :

._ PRINTD ON;

@ :

._ 3! * - 2;
@ : ( * (! 3) (- 2) )

._ EVAL ON;

@ :

._ 3! * - 2;

@ : - 12

._
```

Важное замечание.
| – это встроенная функция, написанная на FLACe: (| #x) = #x . Эта встроенная функция используется для глобальной задержки. Например, если в правой части предложения мы хотим использовать аппликативный терм F(&x) как данные, то для предотвращения вычисления вызова F(&x) следует писать (| F(&x)). В этом случае, согласно статистики (п.2.3.2) аппликативный терм F(&x) как активный не помечается, а вызов (| F(&x)) приведёт к значению F(&x). Далее это значение может использоваться как данные. | – инфиксная операция. Поэтому, если мы будем компилировать нашу программу с загруженным файлом infix, то вместо (| F(&x)) следует писать | F(&x). Недостающие скобки проставляет программа инфиксных операций. Отметим еще, что встроенная функция | выполняется во время компиляции. Поэтому использование инфиксной операции | повышает эффективность программ.

3.3.1.1. Алгоритм расстановки недостающих скобок в инфиксных операциях.

Типом инфиксной операции назовем пару (l , r) , где l – число аргументов, записываемых слева от знака операции, а r – число аргументов, записываемых справа от знака операции. Все инфиксные операции разобьем на три класса:

I операции типа (0 r) – префиксные операции;
II операции типа (l r), где $l \neq 0$ и $r \neq 0$;
III операции типа (l 0) – постфиксные операции.

Один и тот же знак может обозначать разные операции. Эти операции должны относиться к разным классам. Например, знаком + обозначается как унарная операция I-го класса, так и бинарная операция II-го класса. Информация об операциях, обозначаемым одним и тем же знаком, содержится в описании знака операции.

Описанием знака операции мы называем список трех термов. Если данный знак не обозначает операции i-го класса, то на i-м месте списка стоит терм `FALSE`. Если данный знак может обозначать операцию i-го класса, то на i-м месте списка стоит следующий аппликативный терм:

`( <имя операции> (l r)` $\mathtt{l}_x$ $\mathtt{r}_x$ `)` ,

где `(l r)` – тип операции, а $\mathtt{l}_x$, $\mathtt{r}_x$ – левый и правый приоритеты операций (см. ниже).

Например, описание языка операции + – это следующий список:

`(+ (0 1) 210 190) (+ (1 1) 100 100) FALSE`

т.е. знак + может обозначать как унарную (префиксную) операцию, так и бинарную операцию. Постфиксную операцию (класса III) знак + обозначать не может. В унарном случае единственный аргумент стоит после знака операции, а левый и правый приоритеты равны 210 и 190. В бинарном случае знак операции стоит между аргументами, а приоритеты 100 и 100. Ниже приведена таблица описаний знаков всех операций системы FLAC.

Знак операции	Описание знака операции
+	(+ (0 1) 210 190) (+ (1 1) 100 100) FALSE
-	(- (0 1) 210 190) (- (1 1) 100 100) FALSE
/	FALSE (/ (1 1) 196 196) FALSE
*	FALSE (* (1 1) 195 195) FALSE
^	FALSE (^ (1 1) 200 199) FALSE
!	FALSE FALSE (! (1 0) 255 255)
’	(’ (0 1) 255 255) FALSE FALSE
|	(| (0 1) 255 255) FALSE FALSE
,	FALSE (, (1 1) 255 255) FALSE

Алгоритм расстановки скобок представляет собой конечный автомат с двумя состояниями:

- `St` = `At` – состояние, при котором был аргумент и может последовать знак операции.
- `St` = `Bg` – начальное состояние, при котором ещё не было аргументов.

Алгоритм состоит из двух этапов. На первом этапе определяется класс операции. На втором расставляются скобки.

Класс операции определяется по описанию знака операции и текущему состоянию. В следующей таблице приведены правила для определения класса

операции и состояния, в которое переходит конечный автомат после просмотра знака операции.

	Описание операции			Текущее состояние	Класс	Новое состояние
	I	II	III			
1	FALSE	FALSE	FALSE	&x	–	At
2	FALSE	FALSE	true	&x	III	At
3	FALSE	true	FALSE	&x	II	Bg
4	FALSE	true	true	&x	III	At
5	true	FALSE	FALSE	&x	I	Bg
6	true	FALSE	true	At	III	At
7	true	FALSE	true	Bg	I	Bg
8	true	true	FALSE	At	II	Bg
9	true	true	FALSE	Bg	I	Bg
10	true	true	true	At	III	At
11	true	true	true	Bg	I	Bg

В этой таблице через `&x` мы обозначаем произвольное состояние.

Первая строка таблицы отвечает терму, не являющемуся знаком операции. Объем статьи не позволяет останавливаться на причинах выбора именно таких правил определения класса. Так как файл `infix.fl` доступен пользователю, то он сам может придумать свои собственные правила. На втором этапе скобки проставляются согласно типу операции. При расстановке скобок возникает конкуренция между операциями за аргументы. Например, в списке

```
5 * 2 - 3
```

две операции `*` и `-` конкурируют между собой за аргумент 2. Победа в этой конкуренции определяется следующим образом. Рассмотрим два числа. Первое – правый приоритет левой операции (в нашем примере это $\mathtt{r}_x$ для `*`, т.е. 195). Второе – левый приоритет правой операции (в нашем примере $\mathtt{l}_x$ для бинарного, т.е 100). Если первое число больше или равно второму, то побеждает левая операция. В противном случае побеждает правая операция. В нашем примере $195 > 100$. Т.е. побеждает левая операция. Следовательно, скобки расставляются так

```
(- (* 5 2) 3)
```

Опишем теперь алгоритм расстановки скобок более подробно. Это стандартный алгоритм конечного автомата с магазином. Отличие от обычного алгоритма состоит в том, что операция помещается в магазин вместе со своим правым приоритетом, а выталкивается из магазина левым приоритетом другой операции (конкурирование операций). Кроме того, в магазин вместе с операцией помещается тип операции и все термы, которые могут быть левыми аргументами операции. Без ограничения общности будем анализировать только линейный список. Этот входной список просматривается слева направо. Если очередной терм списка не является знаком операции, то он переносится в «сумку» `res` левых аргументов операции. В конце работы в этой «сумке» будет лежать результат. Кроме этой «сумки», имеется еще магазин m. В начальный момент магазин пуст.

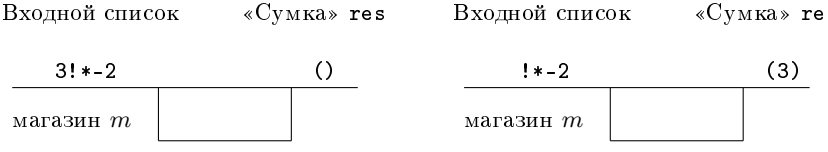
Если очередной терм – знак операции, то определяется класс этой операции. Из описания знака операции извлекается соответствующий аппликативный терм. Операция определена. Остается расставить скобки. Эта операция конкурирует с операцией из магазина. Если побеждает операция из магазина, то на нее (вместе с нужным числом аргументов) навешиваются скобки. При этом левые аргументы лежат в магазине вместе с операцией, а правые находятся в «сумке» `res`. Полученный аппликативный терм помещается в «сумку» `res`. Он может быть аргументом нашей операции. Теперь наша операция конкурирует с очередной операцией из магазина. Если побеждает наша операция, то она кладется в магазин вместе со своим типом, «сумкой» `res` и правым приоритетом. Таким образом, конкурентная борьба каждой операции в конце концов заканчивается тем, что эта операция помещается в магазин вместе с «сумкой» `res`. Далее рассматривается очередной терм входного списка. По окончании входного списка магазин освобождается.

Пример 1. Пусть входной список – это

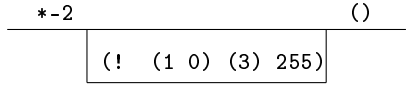
```
3!*-2
```

В этом списке пять термов[5].

Первый терм `3` не является знаком операции. Следовательно, терм `3` переносится в «сумку» `res`. Схематически будем обозначать это так:

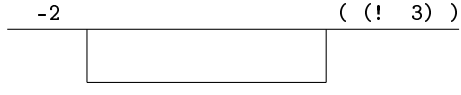


Следующий терм `!`. Это знак операции класса III. Магазин пуст. Операция `!` помещается в магазин.



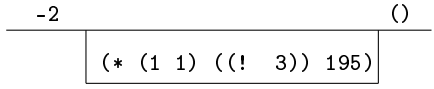
Заметим, что «сумка» `(3)` переместилась в магазин вместе с операцией `!`. Конечный автомат переходит в состояние `At`.

Очередной терм `*`. Это бинарная операция, она начинает конкурировать с операцией `!` из магазина. Левый приоритет `*` равен `195`, правый `255`. Побеждает факториал. Следовательно, факториал выталкивается из магазина с расстановкой скобок.



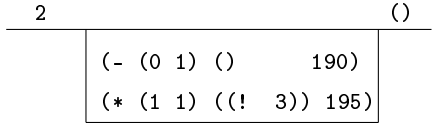
[5] В данном случае разделение элементов списка пробелами, согласно синтаксису FLACa, не обязательно.

Магазин пуст. Теперь нашей операции * конкурировать не с чем. Значит, * помещается в магазин.

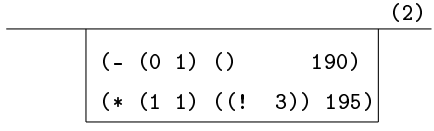


Конечный автомат переходит в состояние Bg.

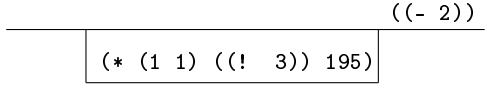
Очередной терм -. Это унарная операция. (см. таблицу определения класса). Ее левый приоритет 210. Правый приоритет операции * из магазина 195. Побеждает минус. Следовательно, минус помещается в магазин.



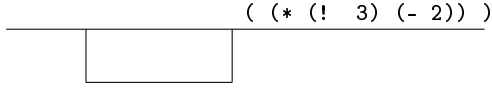
Очередной терм 2. Это не знак операции.



Входной список закончился. Освобождаем магазин. Первой из магазина выходит операция - . У этой операции один правый аргумент. Он лежит в «сумке» res. При выходе из магазина навешиваются скобки.



Теперь из магазина выходит операция *. У нее один левый аргумент и один правый. Левый лежит в магазине вместе со *. Это терм (! 3). Правый аргумент лежит в «сумке» res. Это терм (- 2). Навешивая скобки, получим



В «сумке» res лежит результат (* (! 3) (- 2)).

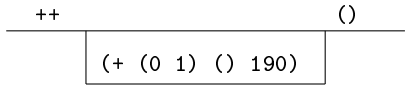
Пример 2. Пусть входной список

+++

В этом списке три терма[6].

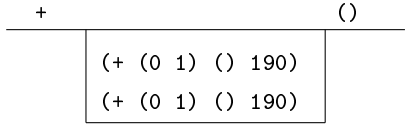
Первый терм +. Это знак операции. Это операция класса I. Помещаем ее в магазин.

[6] В данном случае разделение элементов списка пробелами, согласно синтаксису FLACa, не обязательно.

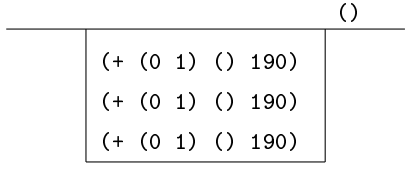


Конечный автомат остается в состоянии `Bg`.

Очередной терм `+`. Это опять унарная операция. Ее левый приоритет `210`. Правый приоритет операции из магазина `190`. Побеждает наша операция. Значит, она помещается в магазин.

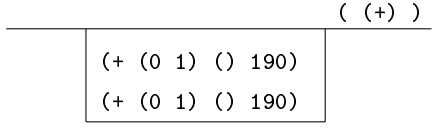


Очередной терм `+`. Аналогично предыдущему имеем:

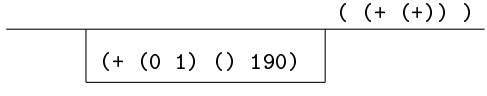


Теперь освобождается магазин. Унарная операция `+` имеет один правый аргумент. Правые аргументы должны находиться в «сумке» `res`.

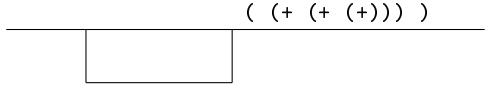
Но «сумка» пуста. Следовательно, имеем



Следующий плюс выходит из магазина.



Затем



Итак, получили результат `(+ (+ (+)))`.

3.3.1.2. Программа инфиксных операций – пример программы на FLACe.

В этом пункте мы приведем программу на FLACe, реализующую алгоритм предыдущего пункта. На примере этой программы демонстрируется еще одно замечательное свойство FLACa – возможность использовать вычисляемые имена функций.

Вот как выглядит начало этой программы:

```
Scan( &st &res &m &1 #x )
                      = ( Func(PrOp(&1) &st) &res &m &1 #x );
Scan( &st &res (#m) ) = Clear(&res  #m);
```

Здесь переменные имеют следующий смысл:

`&st` – состояние конечного автомата;

`&res` – «сумка»;

`&m` – магазин;

`&1` – первый терм входного списка;

`#x` – «хвост» входного списка.

Второе предложение функции `Scan` означает, что по окончании входного списка следует вызвать функцию `Clear` «очистки» магазина. Теперь рассмотрим правую часть первого предложения. Функция `PrOp` выдает в качестве результата описание знака операции `&1`. Функция `Func` определяет класс операции (согласно приведенной выше таблице) и новое состояние конечного автомата. Кроме этого, функция `Func` выдает один из термов `Move` или `Push`. Этот терм как раз и есть обещанное вычисляемое имя функции. Он стоит первым в списке, выдаваемом функцией `Func`. Следовательно, после вычисления функции `Func` этот терм окажется именем аппликативного терма, который является правой частью первого предложения функции `Scan`. Т.е., вслед за функцией `Func` будет вычисляться одна из функций `Move` или `Push`. Функция `Move` вычисляется только в том случае, если `&1` не является знаком операции.

Приведем описание функции `Func`.

```
/*  1         */
Func(FALSE  FALSE  FALSE &x)  = Move          At;
/*  2,4       */
Func(FALSE  &y  (#true)  &x)  = Push (#true)  At;
/*  6,10      */
Func((#t)  &y  (#true)   At)  = Push (#true)  At;
/*  3         */
Func(FALSE (#true) FALSE &x)  = Push (#true)  Bg;
/*  8         */
Func((#t) (#true) FALSE  At)  = Push (#true)  Bg;
/*  5,7,9,11 */
Func( (#true)  #x )           = Push (#true)  Bg;
```

Теперь остается описать функции `Move` и `Push`. Функция `Move` простая. Она переносит очередной терм `&1` в «сумку» `res` и продолжает сканирование входного списка. На FLACe это записывается короче, чем на русском языке:

```
Move(&st (#res) &m &1 #x) = Scan(&st (#res &1) &m #x);
```

Функция Push описывается так:

```
Push(&z &st &res &m &1 #x)
                    = Scan(&st () Cmp(&z &res &m) #x);
```

В этом предложении появилась новая переменная &z. Эта переменная принимает значение аппликативного терма из описания знака операции &1. В правой части функции Push стоит вызов функции Cmp. Функция Cmp организует конкуренцию между нашей операцией &1 (информация о ней содержится в переменной &z) и операциями из магазина. По окончании конкурентной борьбы операция &1 окажется в магазине. «Сумка» res при этом станет пустой (() в правой части предложения Push). Затем функция Scan продолжает рассмотрение входного списка.

Результатом функции Cmp должен быть новый магазин. Вот описание этой функции:

```
Cmp( (#x &lx &rx) &res ((#y &ry) #m) ) =
    Cmp1( (#x &lx &rx) LESS(&ry &lx) &res  ((#y &ry) #m) );
```

Поясним смысл переменных.

- (#x &lx &ry) – аппликативный терм из описания знака нашей операции. При этом &lx и &rx – левый и правый приоритеты нашей операции.
- (#y &ry) – аппликативный терм из магазина. &ry – правый приоритет соответствующей операции.

Для конкуренции операций нужно сравнить правый приоритет &ry операции из магазина с левым приоритетом &lx нашей операции. Это делает функция LESS(&ry &lx). Если эта функция выдаст значение FALSE, значит &ry > &lx, т.е. побеждает операция из магазина:

```
Cmp1( &z FALSE &res (&op((&l &r) &res1 &ry) #m) ) =
        Cmp( &z Br(&res1 &op &res &l &r) (#m) );
```

Если функция LESS выдаст значение TRUE, то побеждает наша операция. Соответствующий аппликативный терм нужно поместить в магазин. На этом конкуренция заканчивается.

```
Cmp( (#x &lx &rx) #y &res (#m) ) = ( (#x &res &rx) #m);
```

Во втором предложении функции Cmp1 нужно только пояснить, что переменная #y всегда будет принимать значение TRUE, выдаваемое функцией LESS.

Первое предложение функции Cmp1 нужно рассмотреть подробнее. В его левой части аппликативный терм

```
&op( (&l &r) &res1 &ry )
```

имеет смысл первого аппликативного терма магазина, где (&l &r) – тип операции op, &res1 – «сумка», из которой будут выбираться левые аргументы операции &op. В правой части предложения стоит вызов функции Cmp, т.е. нужно продолжить конкуренцию нашей операции &z с операциями из магазина. При этом аппликативный терм

```
&op( (&l  &r) &res1 &ry )
```

удаляется из магазина. Новый магазин – (#m). А на операцию &op и на ее аргументы (левые из «сумки» &res1, правые из «сумки» &res) нужно навесить

скобки и поместить в новую «сумку». Это выполняет функция `Br`. Выпишем отдельно вызов этой функции: `Br( &res1 &op &res &l &r )`. Результатом должна быть новая «сумка». `Br` – рекурсивная функция. Сначала мы будем из «сумки» `&res1` выбирать новые аргументы операции `&op`, одновременно уменьшая `&l`. Когда `&l` дойдет до нуля, будем выбирать правые аргументы из сумки `&res`, уменьшая `&r`. Выбранные аргументы мы будем обозначать `#arg`. При вызове `Br` переменная `#arg` принимает значение пустого списка. Теперь приведем описание функции `Br`.

```
Br( (#res1) &op #arg (#res) 0 0) = ( #res1 &op(#arg) #res);

Br( &res1 &op #arg (&1 #res) 0 &r ) =
                Br( &res1 &op #arg (#res) 0 SUB(&r 1) );

Br( (#res1) &op #arg () 0 &r ) = ( #res1 &op(#arg) );

Br( (#res1 &1) &op #arg &res &l &r ) =
                Br( (#res1) &op &1 #arg &res SUB(&l 1) &r);

Br( () &op #arg &res &l &r) = Br( () &op #arg &res 0 &r);
```

Первое предложение – это выход из рекурсии. На операцию `&op` и аргументы `#arg` навешиваются скобки: `&op(#arg)`. Из полученного аппликативного терма и остатков «сумок» `#res1` и `#res` формируем новую «сумку»: `( #res1 &op(#arg) #res )`.

Второе предложение – это выборка правых аргументов операции. При этом уменьшается `&r`.

Третье предложение – это ситуация когда еще нужны правые аргументы, а сумка, из которой их выбирать, уже пуста. В этом случае нужно формировать новую «сумку»: `( #res1 &op(#arg) )`.

Четвертое предложение – это выборка левых аргументов операции. При этом уменьшается `&l`.

Пятое предложение – это ситуация когда еще нужны левые аргументы, а соответствующая сумка пуста. Тогда нужно выбирать новые аргументы.

Функция `Br` используется еще при «очистке» магазина:

```
Clear( &res &op( (&l &r) &res1 &ry ) #m )
                = Clear( Br(&res1 &op &res &l &r)  #m);
Clear( (#res) ) = #res;
```

Описание программы закончим описанием функции `PrOp`:

```
PrOp(+) = (+ (0 1) 210 190)  (+ (1 1) 100 100)        FALSE     ;
PrOp(-) = (- (0 1) 210 190)  (- (1 1) 100 100)        FALSE     ;

PrOp(/) =       FALSE        (/ (1 1) 196 196)        FALSE     ;
PrOp(*) =       FALSE        (* (1 1) 195 195)        FALSE     ;

PrOp(^) =       FALSE        (^ (1 1) 200 199)        FALSE     ;

PrOp(!) =       FALSE              FALSE        (! (1 0) 255 255);
```

```
PrOp(') = (' (0 1) 255 255)        FALSE              FALSE     ;

PrOp(|) = (| (0 1) 255 255)        FALSE              FALSE     ;

PrOp(,) =       FALSE        (, (1 1) 255 255)         FALSE     ;
```

Отметим еще, что эта программа несколько отличается от программы, поставляемой в файле `infix`. Системная программа в файле `infix` более корректно работает с задержкой функций.

3.3.2. *Рациональная арифметика.*

Арифметические функции работы с рациональными числами - это функции:
`/` – функция деления;
`+` – функция сложения;
`-` – функция вычитания;
`*` – функция умножения;
`^` – функция возведения в степень;
`!` – функция факториала (для неотрицательных целых чисел);
`LESSR` – функция сравнения рациональных чисел.

Эти функции описаны в модуле `operation` в файле `infix.fl`. Описания этих функций используют инфиксные операции. Например, функция `+` может быть описана (в случае целых аргументов) так:

```
_x  +  _y = ADD(_x  _y);
```

3.4. Компилятор.

Система FLAC выполняет программы на некотором промежуточном языке. Компилятор с FLACa на этот язык содержится в модуле `CFLAC`. Для использования компилятора его необходимо загрузить: `LOAD(CFLAC);`. Формат вызова компилятора

```
CFLAC( <имя файла без расширения> );
```

Компилятор компилирует файл с именем `<имя файла>.fl`. Промежуточный код помещается в файл с именем `<имя файла>.cod`. Для загрузки файла `<имя файла>.cod` в систему нужно выполнить вызов

```
LOAD(<имя файла>);
```

§ 4. Встроенные функции

Функции ввода-вывода:

OPEN	CLOSE	PRINT	PRINTD	FPRINT	FOUT	SPASE
READ	FREAD	GETB	PUTB	GETBYTE	BELL	NL

Системные функции:

SYSTEM	SYNTAX	RUNEND	RECLAIM	TIME	TRACE	LOAD
TYPE	PRESS	EXPLOD	HOOD	LIST	FIRST	KILL
REST	NIL	nl	space	EVAL		

Функции работы с глобальными переменными:

```
POP     PUSH      TOP     RETOP
```

Функции арифметики:

```
ADD     SUB       MULT      DIV        LESS       RANDOM
```

Функции задержки:

```
QUOTE    '      |
```

4.1. Функции ввода – вывода.

Функция OPEN. Функция открытия файла. Формат вызова:

OPEN(<имя> <мода>)

, где <имя> – имя открываемого файла, <мода> – один из символов r,w,a. В зависимости от моды файл открывается на чтение, запись и дополнение соответственно. При открытии файла система присваивает ему некоторый номер. Этот номер выдается функцией OPEN в качестве результата. Число одновременно открытых файлов в системе не может превышать 10. Система имеет всегда 3 открытых файла:

стандартный файл ввода (с клавиатуры) – номер 1;

стандартный файл вывода (на экран) – номер 0;

стандартный файл для сообщений об ошибках – номер 2.

Пример вызова: `OPEN(infix.fl r);`

Функция CLOSE. Функция закрытия файла. Формат вызова:

CLOSE(<номер файла>)

Функция закрывает файл с указанным номером. Результат функции – пустой список.

Функция PRINT. Формат вызова:

PRINT(<список>)

Функция печатает <список> на экране. Имена всех аппликативных термов печатаются за скобками. Специальный атом, заключенный в двойные кавычки, печатается без этих кавычек. Результат функции – пустой список.

Пример: вызов `PRINT("A B" (A B))` приведет к печати

```
A B A(B)
```

Функция PRINTD. Формат вызова:

PRINTD(<список>)

Функция печатает <список> на экране. Имена всех аппликативных термов печатаются внутри скобок. Отрицательные и рациональные числа печатаются в инфиксной форме. Результат – пустой список.

Пример: вызов `PRINTD("A B" A(B) (-1) (/ 1  2))` приведет к печати

```
A B (A B) -1 1/2
```

Функция FPRINT. Функция аналогична функции PRINT. Формат вызова

FPRINT(<номер> <список>)

Функция печатает <список> в файл с номером <номер>. Результат – пустой список.

Функция FOUT. Аналогична функции FPRINT. Формат вызова:

FOUT(<номер> <список>)

Отличие от функции FPRINT состоит в том, что в случае необходимости (наличии в тексте атома специальных символов), специальные атомы печатаются в двойных кавычках. Результат – пустой список.

Пример: вызов FOUT(1 "A_B" "AB") приведет к печати на экране

"A_B" AB

Функция READ. Функция ввода с клавиатуры. Формат вызова:

READ()

Вводимый список ограничивается символом ; . Результат функции – вводимый список.

Функция FREAD. Аналогична функции READ. Формат вызова:

FREAD(<номер>)

Вводит список из файла с указанным номером. Результат функции – вводимый список.

Функция GETB. Функция ввода одного символа. Ввод буферизованный. Результат – код ASCII вводимого символа. Формат вызова:

GETB()

Функция PUTB. Функция вывода символа. Формат вызова

PUTB(<число>)

Функция печатает на экране символ с ASCII – кодом, равным <числу>.

Функция GETBYTE. Функция ввода одного символа. Ввод небуферизованный. Результат – код ASCII введенного символа.
Формат вызова:

GETBYTE()

Функция BELL. Формат вызова

BELL()

Функция выводит символ – звонок. Результат функции – пустой список.

Функция NL. Формат вызова

NL(<число>)

Функция переводит указанное число строк на экране. По умолчании <число> = 1, т.е. вызовы NL(1) и NL() приведут к переводу строки на экране. Результат функции – пустой список.

Функция SPACE. Формат вызова

SPACE(<число>)

Функция печатает указанное число пробелов. По умолчанию <число> = 1.

4.2. Системные функции.

Функция SYSTEM. Результат функции – пустой список. Имеется два формата вызова этой функции. Первый:

SYSTEM()

Функция в этом случае работает следующим образом. Совершается временный выход в DOS. Состояние FLAC-системы при этом сохраняется. Возврат во FLAC-систему производится вводом в DOSe команды EXIT. Эта функция позволяет пользоваться редакторами текста и другим программным обеспечением, сохраняя при этом состояние FLAC-системы.

Второй формат вызова:

SYSTEM(<атом>)

В этом случае работа функции представляет собой выполнение команды DOS, представленной текстом данного атома.

Функция SYNTAX. Функция генерирует синтаксическую ошибку. Обычно эта функция используется вместе с функцией RUNEND (см. [6]). При возникновении ошибок система находит ближайшую в стеке функцию RUNEND и выполняет ее.

Функция RUNEND. Функция перехвата ошибки. Если функция работает после ошибки в системе, то результат функции – список термов. Первый – число, код ошибки. Второй – аппликативный терм ERR(<вызов> <список>), где <вызов> – это вызов функции, при работе которой возникла ошибка, а <список> – это аргументы функции RUNEND в момент ошибки. Если вызов

RUNEND(<список>)

работает при отсутствии ошибки, то результатом будет список

0 N(<список>)

Использование пары RUNEND – SYNTAX продемонстрировано в [6].

Функция RECLAIM. Формат вызова

RECLAIM()

Результат функции – список из двух чисел. Первое – число свободных элементов памяти. Второе – размер свободной области (в байтах) под тексты атомов.

Функция TIME.

Функция TRACE. Функция трассировки. Функция имеет два формата вызова. Первый формат:

TRACE(<имя>)

Вызов этой функции приводит к трассировке функции с указанным именем. Трассируются только функции с общедоступными именами. Трассировка одного вызова заключается в следующем. Печатается вызов и правая часть (с подставленными значениями переменных) того предложения, с левой частью

которого данный вызов отождествляется успешно. Заканчивается трассировка другим вызовом функции TRACE. Второй формат вызова:

TRACE()

Этот вызов приводит к трассировке всех функций. Заканчивается трассировка другим вызовом TRACE().

Функция TYPE. Формат вызова:

TYPE(<терм>)

Функция определяет тип терма. Результат функции число:

1, если <терм> – атом;
2, если <терм> – число;
3, если <терм> – аппликативный терм;
4, если <терм> – имя встроенной функции.

Функция PRESS. Формат вызова:

PRESS(<список>)

Все элементы списка должны быть атомами. Результат функции – атом с текстом, являющимся конкатенацией текстов из <списка>.

Функция EXPOLOD. Формат вызова:

EXPOLOD(<атом>)

Результатом функции является список атомов. На i-ом месте списка стоит атом, текст которого состоит из одного (i-го) символа данного <атома>.

Функция HOOD. Формат вызова:

HOOD(<атом> <список атомов>)

Функция проверяет, совпадает ли первый символ <атома> с текстом одного из атомов <списка атомов>. Если совпадают, то результатом функции является список двух атомов. Первый атом – это тот элемент <списка атомов>, с текстом которого совпадает <атом>. Второй атом – это <атом>. Если не совпадают, то результатом будет <атом>.

Пример. Вызов HOOD(&A & # -) имеет результатом список & &A .

Функция LIST. Функция имеет два формата вызова. Первый:

LIST()

Результатом является список всех доступных атомов. Второй вызов:

LIST(<имя модуля>)

Результатом является список имен всех функций в данном модуле.

Функция FIRST. Формат вызова:

FIRST(<терм> <список>)

Результатом является <терм>.

Функция REST. Формат вызова:

REST(<терм> <список>)

Результатом является <список>.

Функция NIL. Результатом функции всегда является пустой список.

Функция `nl`. Формат вызова:

`nl(<число>)`

Результатом является атом, текст которого состоит из указанного числа символов перевода строки.

Функция `space`. Формат вызова:

`space(<число>)`

Результатом является атом, текст которого состоит из пробелов.

Функция `EVAL`. Формат вызова:

`EVAL(<список>)`

Функция помечает все аппликативные термы как активные. После чего производит вычисление всех помеченных термов. Список, полученный в результате, является результатом функции `EVAL`. Функция `EVAL` активизирует аппликативные термы, не взирая ни на какие способы задержки. Интересно, что эта функция написана на FLACe (см. [2]).

Функция `LOAD`. Формат вызова:

`LOAD(<имя файла>)`

Функция загружает файл с именем `<имя файла>.cod` в систему. Если файл с таким именем уже имелся в системе, то старый файл автоматически удаляется из системы. Эта функция подробно описана в п. 3.2.1.

Функция `KILL`. Формат вызова:

`KILL(<имя модуля>)`

Функция удаляет модуль с указанным именем.

К системным функциям относится также функция компиляции `CFLAC` (см. п.3.4).

4.3. Функции работы с глобальными переменными.

Работа с глобальными переменными организована в системе FLAC с помощью стеков. Стек имеет имя. Имя может быть произвольным термом. В стек можно положить терм, взять терм, изменить терм в верхушке стека, посмотреть терм в верхушке стека. Содержимое стеков сохраняется на время всего сеанса работы с системой.

Функция `PUSH`. Формат вызова:

`PUSH(<имя> <список>)`

Функция `PUSH` последовательно заносит в стек с именем `<имя>` все элементы `<списка>` (слева направо). Результат вызова – пустой список. Если стека с указанным именем не существует, то он заводится.

Функция `POP`. Функция имеет два формата вызова. Первый:

`POP(<имя>)`

Из стека с указанным именем извлекается элемент. Этот элемент – результат вызова. Если в стеке нет больше элементов, то стек удаляется. Второй формат:

`POP(<имя>())`

Результат вызова – список всех термов в стеке, начиная с нижнего терма. Стек удаляется.

Функция TOP. Функция имеет тот же формат вызова, что и POP, и тот же результат. Отличие этой функции от POP состоит в том, что при работе TOP стек остается неизменным. Эта функция используется для того, чтобы посмотреть, что лежит в стеке, не меняя самого стека.

Функция RETOP. Функция имеет два формата вызова. Первый:

RETOP(<имя> <список>)

Функция удаляет из стека с указанным именем верхний терм, затем заносит в стек все термы <списка> (слева направо). Второй формат вызова:

RETOP(<имя>() <список>)

В этом случае функция удаляет все термы из стека и заносит в него все термы <списка> (слева направо).

4.4. Функции арифметики.

Функция ADD. Формат вызова:

ADD(<число$_1$> <число$_2$>)

Аргументами могут быть как положительные, так и отрицательные числа. Результат вызова <число$_1$> + <число$_2$> .

Функция SUB. Формат вызова:

SUB(<число$_1$> <число$_2$>)

Результат вызова <число$_1$> − <число$_2$> .

Функция MULT. Формат вызова:

MULT(<число$_1$> <число$_2$>)

Результат вызова <число$_1$> ∗ <число$_2$> .

Функция DIV. Формат вызова:

DIV(<число$_1$> <число$_2$>)

Результат вызова – список двух чисел q и r, где q – частное от деления числа <число$_1$> на <число$_2$>, а r – остаток. Знак остатка r всегда совпадает со знаком делимого (<число$_1$>). Это условие однозначно определяет значение частного q.

Функция LESS. Формат вызова:

LESS(<число$_1$> <число$_2$>)

Результат вызова – атом TRUE, если <число$_1$> меньше, чем <число$_2$>, и атом FALSE в противном случае.

Функция RANDOM. Генератор случайных чисел. Формат вызова:

RANDOM(<число>)

Результат функции – случайное число в интервале от 0 до <число>∗256 .

К арифметическим функциям относятся также функции работы с рациональными числами. Описание этих функций содержится в файле infix.fl (см. п.3.3.2).

4.5. Функции задержки.

Функция QUOTE. Формат вызова:

```
QUOTE(<терм> <имя модуля>)
```

Если <терм> не аппликативный, то результатом будет <терм>. Если <терм> аппликативный, то функция QUOTE активизирует <терм> и пробует его выполнить; при этом предложения для отождествления выбираются только из модулей, загруженных позже указанного модуля. Т.е. функция QUOTE вычисляет <терм>, «начиная» с модуля, следующего за указанным. Результат функции – это результат вычисления терма.

Функция '. Функцию ' можно использовать только в текстах программ (нельзя вызвать непосредственно из диалога). Формат вызова

```
('<терм>)
```

Если этот вызов находится в модуле с именем <имя модуля>, то результатом ' будет

```
QUOTE(<терм> <имя модуля>)
```

Эта функция вычисляется на этапе компиляции. Знак ' является инфиксной операцией (см. п.3.3.1). Поэтому если программы будут компилироваться при загруженном файле infix, то формат вызова может быть таким:

```
'<терм>
```

Функция |. Эта функция написана на FLACe:

```
(| #x)  = #x;
```

Функция используется для задержки первого аргумента (см. п.2.3.2). Функция выполняется на этапе компиляции. Знак | является знаком инфиксной операции. Если программа компилируется при загруженном файле infix, то формат вызова функции, если ее аргументом является <терм>, может быть таким:

```
|<терм>
```

К средствам задержки относятся также конструкции HOLD и []. (см. п.2.3.2 и замечания в п.2.2).

§ 5. Ошибки

В этом пункте перечисляются коды ошибок и возможные причины их возникновения. Если при работе системы возникает ошибка, то начинает работать функция RUNEND, которая выдает в качестве результата сообщение об ошибке, включающее в себя код ошибки (см. п.4.2).

Код ошибки	Причины
0	Нет ошибки
1	Нет памяти (см.функцию RECLAIM в п.4.2).
2	Прерывание вызванное Ctrl C.
3	Переполнение стека. Происходит при попытке отождествить или напечатать слишком «глубокие» списки.
5	Нарушен баланс скобок.
6	Нет памяти под тексты атомов (см. функцию RECLAME в п. 4.2). В системе заведено слишком много атомов.
8	Файл не открыт.
9	Число открытых файлов превышает максимально возможное (10).
10	Попытка закрыть неоткрытый файл.
11	Синтаксическая ошибка.
12	Попытка описать встроенную функцию (Эта ошибка возникает при попытке описать встроенную функцию, реализованную не на FLACe.)

Список литературы

[1] В. Л. Кистлеров, “Принципы построения языка алгебраических вычислений FLAC”, Препринт, ИПУ АН СССР, Москва, 1987.

[2] Е. А. Гайдар, И. Н. Игнатович, В. Ф. Козадой, А. П. Немытых, В. А. Пинчук, С. В. Чмутов, “Реализация системы программирования FLAC”, Сборник трудов по функциональному языку программирования Рефал, **1**, Издательство Сборник, Переславль-Залесский, 2014(1988), 43–91.

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

[4] Ан. В. Климов, С. А. Романенко, “Система программирования РЕФАЛ-2 для ЕС ЭВМ. Описание входного языка”, Препринт, ИПМ: им. М.В. Келдыша АН СССР, Москва, 1987.

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

[6] Н. А. Чмутова, “Общее решение системы линейных уравнений над евклидовым кольцом”, Сборник трудов по функциональному языку программирования Рефал, **1**, Издательство Сборник, Переславль-Залесский, 2014(1988), 92–117.
Ниже дан список литературы, добавленный редактором сборника.

[7] 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.

[8] S. V. Chmutov, E. A. Gaydar, I. M. Ignatovich, V. F. Kozadoy, A. P. Nemytykh, V. A. Pinchuk, *The symbol analytic transformation language FLAC: sources, executable modules*, ftp://www.botik.ru/pub/local/scp/flac/flac386.zip, 1991.

[9] С. Д. Мешвелиани, *Система сивольных алгебраических вычислений CAC*, NN, 1993.

[10] С. Д. Мешвелиани, *Компъторная алгебраическая система вычислений Docon-FLAC*, ftp://www.botik.ru/pub/local/scp/flac/docon_flac.zip, 1994.

Е.А.Гайдар (E. A. Gaydar)
Переславль-Залесский

И.М.Игнатович (I. M. Ignatovich)
Переславль-Залесский

В.Ф.Козадой (V. F. Kozadoy)
Переславль-Залесский

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

В.А.Пинчук (V. A. Pinchuk)
Переславль-Залесский

С.В.Чмутов (S. V. Chmutov)
Переславль-Залесский

Е.А.Гайдар, И.М.Игнатович, В.Ф.Козадой,
А.П.Немытых, В.А.Пинчук, С.В.Чмутов

Реализация системы программирования FLAC

В статье описывается система программирования FLAC, реализованная в ИПС АН СССР. Библ. 12 наим.

Содержание

§ 1. Введение

Входным языком системы является FLAC (Functional Language for Algebraic Calculation), разработанный В. Л. Кистлеровым в 1986-1987 г. [см. статью [3] в настоящем сборнике]. Его доклады на семинарах в ИПС АН СССР явились первоисточником по языку.

Реализована система на IBM PC XT (MS DOS)[1] в г. Переславле-Залесском группой, руководимой Чмутовым С. В. (Гайдар Е. А., Игнатович И. М., Козадой В. Ф, Немытых А. П., Пинчук В. А.).

Многие идеи реализации были сообщены Кондратьевым Н. В. Авторы благодарны за бескорыстное внимание к их работе Абрамова С. М.

Программы, написанные на FLACe, транслируются на некоторый промежуточный язык. Далее мы будем называть его FAL (FLAC Assembler Language). Великолепное свойство FLACa – множество программ вкладывается в множество данных, вместе с ориентируемостью языка на символьные вычисления, позволяет естественно заниматься метадеятельностью. Иллюстрацией к чему является компилятор с FLACa на FAL, написанный на FLACe.

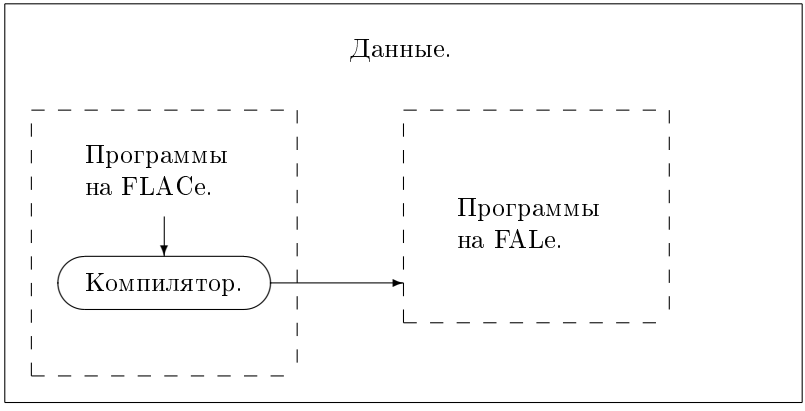


Рис. 1. Язык программирования FLAC.

Язык FAL интерпретируется системой. Система работает в режиме диалога. Структура реализации показана на рис. 2.

Первые версии диалога, администратора и компилятора транслировались «руками»; все последующие – предшествовавшей версией компилятора.

Выбранная структура реализации позволяет легко ее изменять и расширять.

Интерпретатор и встроенные функции написаны на языке «С». «С» позволяет надеяться, с одной стороны на достаточную эффективность, с другой – на переносимость системы.

Статью следует воспринимать как единое целое; мы рассматриваем сначала общие черты реализации, далее уточняем сказанное и вводим новые понятия,

[1] Позже реализация была перенесена на операционную систему UNIX и развивалась. В статье описывается состояние реализации, актуальное на момент написания статьи (1988 г.).

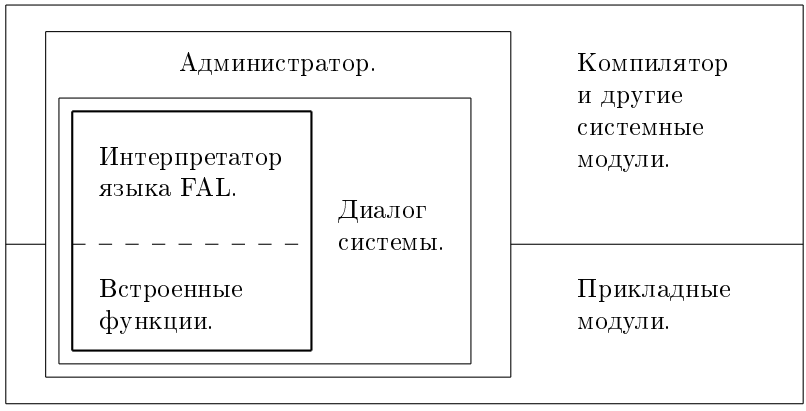


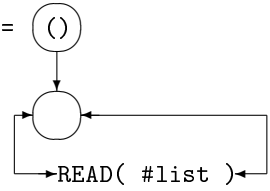
Рис. 2. Система компьютерной алгебры FLAC. (Обведённое жирной чертой реализовано на языке «C», оставшаяся часть – на FLACe.)

если это нужно для понимания последующего текста. Для программистов, работавших с РЕФАЛом, скажем, – данные представлены подвешенными списками со счетчиком ссылок на нижние «этажи» и обратим внимание на п.6.5. Механизм модульности описан в п.9, п.11.

§ 2. Выбранное представление данных эффективно использует память, позволяет быстро копировать, но ограничивает возможности перестановок

Опишем в терминах FLACa отображение `READ` из множества данных языка (`DATA`) в множество ориентированных графов:

```
READ( ) =
READ( (#list) ) =
```




```
READ( _numb ) = NEW_NUMB( _numb )
READ( &term ) = NEW_ATOM( &term )

READ( &term #list ) = READ( &term )←→READ( #list )
```

Отображения `NEW_ATOM` и `NEW_NUMB` задержим на некоторое время. После чего данные будем рисовать. (Нам удобно опускать имена задержанных функций.)

Несколько примеров:

1 `This is data of FLAC` $\overset{\text{READ}}{\longmapsto}$

2 (Это тоже данные FLACa) и `(F(&x #y) = 1234567890)`

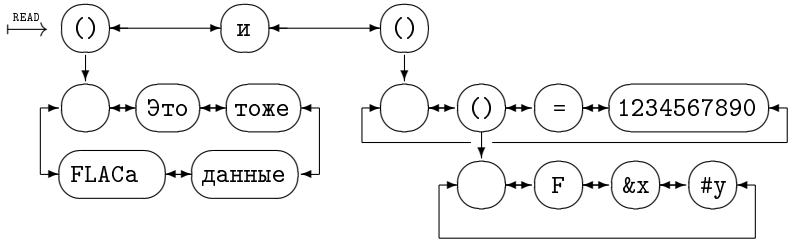


Рис. 3

Наиболее интересным является третий пример:

3
```
(Мы видим (что "два аппликативных терма" совпадают) !)
(2 + 3
 (Мы видим (что "два аппликативных терма" совпадают) !) )
```

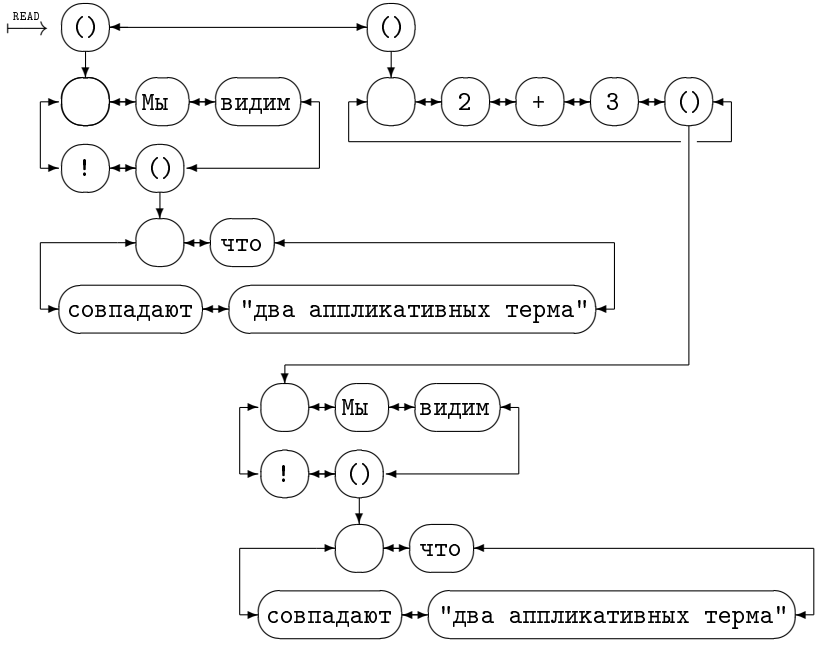


Рис. 4

Графы, обладающие подобным свойством, можно нарисовать более компактно – как показано на рис. 5, объединив совпадающие части.

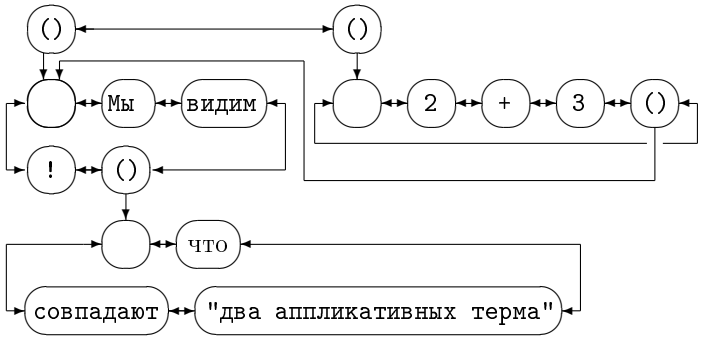


Рис. 5

Последние два рисунка представляют одно и то же данное. Таким образом, мы можем некоторые списки рисовать многими способами. (Существуют три рисунка, соответствующие последнему примеру.) Кроме того, указанное представление списков позволяет быстро строить новые, содержащие в себе уже имеющиеся аппликативные термы; для этого не нужно полностью повторять построение существующего терма, а лишь нарисовать представляющий его верхний узелок и направить из него стрелку на служебный (пустой) узелок терма, копию которого мы делаем.

FLAC-машина [см. п.4] по существу представляет собой машину «переписывания» описанных нами графов; построения новых, используя уже ранее построенные.

Новые графы мы сможем строить тем быстрее, чем больше используем существующие. Один из таких приемов указан нами. Его важным свойством является то, что не происходит разрушения одних рисунков при построении других. Возникают интересные вопросы: можно ли построить граф, соответствующий списку

```
(Мы видим !) (2 + 3 (Мы видим !))
```

из рисунка 5, выкинув вершину (()), или граф, соответствующий списку:

```
"два аппликативных терма" совпадают
```

– просто выдернув нужную часть картинки?

Очевидный ответ – да, если рисунок нам больше не нужен – становится неприемлемым, если картинка очень большая и сложная, в связи с чем нужно потратить много времени, чтобы понять, нужна она или нет.

Возникающую проблему решим, запоминая свойства графа в процессе его построения или при его анализе для других целей, где уже нельзя обойтись без полного просмотра. Конечно, здесь требуется уточнение «нужности» графа. Пока же мы, в качестве одного из критериев «нужности», будем при построении в служебной вершине ( ) аппликативного терма записывать число стрелок, указывающих на эту вершину.

На рисунке 4 во все пустые вершины нужно поставить 1. На рисунке 5 – в первую 2, во вторую 1, в третью 1 (слева направо, сверху вниз).

§ 3. Алгоритм отождествления

Рассмотрим `DATA(FLAC)`. Любой элемент этого множества представляет собой список (конечную или пустую последовательность) термов, поэтому будем обозначать его $LIST$. Множество термов обозначим $TERM$, множество чисел – $NUMB$.

$$NUMB \subset TERM \subset LIST$$

Выделим в каждом $f \in$ `DATA(FLAC)` атомы, интерпретирующиеся в программах на FLACe как переменные (t переменных типа `"&"`, l – типа `"#"` и n – типа `"_"`). По f можно построить функцию

$$f^* : TERM^t \times LIST^l \times NUMB^n \to \texttt{DATA(FLAC)}$$

от $t + l + n$ переменных (сумма может быть нулевой).

Пусть $f, r \in$ `DATA(FLAC)`, будем говорить, что r отождествляется с f, если существует решение уравнения

$$f^*(\bar{t}, \bar{l}, \bar{n}) = r \quad (\bar{t} \in TERM^t, \bar{l} \in LIST^l, \bar{n} \in NUMB^n). \qquad \textbf{[I]}$$

Далее функцию f^* будем называть типовым выражением, а r – объектным.

Существует ли алгоритм, решающий такие уравнения, либо говорящий, что решений нет? Всегда ли единственно решение, когда оно существует?

Ответы станут очевидными, если мы опять попытаемся изобразить множество значений функции f^* в виде графа. Нарисуем граф, соответствующий типовому выражению [см. рис. 6]

```
( #x ) a &y ( b _z  2 ) &y
```

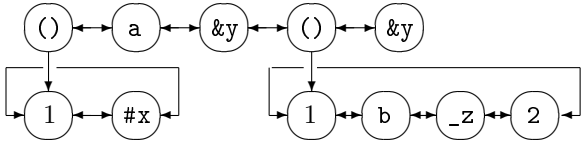


Рис. 6

Множество графов – значений соответствующей функции получается из рисунка 6, если вместо вершин графа с именами переменных подставлять возможные значения этих переменных, причем вместо вершин с одинаковыми именами – одинаковые значения.

Теперь понятно, что решение уравнения **[I]** существует тогда и только тогда, когда граф, отвечающий r, можно получить из f указанным способом. То есть, их графы совпадают с точностью до «переменных вершин» графа f. После «наложения» r на f подграфы в r, попавшие в эти вершины, должны быть определенного типа; они и являются решением нашего уравнения. (Здесь мы не различаем рисунки, представляющие одни и те же данные.)

Утверждения единственности в общем случае нет, но если ограничиться только типовыми выражениями, отвечающими левым частям предложений FLACa, то более одного решения быть не может. Об этом говорит простой алгоритм отождествления (решения уравнения).

Нарисуем «таблицу переменных» и запишем в нее имена переменных нашего типового выражения.

Рассмотрим рисунки, представляющие f и r. На крайние вершины верхних «этажей» графов установим стрелки (левые и правые, соответственно) [см. рис. 7,8,9]. Будем параллельно двигать левые стрелки вправо по вершинам, проверяя совпадение вершин, если стрелка в f не находится на переменной, иначе проверяется, может ли переменная принять значение терма, представителем которого является вершина в r, указанная там левой стрелкой. Если проверка окажется неудачной, то уравнение не имеет решений. В случае успеха укажем стрелочкой от имени переменной в таблице ее значение. Решение отсутствует также тогда, когда один «этаж» окажется короче другого. Движение закончим, когда попадем на переменную типа список или просмотрим весь «этаж». Остановились по первому условию, – аналогичную процедуру проделаем с правыми стрелками, двигая их влево. Ограничение – на одном скобочном уровне может быть не более одной переменной типа список – гарантирует, что мы остановимся, когда правая стрелка в f совпадет с левой. Между левой и правой стрелкой в объектном выражении r окажется значение переменной типа `"#"`, если она присутствует. Нужно еще отметить: если мы получили значение переменной (любого типа), которая ранее уже определилась (более одного вхождения одной и той же переменной), то проверяем совпадение ее значений, и если они не совпадают, объявляем о неудаче.

Данную процедуру необходимо проделать для каждого «этажа» графа f, считая вершину, первую справа от служебной, началом «этажа», первую слева – концом.

Граф f `:= ( #x ) a &y ( b _z 2 ) &y` :

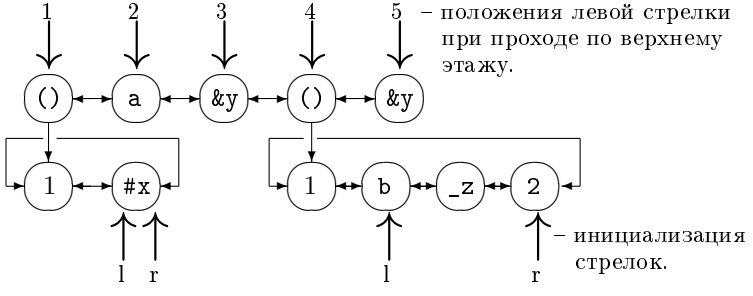


Рис. 7

Отождествление с r = `(1 2 3) a (a ()) (b 3 2) (a ())` показано на рисунке 8. Над (под) стрелками указаны номера шагов алгоритма. Значения переменных:

`&y = ( a () ), #x = 1 2 3, _z = 3`

Теперь отождествим f с r = `() a c ( b a 2 ) c` [см. рис. 9].

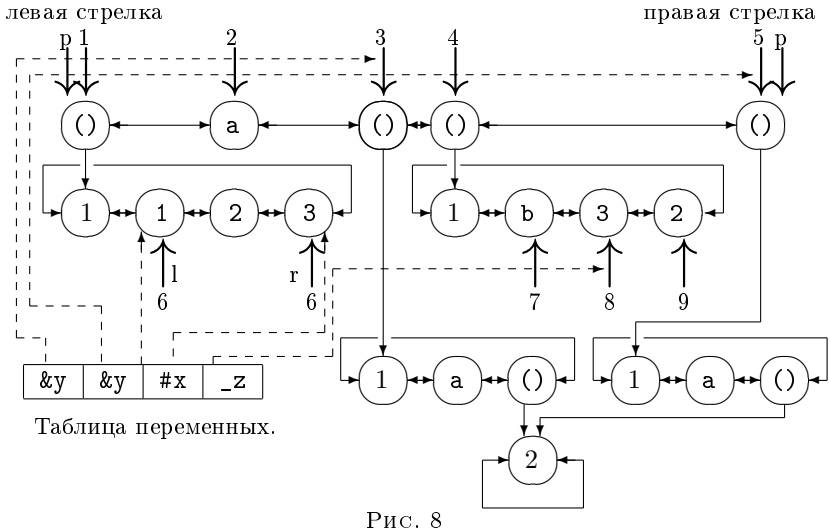

левая стрелка
правая стрелка
p 1
2
3
4
5 p
()
a
()
()
()
1
1
2
3
1
b
3
2
l
6
r
6
7
8
9
&y
&y
#x
_z
1
a
()
1
a
()
2
Таблица переменных.

Рис. 8

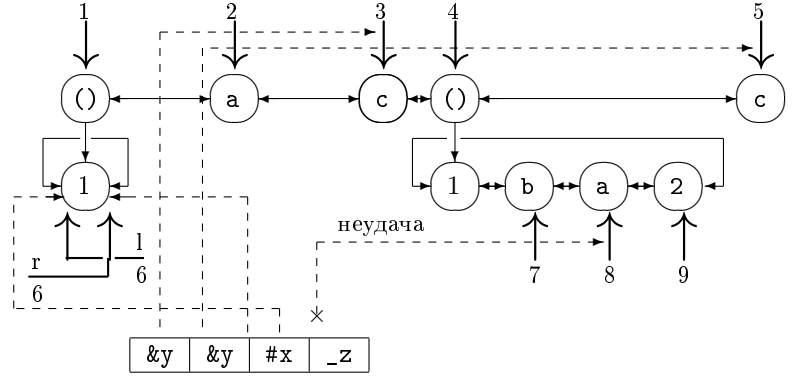

1
2
3
4
5
()
a
c
()
c
1
1
b
a
2
неудача
r
6
l
6
7
8
9
&y
&y
#x
_z
Таблица переменных.

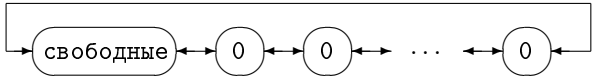
Рис. 9

§ 4. FLAC-машина

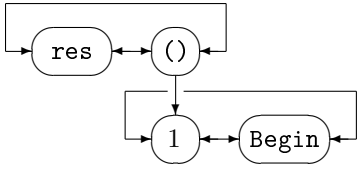
Опишем абстрактную FLAC-машину – машину переписывания рассматриваемых нами графов.

В выключенном состоянии она представляет собой несколько графов:

а) список свободных вершин –

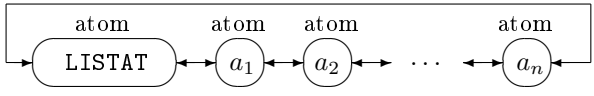


б) граф вызовов функций и хранения всех результатов выглядит так –



Это – окно, через которое можно наблюдать за работой машины.

в) таблица атомов –



Текущий список атомов.

Кроме этого, машина имеет графы-программы (в начальный момент – диалог системы), множество которых можно изменять в процессе работы машины [см. п.9].

После включения первой вычисляется функция «`Begin`». Далее функции вычисляются из вершины стека, сами изменяют стек. (Как он устроен, мы расскажем ниже.)

Функция вычисляется по соответствующему ей графу-программе, который FLAC-машина находит по имени функции. Выбор предложения функции производится в соответствии с семантикой языка [см. статью [2]].

Вычисление значения функции от конкретных аргументов есть:

1) Поиск нужного предложения (отождествление).

2) Построение графа, являющегося значением типового выражения правой части найденного предложения в точке, указанной алгоритмом отождествления в таблице переменных. Материалом для построения являются свободные вершины.

3) Замена вычисляемого аппликативного терма в графе вызова построенным графом.

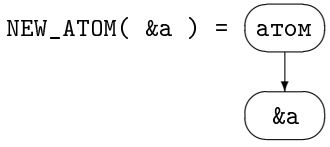
4) Корректировка стека активных функций; вызовы функций, стоящие в правой части предложения, вычисляются слева направо, и к моменту вычисления функции ее аргументы должны быть определены.

Третий шаг пропускается, если: а) в стеке оказался граф, представляющий аппликативный терм, имя которого не атом; б) в машине нет программы для функции с указанным именем; в) ни одно из уравнений $f_i^*(\overline{x}) = r$

(r – вычисляемый терм, а $\{f_i\}$ – все левые части предложений данной функции) не имеет решений. Это явление называется задержкой [см. статью [3]].

§ 5. Представление атомов

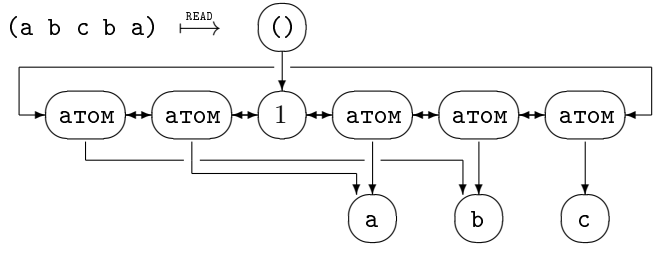
Для того чтобы рассказать о таблице атомов, активизируем функцию `NEW_ATOM` [см. п.2].



Множество `DATA(FLAC)` бесконечно, в то же время и список свободных вершин ограничен, и не все данные нужны для вычисления конкретных функций. Отображение `READ` «строит» графы по мере необходимости [как отмечалось в п.2, используя построенные].

Нижняя вершина графа, представляющего атом, будет у нас уникальна; она строится только один раз, когда граф атома с указанным именем еще не существует; повторное появление атома приводит лишь к построению верхней вершины, `READ` связывает ее стрелкой с уже имеющейся внутренностью атома (нижней вершиной).

Пример:



Снова возникает проблема эффективности – когда графы большие, много времени понадобится для их просмотра, чтобы ответить на вопрос: построен ли уже граф, отвечающий данному атому.

Таблица атомов решает эту проблему; при первом появлении атома в нее добавляется вершина, связанная с внутренностью. Теперь достаточно просмотреть только граф-таблицу.

Указанное представление позволяет элегантно организовать механизм модульности [см. п.9, п.11], быстро находить программы, соответствующие функциям [см. п.13].

§ 6. Язык FAL

6.1. Операторы отождествления.

Попробуем в алгоритме отождествления выделить некоторое множество элементарных шагов EQ такое, что для любого $f \in$ `DATA(FLAC)` существует конечная последовательность $e_i^f \in EQ$, «решающая» уравнение $f^*(\overline{x}) = y$ для всех $y \in$ `DATA(FLAC)`. (Под решением мы понимаем и сообщение о пустом множестве решений.)

e_i^f – предикаты. В случае успеха, они будут фиксировать решение в таблице переменных.

Несколько замечаний:

1 Будем понимать под графом, начинающимся с вершины (с корнем в вершине), граф, состоящий из самой вершины и вершин, доступных из данной при движении по направлениям ребер графа; первый шаг разрешается делать только по ребру, идущему на нижний «этаж» (если такое ребро существует).

2 Отождествив очередной «этаж», мы не забудем более низкие; проходя через «лестницы» (`()`), оставляем о них информацию в таблице переменных, рисуя стрелки к соответствующему «этажу». Теперь просто найти «этажи», где мы еще не были.

`terml(` вершина графа `)`
{

Просмотрели «этаж» графа объектного выражения, на котором находятся стрелки?

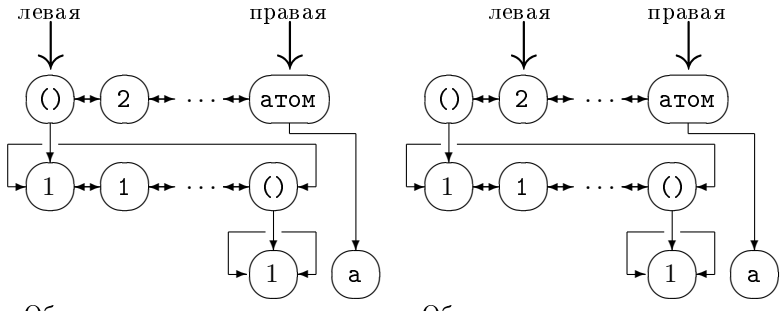
Равны ли графы, начинающиеся с вершины-аргумента и с вершины, на которую указывает левая стрелка?

Передвинуть левую стрелку вправо.
}

Напомним: мы не различаем графы, представляющие одни и те же данные [см. рис. 10].

`termr(` вершина графа `)`
{

Предикат аналогичен `terml`, но рассматривается правая стрелка.
}



Объектное выражение перед работой.

Объектное выражение после работы.

Работа предиката `terml( (1 ()) )`.

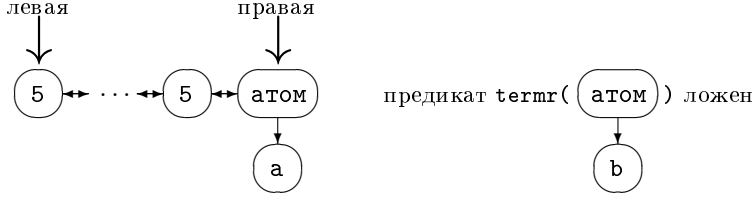


Рис. 10

Значение переменной типа терм запоминает следующий оператор [рис. 11].

```
vterml()   /* vtermr() */
{
```

Просмотрели «этаж» графа объектного выражения, на котором находятся стрелки?

Запомнить ячейку в таблице переменных, на которую показывает левая стрелка.

Увеличить номер (`endtab`) первой свободной ячейки в таблице переменных.

Передвинуть левую стрелку вправо.

```
}
```

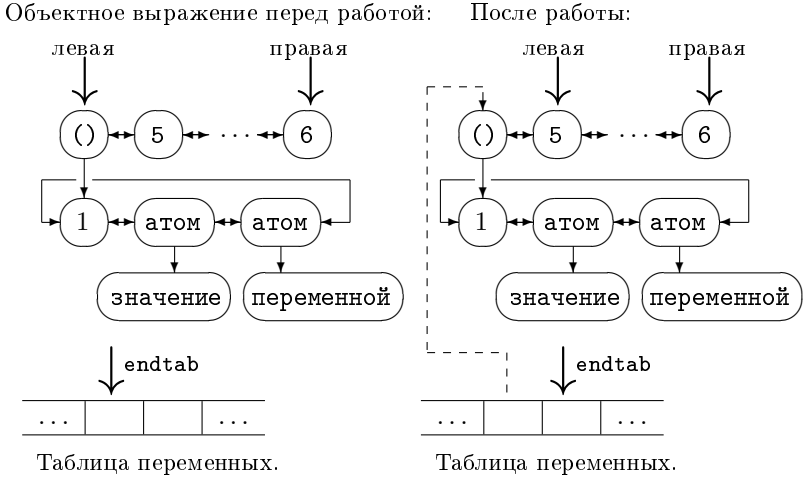


Рис. 11. Работа `vterml()`.

Предикат: число?

```
vnumbl()   /* vnumbr() */
{
```

Просмотрели «этаж» графа объектного выражения, на котором находятся стрелки?

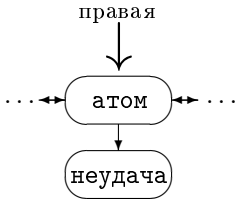
Левая стрелка стоит на числе?

Запомнить ячейку в таблице переменных, на которую показывает левая стрелка.

Увеличить `endtab`.

Передвинуть левую стрелку вправо.

```
}
```



предикат `vnumbr()` ложен.

Рис. 12

Сравнение переменных делает предикат

```
equal(два аргумента – номера ячеек в таблице переменных, содер-
       жащих стрелки к значениям сравниваемых переменных
      )
{
  Равны графы, указанные аргументами (как данные)?
}
```

Важно: атомы сравниваются по физическому совпадению их внутренностей; понятно, что наибольшее время займет работа этого предиката, когда его значение – истина, поэтому сравнение аппликативных термов тоже нужно начинать с проверки физического совпадения служебных вершин (при удаче термы совпадают), если графы большие, данный шаг может оказаться критическим.

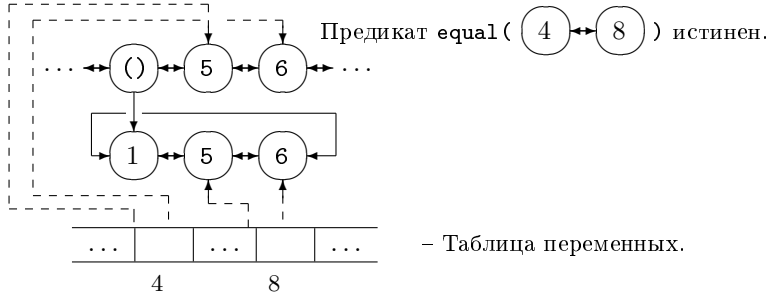


Рис. 13

Оператор `list()` фиксирует переменную типа список.

```
list()
{
```
Заполнить две ячейки в таблице переменных, указав в первой вершину с левой стрелкой, во второй – с правой.
```
}
```

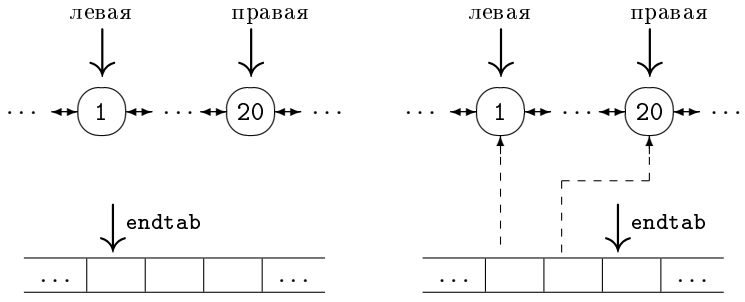


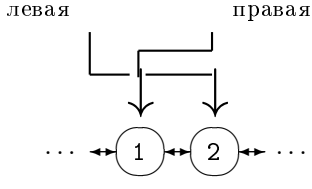
Таблица переменных. Перед работой.

Таблица переменных. После работы.

Рис. 14. Работа `list()`.

Мы просмотрели весь «этаж» графа объектного выражения?

```
empty()
{
```
Левая стрелка является соседней справа по отношению к правой стрелке?
```
}
```



Предикат `empty()` истинен.

Рис. 15

```
brackl()   /* brackr() */
{
```

Этот предикат запоминает нижний «этаж».

Просмотрели очередной этаж?

В таблицу переменных заносятся вершины, которыми будут инициализироваться стрелки для отождествления «этажа»; спуск к нему показывает левая стрелка.

Увеличивается `endtab`.

Левая стрелка передвигается вправо.

```
}
```

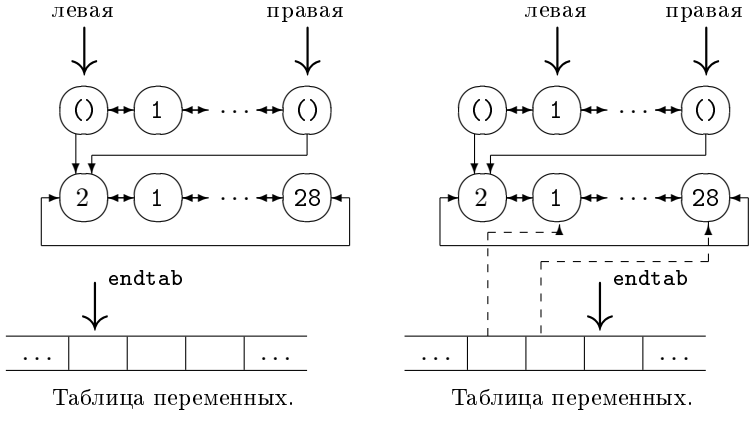


Рис. 16. Предикат `brackl()`.

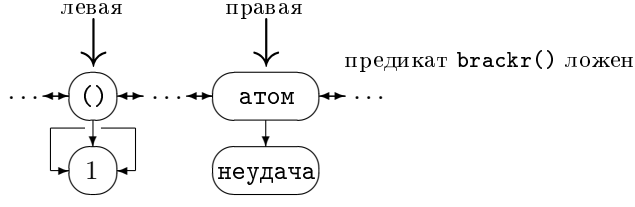


Рис. 17

Завершая описание предикатов отождествления, еще раз обратим внимание на то, что пройти нужно все «этажи» типового выражения и соответствующие им в объектном.

На нижний «этаж» спустимся с помощью предиката

`set(` номер ячейки в таблице переменных, содержащей левую стрелку нужного «этажа» `)`

{

Инициализируются левая и правая стрелки.

Всегда истинен.

}

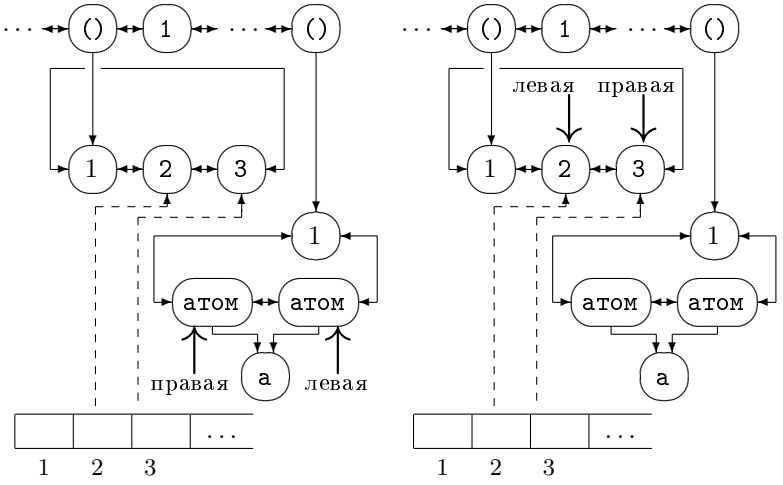


Рис. 18

6.2. Операторы построения.

Чтобы FLAC-машина нарисовала `READ(`f`)` (где $f \in$ `DATA(FLAC)`), ей, как и в случае отождествления, также нужно указать некоторую конечную последовательность операторов (шагов построения) r_n^f. (Их множество обозначим $\mathbb{RE}$.) $r \in \mathbb{RE}$ будут строить `READ(`f`)` в списке свободных вершин. Результат построения находится между отмеченной вершиной [п.4] и последней занятой (включая ее), которую мы обозначаем `endptr`.

`rterm(` вершина существующего графа `)`

{

Рисует копию графа, начинающегося с вершины-аргумента.

Содержимое аргумента заносится в первую свободную вершину.

Если граф не тривиален, то рисуется от новой вершины стрелка к первой вершине нижнего «этажа».

В случае аппликативного терма число в служебной вершине увеличивается на 1. (Тем самым служебная вершина фиксирует, сколько стрелок ведет в нее; сколько раз «этаж» достижим с более верхних.)
}

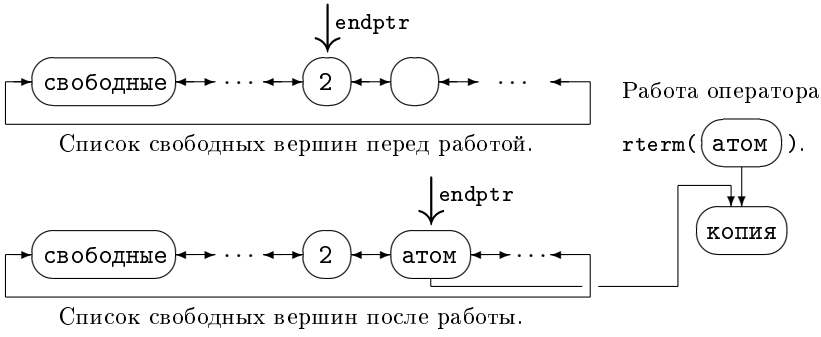


Рис. 19

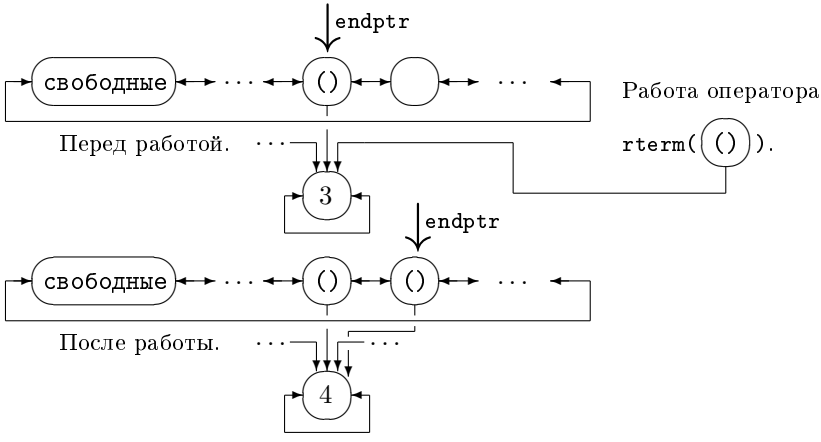


Рис. 20

По определению отображения READ, паре соответствующих друг другу скобок сопоставляется «этаж» графа, причем то, что одна пара скобок находится внутри другой, означает, что с «этажа» первой пары имеется «лестница» на «этаж» второй.

Алгоритм построения в списке свободных вершин конструирует данное f на верхнем уровне, отмечая «лестницы» на будущие «этажи» стрелками (от построенной к предыдущей), и помнит последнюю недостроенную «лестницу» – endlbr [см. рисунок 21]. Только когда «этаж» готов, он опускается вниз, а по стрелке (о которой позаботились) мы можем правильно установить endlbr. Более использованная стрелка не нужна.

Необходимо заметить, что из чисел могут выходить только две стрелки, а из атомов и скобок – три [см. п.13].

`lbr()`
{

Первую свободную вершину заполнить скобками (()) , указать из нее стрелкой на `endlbr`, запомнить как `endlbr`.

Вторая свободная вершина – служебная будущего «этажа»; как раз сейчас мы его строим из свободных элементов, и на данном этапе на эту вершину будет смотреть ровно одна стрелка, которую нарисуем, когда переместим «этаж» вниз. Следовательно, в служебную вершину нужно записать 1.
}

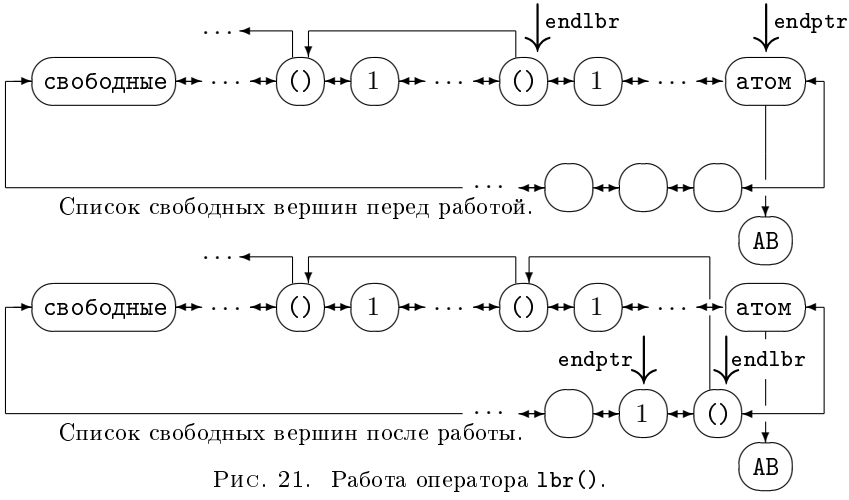


Рис. 21. Работа оператора `lbr()`.

`rbr()`
{

Опускаем на нижний «этаж» список от `endlbr` до `endptr`, включая последнюю.

`endlbr` – «лестница».

Вершина, на которую показывает стрелка из `endlbr`, становится `endlbr`. Стрелка стирается. Бывшая `endlbr` становится `endptr`.

Образовавшаяся «дыра» между занятыми и свободными вершинами устраняется.
}

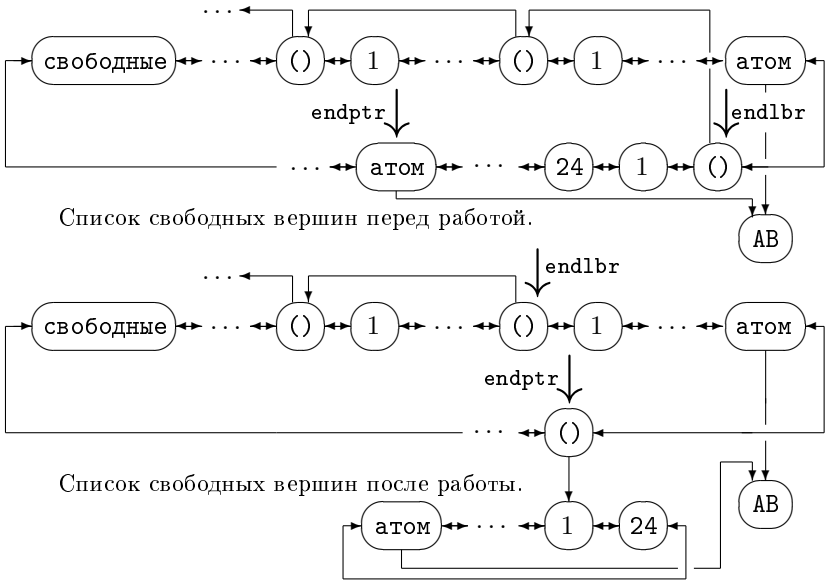


Рис. 22. Работа оператора `rbr()`.

При следующем `rbr()`, построенный «этаж» опустится еще ниже и т. д.

6.3. Оператор копирования.

Уже можно строить любое полностью определенное данное; действительно, оно состоит из простых термов (`rterm`), а также из левых (`lbr`) и правых (`rbr`) скобок.

В процессе работы FLAC-машины нужно уметь вычислять значение функции

$$f^*(\overline{x}) \quad (f \in \texttt{DATA(FLAC)},\ \overline{x} \in TERM^t \times LIST^l \times NUMB^n)$$

в произвольной точке из множества определения [см. п.3], где f – правая часть того предложения языка FLAC – $g = f$, на котором удачно завершилось отождествление.

Переменные функции f^* определяются при решении уравнения $g^*(\overline{y}) = w$ (где w – аппликативный терм, попавший в вершину стека) и фиксируются предикатами отождествления в таблице переменных.

Алгоритм отождествления фиксирован, и типовому выражению g соответствует конкретная последовательность $\{e_n^g\}$.

Пусть V^g множество переменных из g, тогда из сказанного следует, что существует вычислимая функция $F^g : V^g \to \mathbb{N}$ ($\mathbb{N}$ – множество натуральных чисел), устанавливающая соответствие между переменными и номерами ячеек в таблице переменных. Эти номера мы и укажем оператору

`rcopy(` номер в таблице переменных `)`
`{`

В граф свободных вершин копируется список, на начало которого показывает стрелка, выходящая из ячейки таблицы с данным номером, на конец списка показывает стрелка из соседней справа ячейки таблицы переменных.

Пустые списки не копируются.

Для того, чтобы оператор был пригоден для переменных типа терм и типа число, нужно для них в таблице отводить по две ячейки, дублируя стрелки (см. описание предикатов `vterml, vtermr, vnumbl, vnumbr`).

Копируется только верхний «этаж» указанного списка; к нижним увеличивается число стрелок – «лестниц».
}

Эффективное копирование, по скорости и по числу забираемых вершин из списка свободных, обеспечило выбранное представление данных.

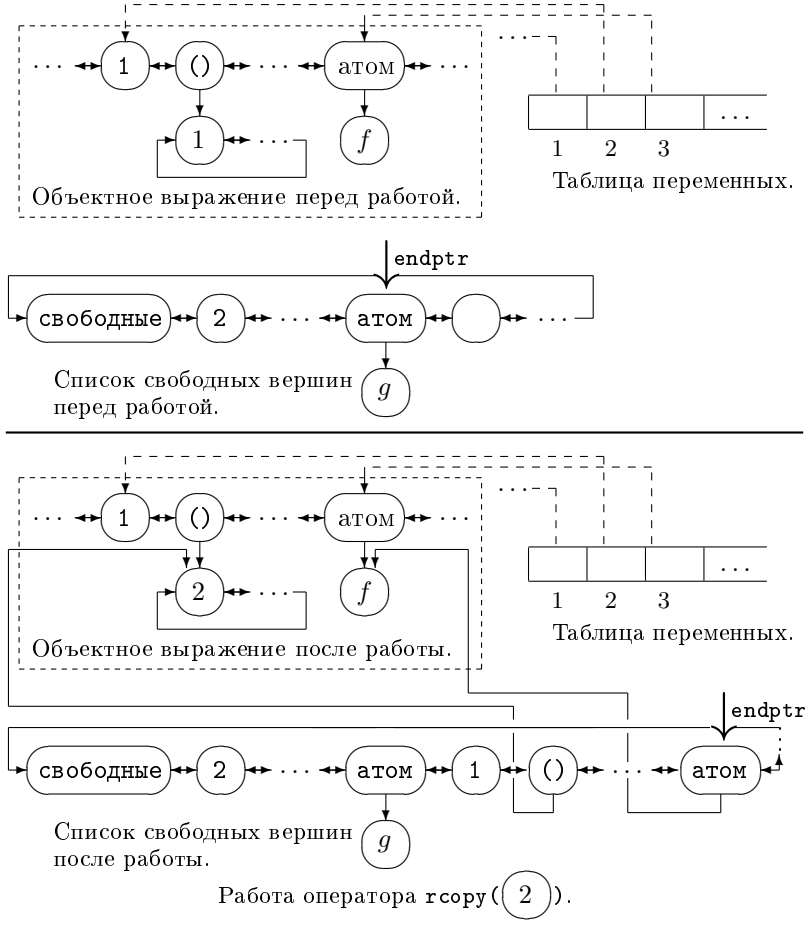


Рис. 23

6.4. Стек активных функций. Программа и данные.

Часть термов, входящих в правую часть предложения, являются данными, часть – программой.

Напомним, что вычисляются все аппликативные термы, кроме:

а) ();
б) термов с числовым именем;
в) термов с именем – переменная типа число;
г) явно задержанных аппликативных термов [см. статью [2]];
д) аппликативных термов с именем – задержанный аппликативный терм.

Порядок вычисления функций определяет, по правой части предложения, простой алгоритм: аппликативный терм нужно положить в очередь на выполнение, если строится его правая скобка; каждая функция, вычислившись, вставляет вместо себя в стек активных функций очередь, образовавшуюся из вызовов в правой части.

Утверждение 1. На первый «этаж» любого аппликативного терма, находящегося в стеке активных функций, «смотрит» ровно одна стрелка. (Под первым мы понимаем «этаж», висящий непосредственно под корнем (()) данного терма.)

Действительно, мы «бросаем» терм в стек, когда завершаем явное построение его первого «этажа». И никогда не копируем вершину, с которой начинается соответствующий граф, пока терм не перейдет в множество данных (см. описание операторов построения и копирования); копируются только аргументы функций.

В служебных вершинах аппликативных термов из стека, информация – «1» лишняя; она эквивалентна принадлежности стеку.

Не так уж много у нас свободных вершин, чтобы не использовать этот удивительный факт; переопределим служебную вершину (на время ожидания термом очереди своего вычисления), «записав» в нее информацию о том, где хранится аппликативный терм, который нужно вычислять за данным, мы организуем стек активных функций.

После вычисления функции, делаем обратное переопределение и восстанавливаем «1».

Увеличивать стек будет оператор

```
act()
{
```

Дно стека активных функций правой части обозначим `lastact`.

Положить аппликативный терм с корнем в `endptr` в стек: из служебной вершины терма `lastact` рисуется стрелка к вершине рассматриваемого терма; аппликативный терм – `endptr` объявляется последним активным (`lastact`).

Перед работой операторов построения `lastact` – служебный аппликативный терм `actptr`, хранящийся во FLAC-машине.

Аппликативный терм, найденный из служебной вершины терма `actptr`, должен вычисляться первым после работы вычисляемой функции; это вершина стека.

```
}
```

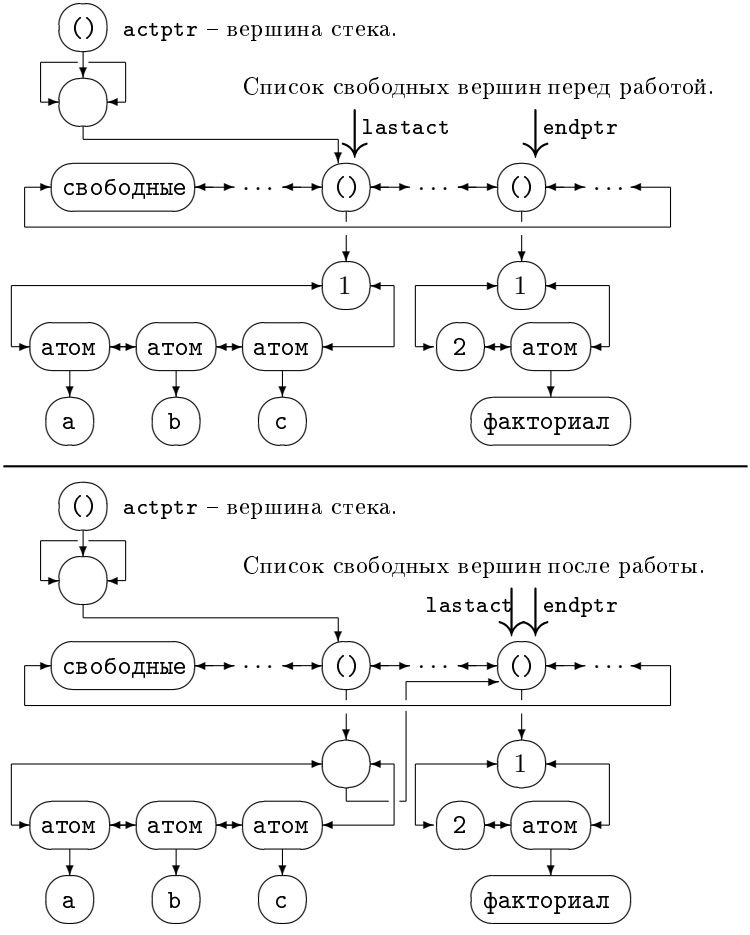


Рис. 24. Работа оператора `act()`.

Механизм задержки переводит аппликативные термы-программы в данные; оператор `act()` позволяет осуществить обратный переход также средствами самого языка FLAC.

Тождественная функция `eval` имеет смысл: она все аппликативные термы из своего аргумента заносит в стек; данные перешли в программу.

```
eval( ( #x ) #y ) = ( eval( #x ) )  eval( #y );
eval( &x #y )     = &x  eval( #y );
eval( )           = ;
```

6.5. Перемещение кусков из существующих графов в строящиеся требует дополнительных действий при отождествлении.

Проблема быстрого построения графов обсуждалась в п.2.

Где могут оказаться ненужные данные, подграфы которых можно использовать для построения правых частей предложений? Нам нужны аргументы вычисляющейся в данный момент функции; именно они определяют правую часть соответствующего предложения.

Значением переменной типа список может оказаться граф, верхний «этаж» которого большой (например, состоит из 1000 вершин), хотя граф при построении копируется только по верхнему «этажу», нас и это не устраивает – 1000 раз будут строиться вершины; потребуется много времени и много вершин.

Первый «этаж» вычисляющегося терма нужен ровно один раз (см. п.6.4) – только нашему аппликативному терму. Куски из него мы и будем переставлять в строящуюся правую часть.

Переменная типа список типового выражения из левой части может встречаться только один раз на каждом «этаже» (см. [2]), в правой части количество ее вхождений неограничено. Перестановка графа – значения переменной – происходит лишь однажды, иначе будут «вырезаться» части строящегося графа.

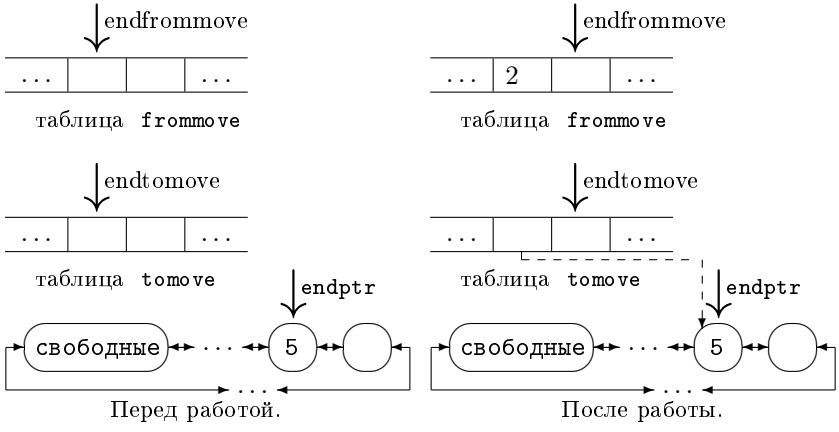
(Пример: `f( #x ) = #x a #x`. Стрелки указывают перестановки.)

`rmoveh(` вершина-число `)`
`{`

Оператор запоминает в таблице `frommove` номер – аргумент, указывающий ячейку в таблице переменных (стрелка из этой ячейки указывает на начало списка, который мы хотим переставить).

Рисуется стрелка из таблицы `tomove` на вершину в списке свободных. После нее мы хотим вставить наш список. (Так как построение идет последовательно, то стрелка рисуется к `endptr`. О самих перестановках см. далее.)

Пустые списки не запоминаются.
`}`



Работа оператора `rmoveh(` 2 `)`.

Рис. 25

Программа:

```
f( &x )         = g( &x a &x );
g( ( #x ) #z ) = #x #z;
```

показывает, что на служебную вершину первого аргумента функции `g` смотрят не менее двух стрелок; нижний «этаж» нужен и в графе – значении переменной

`#z`. Попытка переместить `#x` в правую часть при вызове `f( (1 2 3) )` приведет к результату `1 2 3 a ()`, вместо правильного `1 2 3 a (1 2 3)`.

Единица в служебной вершине «этажа», часть которого после отождествления стала значением переменной типа список, тоже недостаточная информация для вырезания:

```
f( &x )              = g( &x &x );
g( ( &x (#x)) #z ) = #x #z;
```

Если вызвать функцию `f( (1 (2 3)) )`, то вместо правильного результата `2 3 (1 (2 3))`, с перестановкой мы получим `2 3 (1 ())`.

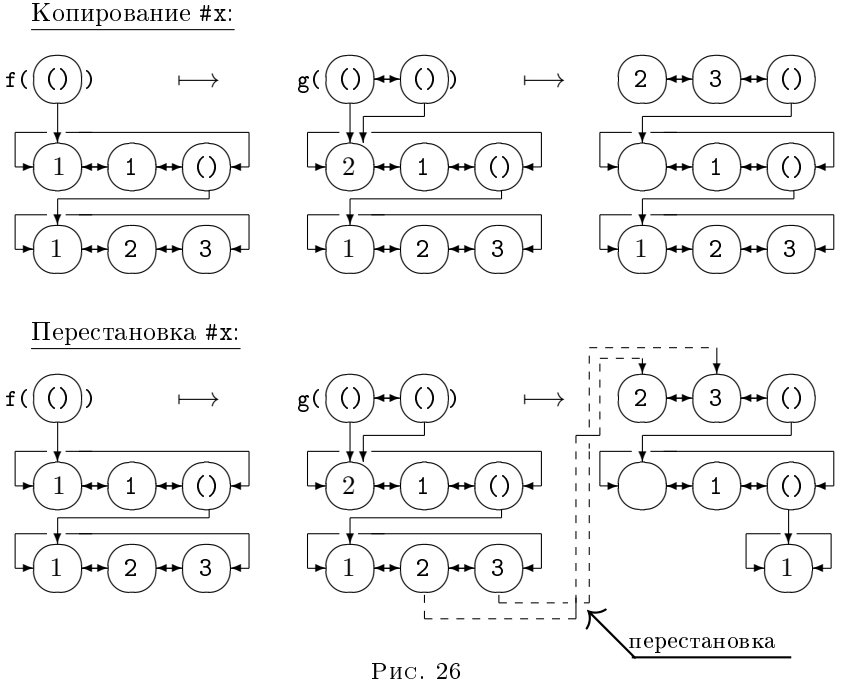


Рис. 26

Алгоритм отождествления спускается по «этажам», просмотрев полностью очередной. Он и поможет нам оставить достаточную информацию о возможности «вырезания» с каждого «этажа» в таблице «вырезаний»:

1) подсписки первого «этажа» можно использовать,
`cut = TRUE`;

2) операторы, проверяющие наличие под стрелками отождествления нижних «этажей», знают о возможности вырезания из них (`cut` & (в служебной вершине «этажа» стоит 1?)); последнюю информацию и необходимо запомнить в таблице «вырезаний» (внесем соответствующие дополнения в описание `brackl, brackr`);

3) переход на следующий «этаж» осуществляет `set`, он должен правильно установить `cut`; добавим предикату еще один аргумент – число (его вызов может выглядеть, например, так `set(` (4)↔(8) `)`, это число – номер ячейки в таблице «вырезаний», где `brackl` [см. п.3]) сообщает о том, существует ли еще путь по стрелкам к нашему «этажу», кроме пройденного нами; содержимое указанной ячейки запоминается в `cut`.

Замечательно: сейчас мы можем написать оператор, «вырезающий» или копирующий нижние «этажи» аргументов функций в зависимости от таблицы «вырезаний».

`rmove(` число, число `)`
`{`

Второй аргумент – номер ячейки в таблице переменных, стрелка из которой ведет на начало списка нужного для переноса; первый – номер ячейки в таблице «вырезаний», где лежит сообщение – можно ли этот список перемещать в другой граф.

Если ответ положительный, работает оператор `rmoveh` с первым аргументом, иначе с тем же аргументом работает `rcopy`.

Пустые списки не запоминаются и не копируются.
`}`

`rmoveh` и `rmove` только запоминают таблицу перемещений, не осуществляя самих перестановок. Нужен специальный оператор `subst` [см. рис. 27], переносящий все графы, указанные в таблице `frommove` (см. описание `rmoveh`).

`subst()`
`{`

Для каждой заполненной ячейки таблицы frommove проделать:

На граф – список для перестановки указывает стрелка из ячейки таблицы переменных с номером, хранящимся в очередной ячейке таблицы `frommove`.

Конец списка находим по соседней справа ячейке таблицы переменных.

Список вставляется за вершиной, которая отмечена стрелкой из соответствующей ячейки таблицы `tomove`.

Образовавшаяся дырка «зашивается».

Оператор всегда присутствует в последовательности $\{r_i\}$ [см. п.6.1] один раз – последним элементом.
`}`

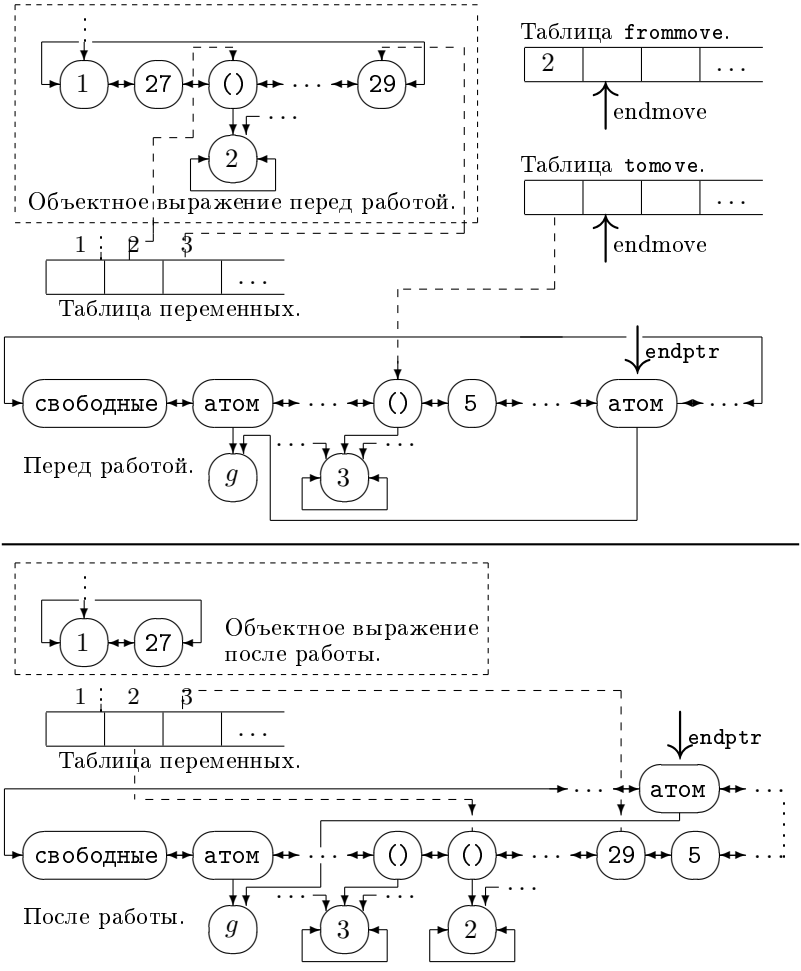


Рис. 27. Работа оператора `subst()`.

6.6. Оператор замены.

Шаги 3 и 4 FLAC-машины [см. п.4] делает оператор `end()`.

```
end()
{
```

Заменяет в графе вызова вычисляющийся аппликативный терм построенным списком, находящимся среди свободных вершин между отмеченной и `endptr`, включая последнюю.

Корректирует стек согласно семантике:

Пусть стрелка `callfunc` отмечает вершину стека. Из служебной вершины аппликативного терма `lastact` проводится стрелка к терму, следующему в стеке за `callfunc`.

`callfunc` устанавливается на вершину, показанную стрелкой из служебной ячейки терма `actptr`.

Стрелка `lastact` устанавливается на `actptr`.

Ненужный граф (бывший ранее `callfunc`) вставляется после отмеченной [см. п.4] в списке свободных, на нее же устанавливается стрелка `endptr`. В служебную ячейку графа заносится «забытая» ранее (в `act()`) «1».

Если результат – пустой список, то выполняется только последний шаг.

Все образовавшиеся «дыры» в графах «зашиваются».
}

Пример работы оператора `end()` показан на рисунке 28.

6.7. Другие операторы.

Чем меньше элементов в последовательностях $\{e_n\}$ и $\{r_m\}$ [см. п.6.1, п.6.2], тем быстрее FLAC-машина отождествляет (строит). Для часто встречающихся комбинаций естественно ввести новые операторы; их работа будет заменять работу указанного набора. Например, термы с известными именами или именами-переменными.

Соответствующие операторы:

```
raterm( имя аппликативного терма )
          = lbr() rterm( имя аппликативного терма );

rvaterm( номер ячейки в таблице переменных,
         где хранится имя аппликативного терма
       )  = lbr() rcopy( указанный аргумент );

rvamvh( номер ячейки в таблице переменных,
        где хранится имя аппликативного терма
      )  = lbr() rmoveh( указанный аргумент );

rvamove( число1, число2 ) = lbr() rmove( число1, число2 );

rbrack() = rbr() act();

endop() = sbst() end();
```

Операторы отождествления аппликативного терма и его имени расположены в последовательности $\{e_n\}$ не подряд, но имя терма легко найти, если стрелка (левая, правая) стоит на корне данного графа. Отождествив терм-имя, мы в качестве начала нижнего этажа запомним следующую за именем вершину. (Оператор `aterml( простой терм – имя аппликативного терма )`, аналогично `atermr`.)

Операторы `vatl()` и `vatr()` заполняют четыре ячейки в таблице переменных; две первые нужны для хранения переменной – имени аппликативного терма, за ними хранятся будущие левая и правая стрелки.

В конце статьи приведена статистика: насколько часто встречается та или иная подпоследовательность из $\{e_n\}$, $\{r_m\}$.

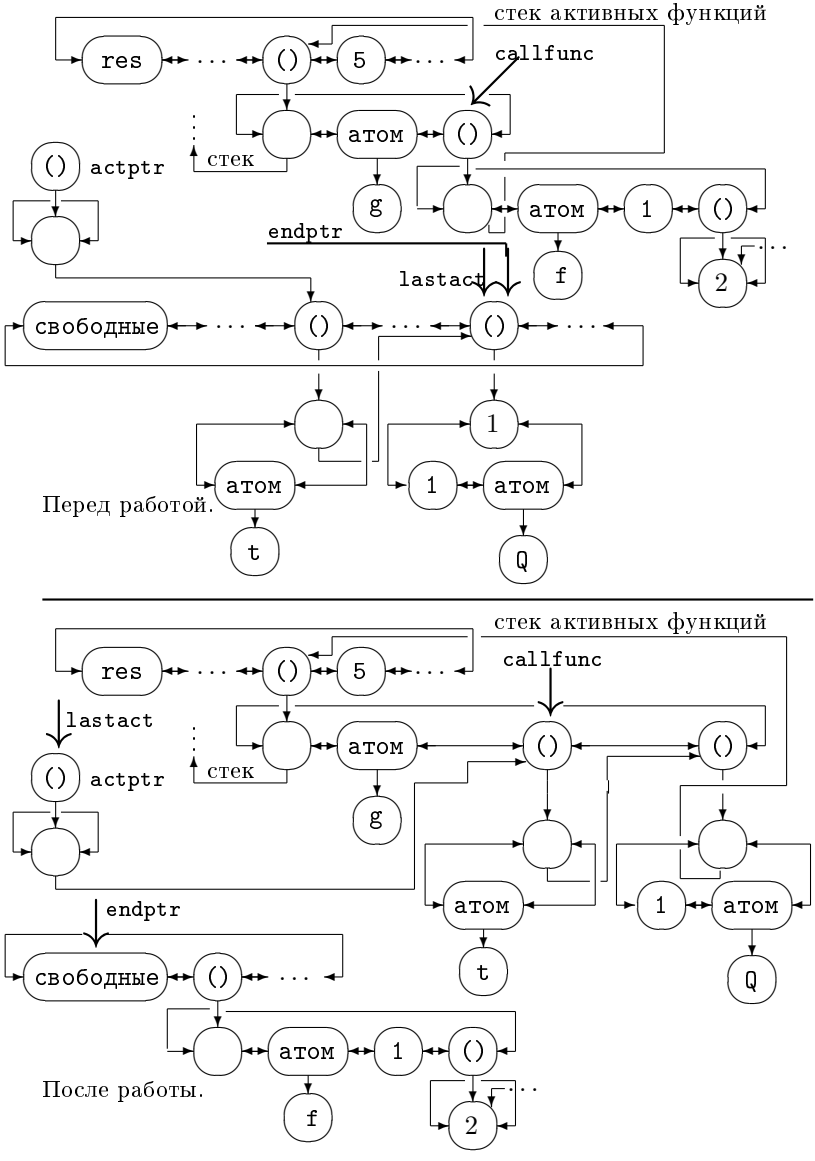


Рис. 28. Работа оператора `end()`.

§ 7. Программы на языке FAL

Рассмотрим предложение на языке FLAC $f($ x $) = y$ (f – атом, $x, y \in$ `FLAC(DATA)`). Мы научились по $f($ x $)$ строить последовательность $\{e_n^x\}$, решающую уравнение

$$f^*(x) = z \qquad \textbf{[I]}$$

для любого конкретного $z \in$ `FLAC(DATA)`; последовательность $\{r_m^y, \texttt{endop()}\}$ умеет строить правую часть нашего предложения по корню уравнения **[I]**. Предложением определяются не только сами операторы e_i, r_j, но и их аргументы.

Построенное отображение из множества предложений в множество конечных последовательностей операторов FAL является компилятором с FLACa на FAL.

Если мы закодируем e_i и r_j натуральными числами (см. таблицу в п.16), а аргументы будем записывать следом за оператором, опуская скобки, – получим программу на FALe, которая есть список из `DATA(FLAC)`. (Пример: `equal(2 8)` выглядит последовательностью `19 2 8`.) При работе FLAC-машины счетчик команд помнит вершину графа кодов, соответствующую выполняющемуся в данный момент оператору. Все e_i, r_j, `end()` «знают», сколько у них аргументов, и считают таковыми вершины, стоящие за ними; каждая команда (оператор языка FAL) передвигает счетчик за аргументы, где и находится следующий элемент последовательности $\{e_n\}$ ($\{r_m\}$, `end()`). Следовательно, всегда выполняется оператор, указанный счетчиком команд.

Чтобы помнить, какая функция вычисляется FAL-программой, поместим последнюю в скобки аппликативного терма с именем нашей функции; отделим также скобками операторы $\{e_n\}$ от операторов $\{r_m, \texttt{endop()}\}$.

Мы завершили описание компилятора с языка FLAC на язык FAL. Программы на FLACe и программы на FALe являются частью данных. Следовательно, компилятор `CFLAC` можно написать на FLACe. Данные функции `CFLAC` – программы на FLACe, результат – программы на FALe.

Примеры: (см. таблицу кодов)

```
1. CFLAC( eval( (#x) #y ) ) = ( eval( #x ) )  eval( #y ) )
       = eval( (20 16 17 0 0 16)
               (7 1 eval 23 0 4 5 5 1 eval 24 2 5 6)
             )
   CFLAC( eval( &x #y ) = &x eval( #y ) )
       = eval( (10 16)  (4 0  1 eval 24 2 5 6))

   CFLAC( = ) = ((18)  (6))
```

В первом предложении команда (23) «знает», что у нее должно быть два аргумента, – первым считает (0), вторым (4); после выполнения счетчик команд установлен на (5).

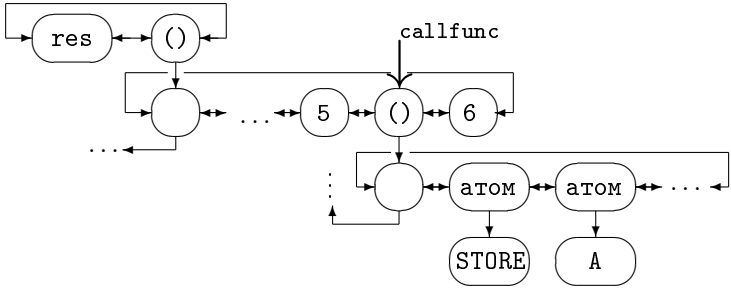
```
2. CFLAC( f( &x )  = g( &x &x ) )
            = f( (10 18) (1 g 4 0 4 0 5 6))
```

```
CFLAC( g( ( &y (#y)) #z ) = #y #z )
         = g( (14 16 17 2 0 20 18 17 6 1 16)
              (23 1 8 24 4 6)
            )
```

§ 8. Встроенные функции

Встроенная функция – это отображение из множества наших графов в себя. Как и для операторов языка FAL, процесс их вычисления FLAC-машиной нельзя разбить на более элементарные операции.

Рассмотрим ситуацию, когда `callfunc` указывает на граф – аппликативный терм с именем встроенной функции:



Работа встроенной функции:

1) По стрелке `callfunc` находятся собственные аргументы; на первом «этаже» после имени функции. Проверяется определена ли наша функция на таких графах-аргументах. Это, по существу, отождествление средствами самой функции.

2) Если неудача, `callfunc` передвигается на один шаг вниз по стеку. (Задержка: значением функции является ее вызов, поэтому ничего, кроме стека, менять в графе, где хранятся результаты вычислений, не нужно.) Следовательно, встроенные функции нельзя доопределять; у них всегда одно «предложение».

3) Если функция определена на указанных аргументах, то строится в списке свободных вершин граф-результат; вызов «выкидывается» из стека (то есть `callfunc` делает шаг вниз по стеку).

4) Результат вставляется на место вызова, который, в свою очередь, «вшивается» в список свободных после отмеченной вершины. До и после работы функции `endptr` указывает на последнюю, перемещается по графу свободных вершин во время построения значения.

Примеры:

1) Функция STORE определена на непустом списке, первый элемент которого должен быть атомом, отличным от имени любой встроенной функции. Значение функции – пустой список, но она изменяет графы, хранящиеся внутри FLAC-машины.

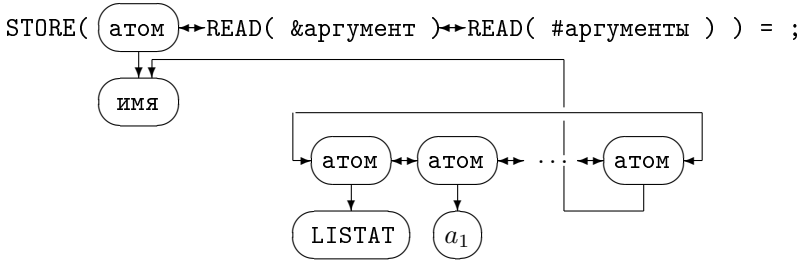


Таблица атомов перед вычислением STORE.

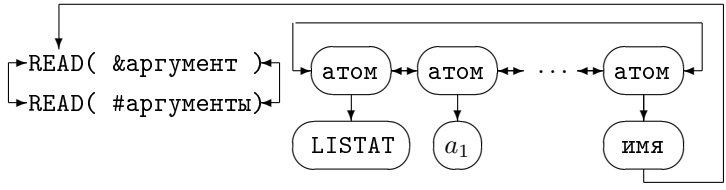
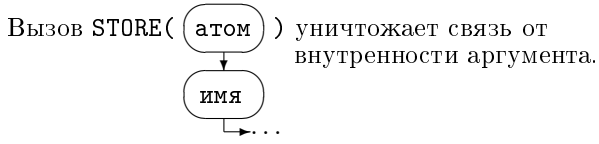


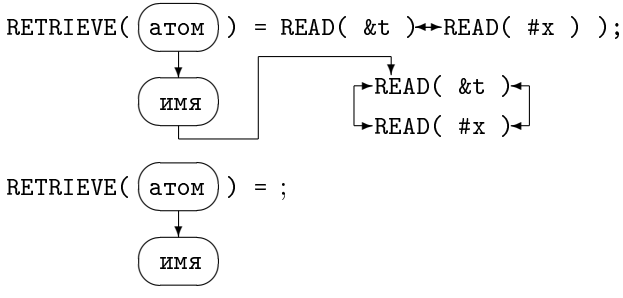
Таблица атомов после вычисления STORE.



Можно считать, что STORE присваивает своему первому аргументу значение – список оставшихся аргументов.

Значение доступно через таблицу атомов; при повторном присваивании предыдущее значение «вшивается» в граф свободных вершин после endptr.

2) Функция RETRIEVE определена на одном атоме.



При работе функции побочных эффектов нет (значение атома не теряется).

Мы видим: встроенные функции, кроме быстрого построения графов, позволяют менять списки, хранящиеся во FLAC-машине и недоступные операторам языка FAL, или обращаться к этим спискам.

§ 9. Загрузка. Механизм модульности.

Отображение `READ` переводит программу (которая находится в файле с расширением «`.fl`») в графы; значение компилятора – граф (программа на языке FAL) подается аргументом отображению `FPRINT`, обратному к `READ`; и в файл с расширением «`.cod`» записывается результат трансляции. (Это общение FLAC-машины с «внешним миром»; `READ` и `FPRINT` – встроенные функции.)

Функция `READ` «читает» данные из файла до ограничителя «`;`», при повторном вызове происходит ввод следующего предложения.

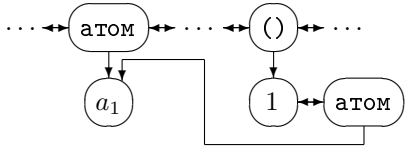
Содержимое FLAC-машины [см. описание п.4] дополняет программами функция `LOAD`. Ее аргумент – имя файла с FAL программой. Эта функция реализует механизм модульности; написана на языке FLAC.

Здесь ключевое понятие – таблица атомов; существенным является и то, что сравнение двух атомов при отождествлении происходит по их «внутренностям» [см. п.6.1].

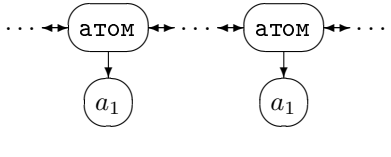
Определение 1. Два графа-атома равны тогда и только тогда, когда «внутренность» первого атома является «внутренностью» второго (*«внутренности» представлены одной вершиной графа*).

Отсюда следует совпадение имен атомов. Обратное, как мы увидим, неверно.

Примеры:

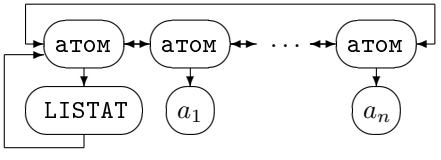


Атомы равны.



Разные атомы.

В текущей таблице атомов установим еще одну связь: между «внутренностью» отмеченной вершины и самой вершиной.



Мы первому (отмеченному) атому присвоили список – таблицу атомов [см. п. 8]; теперь легко найти таблицу из системы FLAC – значение функции `RETRIEVE( LISTAT )` есть ее копия.

LOAD(имя)
{

1) Читаются два первых предложения из файла «имя.cod»:

module "имя модуля";
PORT(имя_1, имя_2, ..., имя_n); **[I]**

(они и end оставляются компилятором без изменений). Атомы module, "имя модуля", PORT, имя_1, имя_2, ..., имя_n попали в текущую таблицу атомов [см. описание NEW_ATOM, п.5]; к их внутренностям можно добраться из таблицы.

2) Формируем новую таблицу атомов: общедоступные имена в системе [см. статью [2] в данном сборнике], "имя модуля", имя_1, имя_2, ..., имя_n, имена всех модулей, уже загруженных в систему.

Текущая таблица атомов, без отмеченной вершины, сохраняется с помощью функции STORE. После этого

STORE(LISTAT "#новая таблица")

объявляет текущей таблицу, построенную нами.

3) Очередное прочитанное данное – предложение на FALe. Простым замечанием к операторам языка FAL является

Утверждение 2. Пусть f(x) = y предложение языка FLAC (x,y ∈ DATA(FLAC)).

CFLAC(f(x) = y) = f(&l &r).

Тогда множество атомов принадлежащих термам &r и &l есть множество атомов из x и y соответственно, без атомов, которые интерпретируются как переменные.

Из них закрытые в модуле [см. [2]] будут строиться вместе с «внутренностями» и попадать в текущую таблицу атомов, открытые атомы будут строиться по верхнему «этажу», то есть «внутренности» этих атомов совпадают с соответствующими «внутренностями» атомов в предшествующей таблице атомов.

Граф, представляющий предложение на FALe, запоминается:

a) Наш модуль загружается первым (ранее LOAD не работала);

STORE(f "имя модуля"(&r &l)). Операторы FALa из других предложений той же функции (из рассматриваемого модуля) добавляются справа к уже загруженным в порядке поступления.

В конце работы LOAD по имени f будет храниться граф аппликативного терма

"имя модуля"($\&l_1$ $\&r_1$ $\&l_2$ $\&r_2$ ... $\&l_n$ $\&r_n$).

Под именем MOD запоминается

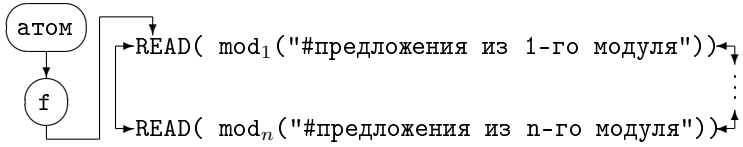
```
"имя модуля"(
  "#список имен всех функций в этом модуле"
            ) .
```

б) Ранее какие-то модули уже были загружены.

```
STORE(f RETRIEVE( f )
      "имя модуля"( &l1 &r1 ...&ln &rn )
     )
```

Подмена таблицы атомов гарантирует нас от добавления предложений к закрытой функции `f` из другого модуля (стрелка к содержимому идет из внутренности). Аналогично **а)** корректируется `MOD`.

Таким образом, программа – описание функции всегда хранится по атому, соответствующему имени этой функции:



$\texttt{mod}_1$, ..., $\texttt{mod}_n$ – имена модулей, где описана наша функция. А их порядок – порядок поступления (загрузки) в систему.

4) Модуль заканчивается предложением `end`. Восстанавливаем таблицу, имевшуюся в системе до подмены текущего списка атомов.

```
} /* завершили описание функции LOAD */
```

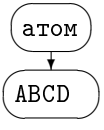
Пример:

Предположим, что перед началом работы `LOAD` атом `ABCD` уже находится в таблице. Он не принадлежит множеству общедоступных.

а) Атома `ABCD` нет в списке $\texttt{имя}_1$, ..., $\texttt{имя}_n$, `"имя модуля"` [см. [2]].

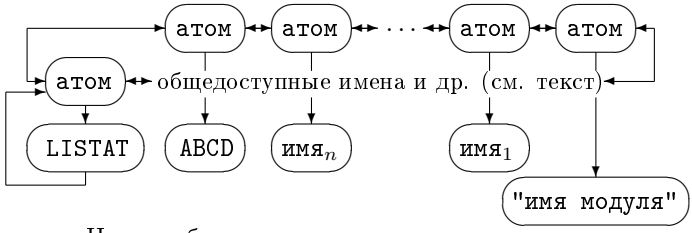
Следовательно, в новую таблицу, при ее инициализации, данный атом не попадет, хотя его «внутренность» имеется в системе.

В загружаемом модуле встретилось имя `"ABCD"`, – функция `NEW_ATOM( ABCD )` не находит в текущем списке атомов атома

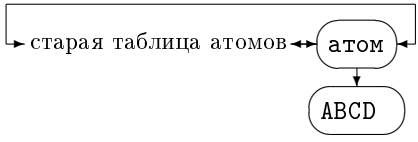


, строит его и добавляет в список. «Внутренность» построенного атома не совпадает с «внутренностью» атома с тем же именем, но существовавшем в системе до работы `LOAD` (так как одна «внутренность» уже существовала, другую построили в данный момент).

Сравнение атомов происходит по совпадению «внутренностей», значит имеем два разных атома.



Новая таблица атомов.



б) Второе предложение в модуле содержит наш атом (`PORT(` имя$_1$, ..., `ABCD`, ..., имя$_n$ `)`).

В этом случае новая таблица будет содержать атом с рассматриваемым именем. `NEW_ATOM( ABCD )` устанавливает ещё одну стрелку на имеющуюся «внутренность», не расширяя таблицу атомов (см. описание `NEW_ATOM`). Атом, построенный `READ`, отождествляется с уже бывшим.

Обратная к `LOAD` – функция `KILL( "имя модуля" )`. Она ищет в списке, хранящимся под именем `MOD` (см. описание `LOAD`), аппликативный терм

`"имя модуля"( "#список имен функций" )`.

Затем «выкидывает» те предложения функций (из списка имен функций), которые описаны в модуле `"имя модуля"`.

Администратор системы FLAC – это функции `LOAD` и `KILL`.

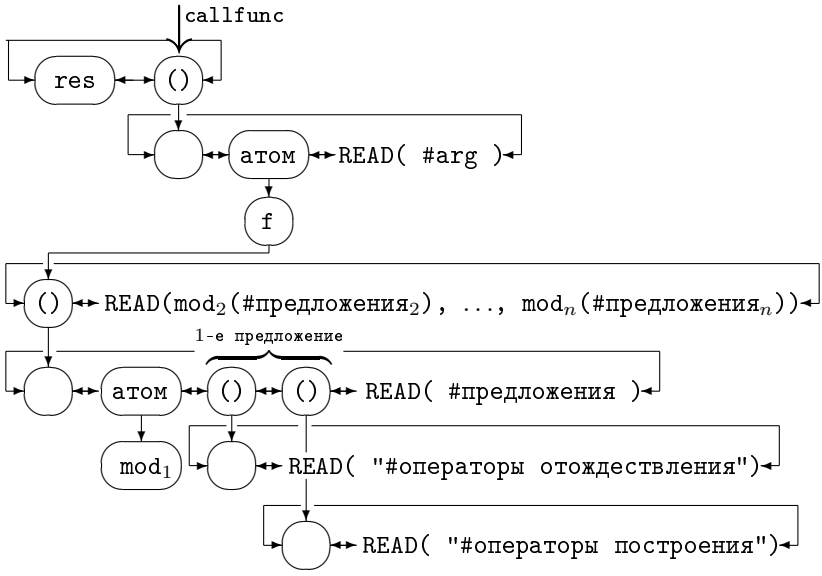
В момент включения FLAC-машины она содержит одну программу – диалог. Диалог представляет собой зацикленную программу, поддерживающую интерфейс системы с пользователем. Функция `Begin` (см. описание FLAC-машины) – это первая функция диалога.

`LOAD` и `KILL` в начальный момент отсутствуют; администратор загружается диалогом. После загрузки формируется таблица атомов – из общедоступных имен атомов; от пользователя закрываются имена системных атомов (например, имена функций диалога), имена некоторых встроенных функций (`STORE`, `RETRIEVE` и др.). Жесткие меры применяются для корректной работы системы; функция `STORE` позволяет, например, *переопределить* `LOAD`, «выкинуть» текущий список атомов.

Новая таблица атомов построена; диалог выходит на бесконечный цикл.

§ 10. Интерпретатор

Пусть в вершине стека активных функций находится аппликативный терм `f( #arg )`:



Двигаясь по стрелкам от `callfunc`, FLAC-машина находит имя функции.

Программа – правило вычисления этой функции находится очень быстро; нужно пройти по стрелке из «внутренности» атома (если стрелки нет – задержка).

Предложения функции хранятся в порядке их описания в программе; операторы языка FAL – в порядке их выполнения.

Счетчик команд инициализируется на первую команду отождествления в первом предложении, в первом по загрузке модуле. FLAC-машина всегда выполняет команду, указанную счетчиком. (Напомним, что операторы сами его передвигают.) Один из предикатов отождествления ложен – переход на следующее предложение. Не нашли нужного предложения в текущем модуле – нужно продолжать поиск в следующем. Все предложения исчерпаны – задержка. Уравнение, соответствующее типовому выражению левой части, решено – работают операторы построения; `endop` изменяет стек активных функций, и все начинается сначала.

§ 11. Оператор языка FAL – встроенная функция `QUOTE`

Оператор языка FAL – `QUOTE` – вызывается непосредственно из программ, написанных на FLACe:

```
QUOTE( &f( #arg ) &mod )                    [I]
```

Область определения: `&f` – имя функции, `&mod` – имя загруженного модуля.

Формально функцию `QUOTE` можно почти описать на FLACe:

```
QUOTE( &f( #arg ) &mode) = &f( #arg );     [II]
```

«Маленькая» тонкость: FLAC-машина пропустит предложения функции `&f`, описанные до `&mod` включительно; отождествление начнется с предложений, которые находятся в модулях, загруженных после модуля `&mod`.

Пример.

```
module first;
PORT( f );

   f( 1 ) = 1;
   f( 2 ) = QUOTE( f(3) first );
   f( 3 ) = first;

end;

module second;
PORT( f );

 f( 2 ) = 2;
 f( 3 ) = second;

end;
```

Результат вызова `f(2)` зависит от порядка загрузки:

a) `LOAD(first) LOAD(second)`:

```
        f(2) = second;
```

b) `LOAD(second) LOAD(first)`:

```
        f(2) = 2;
```

Следует обратить внимание: пример показывает, что в описании функции могут быть два предложения с совпадающими левыми частями.

Следующий текст поясняет рисунок из п.10.

Описание функции `QUOTE` на языке FAL:

```
QUOTE((22) ());
```

[III]

(22 – код оператора `quote`).

Адекватного текста на языке FLAC, очевидно, нет.

Оператор `quote` занимает выделенное место среди других, но его естественно отнести к предикатам отождествления. Как предикат, `quote` тождественно ложен; всю нужную работу для функции `QUOTE` он делает сам; операторы построения в **[III]** отсутствуют.

Если аргументы `#x` принадлежат области определения, `quote` работает согласно «описанию» **[II]**, далее «обманывает» FLAC-машину – имитирует, что произошло неудачное отождествление объектного выражения `&f( #arg )` с левой частью последнего предложения (в модуле `&mod`) функции `&f`.

Если аргументы `#x` не принадлежат области определения, то `quote` прекращает работу. Так как `quote` тождественно ложен, в этом случае `QUOTE( #x )` задерживается.

Нужная реакция FLAC-машины обеспечена.

§ 12. Арифметика

Активизируем функцию `NEW_NUMB` [см. п.2].

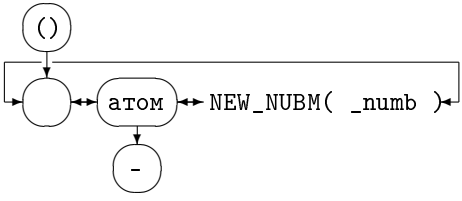
Пусть функция `Calc` переводит десятичное представление натурального числа в его представление в системе счисления по основанию β ($\beta \in \mathbb{N}_+$); упорядоченную последовательность десятичных цифр в упорядоченную последовательность β-цифр.

`Calc(` β`,` $\overline{d_n \ldots d_0}$ `)` = $\overline{D_m \ldots D_0}$.

($d_i, D_j \in \mathbb{N}$, а $i : 0 < i < n, 0 < d_i < 10$ и $j : 0 < j < m, 0 < D_j < \beta$).

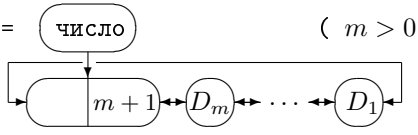
$(m+1)$ – называется β-длиной числа. Если $m > 1$, то число будем называть длинным.

`NEW_NUMB( (- _numb) ) =`



`NEW_NUMB( _numb ) = NUMBER( Calc( 256, _numb ) );`

`NUMBER(` D_0 `) =` 

`NUMBER(` $\overline{D_m \ldots D_0}$ `) =` 

($m > 0$)

Как и в случае аппликативных термов, мы быстро копируем длинные числа, экономно используем свободные вершины, – строя только корень графа и проводя стрелку к служебной вершине существующего. Эта вершина содержит, кроме числа стрелок, указывающих на нее, количество цифр в числе. Еще одно отличие от аппликативных термов: к нижнему этажу нельзя добраться средствами языка FLAC, он доступен только встроенным функциям.

Длина числа помогает, не проходя нижние этажи, быстро обнаружить неравенство чисел при отождествлении (стандартная ситуация); встроенные функции арифметики

`( ADD(_n1,_n2), SUB(_n1,_n2), MULT(_n1,_n2), DIV(_n1,_n2) )`

могут, используя длины аргументов, «понять», сколько им нужно свободных вершин для того, чтобы построить результат.

Последнее важно: алгоритмы сложения, вычитания и умножения «столбиком» двигаются от конца чисел к первым цифрам, формируя результат с последних цифр. Указанный проход по аргументам осуществляется легко, – сдвигаемся вправо от служебной вершины. Длины аргументов определяют (с точностью до 1) длину результата; мы можем строить нижний «этаж» в списке свободных вершин справа налево; список цифр получился в правильном порядке (не тратили время на «перекидывание» стрелок, – смотри в п.13.1 представление вершины).

Первые значащие цифры числа тоже легко получить – стрелки на нижнем «этаже» двунаправленные.

Особо следует сказать о функции `DIV` (деление с остатком).

Деление на ноль приводит к задержке. Первый аргумент делится на второй, не обращая внимание на знаки; знак частного q – произведение знаков делимого и делителя; знак остатка r *совпадает со знаком делимого.* Тождество: $\_n_1 = q * \_n_2 + r$. Значение функции: `DIV( _n1,_n2 ) =` $q\ r$ (список из частного и остатка).

Пусть $\_n_1 = \overline{a_n a_{n-1} \dots a_0}$, $\_n_2 = \overline{b_m b_{m-1} \dots b_0}$ ($n \geqslant m \geqslant 1$) **[I]** представление делимого и делителя в системе счисления по основанию β. В алгоритме деления «столбиком» есть неопределенность – нужно угадывать пробные цифры частного.

Сколько нужно взять первых значащих цифр в делимом (делителе), чтобы сделать предположение о цифре частного? Достаточно $m+1$ первых цифр делимого и все цифры делителя, – это дает точную цифру результата (целая часть от деления выбранных чисел). Но мы обманули себя; как разделить длинное число из $m+1$ цифры на делитель длины m? Попытаемся свести к минимуму число выбираемых значащих цифр.

Обозначим через d правильную первую цифру частного. Что дают первые значащие цифры? Неравенство

$$\frac{a_n}{b_m+1} < d < \frac{a_n+1}{b_m}$$

очевидно. Когда $a_n = \beta - 1$, $b_m = 1$ имеем $\frac{\beta-1}{2} < d < \beta$. То есть пробная цифра $\delta = \left[\frac{a_n}{b_m}\right]$ может оказаться от правильной очень далеко – на расстоянии $\left[\frac{\beta-1}{2}\right] + 1$. Пример деления 90 на 19 в десятичной системе счисления показывает, что оценка точная. Если β большое, такая ситуация, конечно, не устраивает; понадобится много лишних умножений длинных чисел («дорогая» операция).

Первые две значащие цифры делимого и одна делителя спасают положение. В качестве пробной нужно брать

$$\delta = \min\left(\left[\frac{a_n * \beta + a_{n-1}}{b_m}\right], \beta - 1\right) \qquad \textbf{[II]}$$

Теорема 1. $\delta \geqslant d$.

Теорема 2. Для любого $\beta \in \mathbb{N}_+$ если $b_m \geqslant \left[\frac{\beta}{2}\right]$, то $\delta - 2 \leqslant d \leqslant \delta$.

Пробная цифра никогда не отличается от истинной более чем на 2!

Пусть

$$\overline{a'_l a'_{l-1} \dots a'_0} = \left[\frac{\beta}{b_m+1}\right] * \overline{a_n a_{n-1} \dots a_0},$$

$$\overline{b'_k b'_{k-1} \dots b'_0} = \left[\frac{\beta}{b_m+1}\right] * \overline{b_m b_{m-1} \dots b_0}.$$

$$q = \left[\frac{\overline{a'_l a'_{l-1} \dots a'_0}}{\overline{b'_k b'_{k-1} \dots b'_0}}\right] = \left[\frac{\overline{a_n a_{n-1} \dots a_0}}{\overline{b_m b_{m-1} \dots b_0}}\right], r = r' / \left[\frac{\beta}{b_m+1}\right]. \qquad \textbf{[III]}$$

Тогда верна

Теорема 3. $m = k, b'_m \geqslant \left[\frac{\beta}{2}\right]$.

Доказательство теорем можно найти в [5].

Остался случай, когда длинное число ($n \geqslant 1$) делится на цифру ($m = 0$). Он тривиален: нужная цифра равна $\left[\frac{a_n}{b_0}\right]$ если $a_n \geqslant b_0$, и $\left[\frac{a_n * \beta + a_{n-1}}{b_0}\right]$ иначе.

После нормализации делимого и делителя: если $a'_l \geqslant b'_m$, то $\delta = d = 1$ (см. **[I]**), если $a'_l < b'_m$, то $\delta = \min(\left[\frac{a'_l * \beta + a'_{l-1}}{b'_m}\right], \beta - 1)$, и по теореме 2 угадываем правильную цифру за три шага (количество шагов не зависит от β !).

Остаток вычисляем по формуле **[III]**. Здесь уже длинное число делится на цифру.

Отметим еще, что (как и любая другая встроенная функция) `DIV` не имеет права портить нижние «этажи» своих аргументов, – они могут быть нужны еще каким-то числам. Мы построили нормализованное делимое; число контролируем полностью; результат промежуточных вычитаний можно записывать на лидирующие цифры нормализованного делимого, не используя вершины в списке свободных.

§ 13. Отражение состояния FLAC-машины в памяти компьютера

13.1. Представление вершины графа.

Вершины наших графов состоят из четырех полей

TYPE	INF	PREV	NEXT

и занимают 7 слов памяти; поля `PREV` и `NEXT` используются для связывания вершин на одном «этаже» (`PREV` – стрелка влево, `NEXT` – вправо); `INF` реализует стрелки на нижние «этажи», ссылки при организации стека левых скобок, стека активных функций. Поэтому названные поля длиной в 2 слова каждое (минимальная единица памяти, где можно хранить адрес (IBM PC XT)). Если из вершины выходит только две стрелки, в поле `INF` хранится другая информация; вершина-цифра содержит в этом поле свое значение – в первом слове; во втором слове служебная ячейка числа содержит длину этого числа.

Для одного мегабайта оперативной памяти (учитывая, что в ней находится операционная система, система FLAC) по одному слову на количество ссылок и на длину чисел вполне достаточно. Количество вершин не может превысить $2^{20}/14 \approx 2^{16}$.

Поле `TYPE`, занимающее слово, содержит значение признака (типа) вершины. Младшие биты соответствуют типам (цифра, число, скобка, атом, атом – имя встроенной функции); старшие свободные биты используются для временного хранения информации. (Например, при сборке «мусора» среди атомов [см. п.14].)

13.2. «Внутренности» атомов.

«Внутренность» атома имеет тип атома или имени встроенной функции.

Длина текста атома ограничивается 80 символами и не может поместиться в поле `INF`. Тексты атомов хранятся в отдельном массиве; поле `INF` «внутренности» содержит адрес текста. В массиве конец текста данного атома находится по символу `'\0'`.

Поле `PREV` «внутренности» содержит:

а) адрес графа-программы, если атом – имя функции, написанной на FLACe;
б) указатель на функцию, если атом – имя встроенной функции (написанной на «С»);
в) ноль в альтернативном случае.

«Внутренности» атомов образуют однонаправленный список – кольцо [см. п.14]; его связывает поле `NEXT`.

13.3. Встроенные функции и операторы языка FAL.

Операторы языка FAL, как и встроенные функции, вызываются по указателю.

Адреса операторов FALa находятся в массиве: код оператора (см. таблицу в конце статьи) – номер ячейки.

Встроенным функциям тоже соответствует массив: в четных ячейках – адреса функций, в нечетных – указатели на имена.

Такая организация памяти позволяет легко расширять список операторов отождествления (построения), делает гибким аппарат встроенных функций.

13.4. Выбор основания системы счисления.

Корректная арифметика на выбранном компьютере при переходе результата операции за слово становится неудовлетворительной по скорости. Эксперименты показали, что на длинных числах встроенные во FLAC функции арифметики работают быстрее при основании 256 (половина слова), чем при $\beta = 2^{16}$. Несмотря на то, что длины чисел увеличились в два раза. По той же причине при выводе чисел основание $\beta = 100$.

§ 14. Счетчик ссылок позволяет избавиться от глобальной сборки «мусора». Сборка «мусора» в области атомов.

Встроенные функции и отображение `READ` строят графы, используя свободные вершины.

Когда графы становятся ненужными?

а) После того как отработал очередной вызов функции, его граф – аппликативный терм «вшивается» в список свободных вершин (мы будем говорить: граф «выкинут»).
б) Встроенная функция рисовала графы на время своей работы; она должна сама позаботиться, чтобы и они оказались среди свободных вершин.

В списке свободных вершин графы представлены своими верхними «этажами». При выборе очередной свободной вершины, если она (`()`) или (`число`), смотрим на служебную вершину. Если количество стрелок, указывающих на нее – «1», – нижний «этаж» поднимаем в список свободных (после `endptr`),

иначе уменьшаем содержимое вершины на 1, обрываем связь с ней, объявляя «лестницу» вниз вершиной типа цифра. (На нижний «этаж» еще ведут стрелки, следовательно, он не потерян.) Графы обрабатываются рекурсивно.

Спуск вниз имеется еще у вершин (атом) ; аналогичную процедуру с ними проделать нельзя, так как количество стрелок, ведущих к «внутренности», не фиксируется.

При работе LOAD закрытые в модуле атомы не попадают в таблицу атомов диалога; после KILL их верхние вершины окажутся среди свободных, и после переопределения с «внутренностями» связь полностью потеряна.

По этой причине «внутренности» и связываются (функцией NEW_ATOM) в однонаправленное кольцо [см. п.13.2]; теперь к любой «внутренности» можно добраться через произвольно взятую вершину (атом) .

Суммарная длина имен атомов, появляющихся (посредством READ) во FLAC-машине, ограничена длиной массива текстов атомов. Верхние вершины закрытых атомов исчезают, не освобождая место, занимаемое в массиве; возникает необходимость корректировки массива (соответственно полей INF нужных внутренностей).

Алгоритм сборки «мусора» в области атомов:

1) Просматриваем все графы (исключая список «внутренностей»), имеющиеся во FLAC-машине, отмечая попадающиеся атомы одним из старших битов в полях TYPE «внутренностей».

2) Идем по списку «внутренностей». Неотмеченные (следовательно, ненужные) выбрасываем в список свободных, одновременно очищая массив текстов. В нужной вершине опускаем поднятый ранее флажок.

§ 15. Реакция системы на ошибки. Встроенные функции RUNEND, SYNTAX.

В некоторых случаях работа FLAC-машины приводит к ошибке.

Например:

а) использованы все свободные вершины, и очередная правая часть предложения не может быть построена;

б) у отображения READ аргумент не принадлежит множеству данных, – соответствующая встроенная функция вычислиться не смогла.

В случае ошибки оператор языка FAL (встроенная функция), в котором ошибка обнаружилась, сообщает о ней машине. FLAC-машина, устранив все «дыры» в списке свободных вершин, начинает движение по стеку активных функций вниз до первого вызова встроенной функции RUNEND. При движении из графа вызовов «выкидываются» пройденные аппликативные термы, кроме первого (где возникла ошибка).

Работа начинается заново с найденного вызова RUNEND.

`RUNEND( #x ) = {` список из двух термов: 1) функция работает после ошибки в системе: первый терм число – код ошибки, второй – `ERR( &1 #2 )` (`&1` – вызов функции, где возникла ошибка, `#2` – определившиеся аргументы функции `RUNEND`); 2) при нормальной работе FLAC-машины пришла очередь вычисления: результат – `0 N( #x )`.
`}`

Пропустить часть функций в стеке можно и языковыми средствами; встроенная функция `SYNTAX( #x )` имитирует синтаксическую ошибку. «`^C`» тоже устанавливает ошибку.

§ 16. Таблица кодов языка FAL. Статистические тесты.

Предикаты отождествления.			Операторы построения, замены.		
имя	код	количество аргументов	имя	код	количество аргументов
terml	8	1	rterm	0	1
termr	9	1	lbr	7	0
vterml	10	0	rbr	3	0
vtermr	11	0	rcopy	4	1
vnumbl	27	0	rmoveh	24	1
vnumbr	28	0	rmove	23	2
equal	19	2	raterm	1	1
list	16	0	rvaterm	2	1
empty	18	0	rvamvh	26	1
brackl	20	0	rvamove	25	2
brackr	21	0	rbrack	5	0
set	17	2	endop	6	0
aterml	12	1			
atermr	13	1			
vatl	14	0			
vatr	15	0			
quote	22	0			

О количестве оперативной памяти, оставшейся под задачу, сообщает встроенная функция `RECLAIM()`. Вызовем ее после загрузки FLAC-системы (в памяти диалог и администратор) – значение функции `34357 4222`. (MS DOS version 3.30)

Загружен компилятор: `RECLAIM() = 31136 3714`.

Загружена программа решения системы линейных уравнений над евклидовым кольцом (текст программы см. в статье [8]):

`RECLAIM() = 31722 3796`.

Первое число здесь означает количество свободных вершин, второе – сколько осталось свободных байт в массиве текстов атомов.

Последнее число относительно, так как может произойти «сборка мусора» [см. п.14].

Скорость выполнения некоторых операторов языка FAL. (Время работы предикатов отождествления приведено для ситуации, когда их значение – истина.) 10 000 вызовов.

Оператор.	Время выполнения. (десятые доли секунды)	Оператор.	Время выполнения. (десятые доли секунды)
`terml`, `termr`	28	`rterm`	41
`vterml`, `vtermr`	26	`lbr`	57
`list`	18	`rbr`	36
`empty`	12	`rcopy` (одной вершины)	54
`brackl`, `brackr`	38	`raterm`	96
`aterml`, `atermr`	50	`rbrack`	45
`set`	23		
`vatl`, `vatr`	52		

Количество вызовов операторов языка FAL.

а) Компиляция программы решения системы линейных уравнений [см. [8]].

б) Решение невырожденной системы линейных уравнений (16×16) над кольцом Гауссовых чисел [см. [8]].

в) Проход по списку из 5000 вершин [см. [2]].

г) Вычисление числа 300! [см. [2]].

код	а)	б)	в)	г)
0	21078	1899	203	303
1	27833	33265	5363	909
2	264	2971	103	2
3	4641	9900	5151	0
4	23950	51958	106	605
5	28746	34889	5365	911
6	21169	12487	5260	309
7	3571	4288	0	0
8	4668	319	0	0
9	1081	0	0	0
10	23007	21192	2	2
11	662	476	5252	0
12	590	318	0	0
13	3302	0	0	0
14	664	5259	0	0
15	84	0	0	0
16	15630	5982	10406	4
17	5278	13146	5153	2
18	6909	19587	6	306
19	1791	436	0	0
20	3576	362	2	2
21	533	48	0	0
22	0	0	0	0
23	2489	289	2	2
24	11701	444	5050	1
25	1719	4203	0	0
26	0	62	0	0
27	240	2	0	0

Наиболее часто встречающиеся последовательности операторов. Можно, добиваясь эффективности как по памяти так и по скорости, добавлять новые операторы, имитирующие работу указанных последовательностей.

В таблице каждая последовательность находится в скобках, элемент последовательности – код оператора.

Частота.	Объектный модуль компилятора. (cflac.cod)	Частота.	Объектный модуль программы решения системы линейных уравнений.
от 30 до 35 раз	(1, 0) (0, 5)	от 50 до 55 раз	(5, 6) (1, 4)
27	(5, 1)	от 45 до 50	(16,17), (4, 5) (17,16)
от 10 до 15	(1, 1), (5,5) (5, 6), (0,0) (0, 1), (5, 1, 0) (1, 0, 5) (0, 5, 1) (0, 5, 5)	42	(4, 4)
		от 30 до 35	(10,10), (11,16) (4, 4, 5)
		от 25 до 30	(10,18), (1, 4, 4) (16,17,16)
до 5 до 10	(1, 4), (0, 0, 5) (1, 0, 0), (1, 5) (5, 1, 1), (5, 0) (1, 0, 1) (5, 0, 5) (1, 1, 0) (5, 5, 1), (0, 5, 0) (0, 0, 0), (5, 5, 6) (0, 1, 0) (0, 5, 1, 0) (5, 1, 0, 0) (1, 1, 0, 5) (0, 0, 5, 1) (0, 5, 0, 5) (1, 0, 0, 0) (1, 0, 5, 5) (1, 0, 5, 1)	от 20 до 25	(5, 5), (1, 1) (1, 5), (3, 5) (7, 1), (5, 3) (18,17) (1, 4, 4, 5)
		от 15 до 20	(4, 1), (5, 1) (3, 4), (11,16,17) (3, 5, 6) (4, 5, 5) (10,10,10)
		от 10 до 15	(17,11), (10,14) (4, 6), (3, 6) (7, 3), (5, 4) (8,18), (3, 1) (3,7), (4,4,5,5) (16,17,11) (17,16,17)
		от 5 до 10 (4 оператора)	(17,16,17,16) (4,1,4,4),(1,4,1,4) (1, 1, 4, 4) (16,17,11,16) (11,16,17,16) (4, 5, 5, 6) (11,16,17,11) (16,17,16,17) (10,10,10,19)

Список литературы

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

[2] Е. А. Гайдар, И. Н. 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, **1**, Издательство Сборник, Переславль-Залесский, 2014(1988), 11–42.

[3] В. Л. Кистлеров, "Принципы построения языка алгебраических вычислений FLAC", Препринт, ИПУ АН СССР, Москва, 1987.

[4] Д. Кнут, *Искусство программирования для ЭВМ. Получисленные алгоритмы*, **2**, Мир, Москва, 1977.

[5] Дж. Э. Коллинз, М. Миньотт, Ф. Уинклер, "Арифметика в основных алгебраических областях", *Компьютерная алгебра. Символьные и алгебраические вычисления*, ред. Б. Бухбергер, Дж. Коллинза и Р. Лоос, Мир, Москва, 1986.

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

[7] Ан. В. Климов, С. А. Романенко, "Система программирования РЕФАЛ-2 для ЕС ЭВМ. Описание входного языка", Препринт, ИПМ: им. М.В. Келдыша АН СССР, Москва, 1987.

[8] Н. А. Чмутова, "Общее решение системы линейных уравнений над евклидовым кольцом", Сборник трудов по функциональному языку программирования Рефал, **1**, Издательство Сборник, Переславль-Залесский, 2014(1988), 92–117.

Ниже дан список литературы, добавленный редактором сборника.

[9] 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.

[10] S. V. Chmutov, E. A. Gaydar, I. M. Ignatovich, V. F. Kozadoy, A. P. Nemytykh, V. A. Pinchuk, *The symbol analytic transformation language FLAC: sources, executable modules*, ftp://www.botik.ru/pub/local/scp/flac/flac386.zip, 1991.

[11] С. Д. Мешвелиани, *Система сивольных алгебраических вычислений CAC*, NN, 1993.

[12] С. Д. Мешвелиани, *Компьторная алгебраическая система вычислений Docon-FLAC*, ftp://www.botik.ru/pub/local/scp/flac/docon_flac.zip, 1994.

Е.А.Гайдар (E. A. Gaydar)
Переславль-Залесский

И.М.Игнатович (I. M. Ignatovich)
Переславль-Залесский

В.Ф.Козадой (V. F. Kozadoy)
Переславль-Залесский

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

В.А.Пинчук (V. A. Pinchuk)
Переславль-Залесский

С.В.Чмутов (S. V. Chmutov)
Переславль-Залесский

Нина А. Чмутова

Общее решение системы линейных уравнений над евклидовым кольцом

В статье описана программа для нахождения всех решений системы линейных уравнений над произвольным евклидовым кольцом. Программа написана на языке программирования FLAC. На примере этой программы демонстрируется естественность работы в языке FLAC с абстрактными математическими объектами. Библ. 12 наим.

Содержание

Работа выполнена в 1988 г.

§ 1. Введение

Решение систем линейных уравнений является классической и хорошо изученной областью. Практически в каждом учебнике линейной алгебры имеются разделы, посвященные этой области. Отметим также недавно вышедшую книгу [1], целиком посвященную решению систем линейных уравнений над евклидовым кольцом. В настоящей статье описывается программа для нахождения всех решений системы линейных неоднородных уравнений над произвольным евклидовым кольцом. Самый распространенный пример евклидового кольца – кольцо целых чисел. Таким образом, наша программа (в этом частном случае) находит все решения в целых числах линейных уравнений с целочисленными коэффициентами. Программа написана на языке программирования FLAC. Язык FLAC (Functional Language for Algebraic Calculation) разработан В. Л. Кистлеровым в 1986/87г. О языке FLAC см. [2],[3, 4][1]. Программа тестировалась на решении систем линейных уравнений над следующими кольцами:

- кольцо целых чисел $\mathbb{Z}$;
- кольцо рациональных чисел $\mathbb{Q}$;
- кольцо целых эйлеровых чисел $\mathbb{D}_3$ (кольцо комплексных чисел $a + b * e^{i*\frac{2*\pi}{3}}$, где $a, b \in \mathbb{Z}$);
- кольцо целых гауссовых чисел $\mathbb{D}_4$ (кольцо комплексных чисел $a + b * i$, где $a, b \in \mathbb{Z}$);
- кольцо целых чисел по модулю 3.

В следующей таблице приведены данные о времени работы программы на IBM PC XT.

Кольцо	Размер системы	Время работы
Int	6×6	7 сек.
	6×3	12 сек.
$\mathbb{D}_3$	6×6 невырожденная	33 сек.
$\mathbb{D}_4$	16×16 невырожденная	2 мин. 45 сек.

Эта программа использовалась для нахождения графов неприводимых представлений конечных групп, порожденных унитарными отражениями в двумерном комплексном пространстве. О графах представлений групп см. [5].

Кольца $\mathbb{D}_3, \mathbb{D}_4$ (и более общие кольца $\mathbb{D}_k$ комплексных чисел вида

$$a_0 + a_1 * e^{i*\frac{2*\pi}{k}} + \cdots + a_{k-1} * e^{i*\frac{2*\pi*(k-1)}{k}},$$

где $a_0, ..., a_{k-1} \in \mathbb{Z}$) играют важную роль в теории алгебраических чисел, и в частности в исследованиях, связанных с Великой теоремой Ферма [6],[7],[8],[9].

Эти кольца важны также для теории представлений конечных групп, поскольку характер всякого конечномерного представления конечной группы является функцией на группе со значениями в одном из таких колец [10].

[1] См. также [11],[12].

Программа для решения систем линейных уравнений состоит из 4-х модулей. Кроме этого, для решения конкретной системы над конкретным кольцом, необходим еще модуль `ring`, содержащий описание данного конкретного кольца. Таким образом, если пользователь пожелает решить систему над другим кольцом, для этого достаточно загрузить в систему новый модуль `ring` (старый модуль при этом автоматически удаляется из системы).

План статьи таков. В п.2 приводится определение евклидова кольца и примеры описания евклидовых колец на языке FLAC. В п.3 излагается алгоритм решения системы линейных уравнений. В п.4 описывается программа, реализующая алгоритм п.3. В п.5 приводятся тексты программы на языке FLAC.

Вопросы представления данных и реализации алгоритмов на языке FLAC обсуждались на семинаре по функциональному программированию в ИПС АН СССР. Автор пользуется случаем поблагодарить всех участников семинара, в особенности А. П. Немытых и В. А. Швачко за многочисленные полезные обсуждения.

§ 2. Евклидовы кольца

2.1. Определение евклидова кольца.

Евклидовым кольцом называется коммутативное, ассоциативное кольцо с единицей, снабженное нормой $\|W\|$ такой, что для любых двух элементов a, b $(b \neq 0)$ существуют такие элементы d, r , что $a = d * b + r$; причем либо $r = 0$ либо $\|r\| < \|b\|$.

Заметим, что под словом «существуют» в определении евклидова кольца подразумевается, что существует алгоритм (назовем его eualg), который по паре ненулевых элементов a, b выдает пару элементов d, r с указанными свойствами.

Существование алгоритма деления с остатком в евклидовом кольце позволяет реализовать алгоритм Евклида для нахождения наибольшего общего делителя (НОД) двух элементов кольца. Таким образом, в евклидовом кольце существует НОД любых двух элементов. Следовательно, существует НОК (наименьшее общее кратное) любых двух элементов. Кроме того, в евклидовом кольце справедлива основная теорема арифметики об однозначном разложении на простые множители. Эти факты играют ключевую роль во всех алгоритмах связанных с евклидовыми кольцами, включая и наш.

2.2. Описание евклидовых колец на языке FLAC.

Программа для решения систем линейных уравнений предполагает следующее:

- Каждый элемент кольца представляется своей нормальной формой, которая является термом языка FLAC (см. [3]). Это означает, что каждый элемент кольца представляется единственным термом языка FLAC и если термы, представляющие два элемента кольца, совпадают, то сами элементы тоже совпадают.

  Всюду далее предполагается, что элементы кольца представлены своими нормальными формами.

- Должны быть описаны следующие функции:
  `zero` – функция, выдающая нулевой элемент кольца;
  `unit` – функция, выдающая единицу кольца;
  `add` – функция сложения двух элементов кольца;
  `sub` – функция вычитания элементов кольца;
  `mult` – функция умножения элементов кольца;
  `eualg` – функция деления с остатком.

Эти требования означают, что в множестве термов языка FLAC выделяется некоторое подмножество, на котором вводится структура евклидова кольца, изоморфного исходному евклидову кольцу.

Перечисленные выше функции описываются в модуле с именем `ring`. Этот модуль и программа для решения систем уравнений совершенно независимы. Для решения системы над другим кольцом достаточно сменить этот модуль.

Приведем примеры модулей, описывающих различные кольца.

2.2.1. *Кольцо целых чисел.*

Множество термов, представляющих элементы кольца – это множество чисел (положительных, отрицательных и нуля) языка FLAC. Изоморфизм тождественный. Модуль выглядит следующим образом:

```
module ring;

PORT( zero unit add sub mult eualg );

  zero() = 0;
  unit() = 1;

  add( &x &y )   = ADD( &x &y );
  sub( &x &y )   = SUB( &x &y );
  mult( &x &y )  = MULT( &x &y );
  eualg( &x &y ) = DIV( &x &y );

end;
```

Функции `ADD`, `SUB`, `MULT`, `DIV` – это встроенные функции системы FLAC. Результатом вызова функции `DIV(a b)` является список из двух чисел q, r таких, что $\mathtt{a} = \mathtt{b} * q + r$; $\|r\| < \|\mathtt{b}\|$; знак r совпадает со знаком `a`.

2.2.2. *Кольцо целых чисел по модулю* 3.

Термы, представляющие элементы кольца, это числа 0; 1; 2. Это кольцо является полем. Всякое поле – евклидово кольцо. Норма – функция, тождественно равная 1. Любые два элемента делятся без остатка.

```
module ring;

PORT( zero unit add sub mult eualg );

  zero() = 0;
  unit() = 1;
```

```
    add( &a &b ) = REST( DIV( &a + &b , 3 ) );
    sub( &a &b ) = N( &a - &b );

         N( (- &n) ) = 3 - &n;
         N( &n )     = &n;

    mult( &a &b ) = REST( DIV( &a * &b , 3 ) );

    eualg( 0 &a ) = 0  &a:
    eualg( &a 1 ) = &a 0;
    eualg( 1 2 )  = 2  0;
    eualg( 2 2 )  = 1  0;

end;
```

В этом примере мы использовали инфиксные операции во FLACe. Функция `REST` – это встроенная функция; ее значение – аргумент без первого элемента. На FLACe описание функции `REST` выглядит так:

```
REST( &x #x ) = #x;
```

`N` – функция приведения элемента к нормальной форме в том случае, если в результате вычитания получится отрицательное число. Например, нам нужно вычислить `sub( 0 2 )`. В этом случае `&a` равно 0, а `&b` равно 2. Результат функции вычитания `&a - &b` – это аппликативный терм `(- 2)`. Напомним, что отрицательные числа во FLACe представляются аппликативными термами с именем `"-"` (см. [3]). Результатом функции `N` будет $3 - 2 = 1$. Таким образом результат `sub( 0 2 )` – это 1. Действительно, $0 - 2 = 1(mod\ 3)$.

2.2.3. *Кольцо целых гауссовых чисел.*

Целое гауссово число $a + b * i$, где a и b – целые числа, естественно представлять в языке FLAC аппликативным термом `(a b)`. При этом операции сложения, вычитания и умножения выглядят так:

```
(a b) + (c,d) = (a + c     b + d)
(a b) - (c,d) = (a - c     b - d)
(a b) * (c d) = (a*c - b*d   a*d + b*c).
```

На языке FLAC эти правила записываются в виде:

```
add( (&a &b)  (&c &d) )  = |(ADD(&a &c) ADD(&b &d));

sub( (&a &b)  (&c &d) )  = |(SUB(&a &c) SUB(&b &d));

mult( (&a &b)  (&c &d) ) = |(SUB(MULT(&a &c) MULT(&b &d))
                             ADD(MULT(&a &d) MULT(&b &c))
                            );
```

Функции, выдающие нулевой и единичный элементы кольца, имеют вид:

```
zero() = (0 0);
unit() = (1 0);
```

Теперь опишем алгоритм деления с остатком (`eualg`) в этом кольце.

Предположим, что нам нужно разделить целое гауссово число z_1 на z_2. Для этого рассмотрим частное z_1/z_2 этих комплексных чисел. Это рациональное гауссово число, т.е. $Re(z_1/z_2), Im(z_1/z_2) \in \mathbb{Q}$. Пусть $A + B * i$ – ближайшее целое гауссово число к z_1/z_2. Т.е.

$$z_1/z_2 = (A + B * i) + (\alpha + \beta * i)$$

, где $\alpha + \beta * i$ – некоторое рациональное гауссово число. Тогда $A + B * i$ – искомое частное, а $z_2 * (\alpha + \beta * i)$ – искомый остаток. Заметим, что $z_2 * (\alpha + \beta * i)$ – целое гауссово число и $\|z_2 * (\alpha + \beta * i)\| < \|z_2\|$ (в качестве нормы здесь рассматривается квадрат модуля комплексного числа).

Рассмотрим этот алгоритм подробнее:

1 Умножим числа z_1 и z_2 на $\overline{z}_2$ (черта обозначает комплексное сопряжение). Пусть $a_1 + b_1 * i = z_1\overline{z}_2$; $n = z_2\overline{z}_2 = \|z_2\|$.

2 Пусть `div1` – алгоритм, который по паре целых чисел x, n выдает пару целых чисел X и W таких, что $x = X * n + W$ и X – ближайшее целое число к рациональному x/n.

3 Применим алгоритм `div1` к паре a_1, n и к паре b_1, n. Пусть

`div1(`a_1 n`) =` A u
`div1(`b_1 n`) =` B v

4 Число $A + B * i$ – искомое частное.

5 Число $z_2 * (u + v * i)/n$ искомый остаток.

Эквивалентность этого описания алгоритма `eualg` предыдущему описанию следует из очевидных формул:

$$\alpha = u/n; \quad \beta = v/n.$$

Приведем программу на языке FLAC, реализующую алгоритм `eualg`.

Первый шаг алгоритма записывается в виде

```
eualg(&z1 &z2) = d(mult(&z1 dual(&z2)) norm(&z2) &z2);

dual((&a &b)) = |(&a SUB(0 &b));

norm((&a &b)) = ADD(MULT(&a &a) MULT(&b &b));
```

Функция `d` реализует шаги 2-5 нашего алгоритма. Аргументы функции `d` – это целое гауссово число (`&a1 &b1`), целое число n и z_2.

```
d((&a1 &b1) &n &z2) = g(div1(&a1 DIV(&a1 &n) &n)
                        div1(&b1 DIV(&b1 &n) &n)
                        &n &z2);
```

Шаги 2-3 алгоритма `eualg` реализует функция `div1`. А функция `g` реализует шаги 4-5. Вот ее описание:

```
g(&A &u &B &v &n &z2)
              = |(&A &B) |(g1(mult(&z2 |(&u &v)) &n));

g1((&a &b) &n) = FIRST(DIV(&a &n))
                 FIRST(DIV(&b &n));
```

Здесь мы воспользовались встроенной функцией FIRST, выдающей первый элемент списка аргументов.(На FLACe описание функции FIRST выглядит так: FIRST(&x #x) = &x;).

Остается еще описать алгоритм div1 нахождения ближайшего целого к данному рациональному. Действие этого алгоритма напоминает действие встроенной функции DIV (см. п. 2.2.1 или [3]). Однако, встроенная функция DIV в качестве частного может выдавать не ближайшее целое.

Алгоритм div1:

1 Пусть q и r – результат применения функции DIV к числам x, n.
2 Если $x < 0$, то
  2.1 положим $q := -q$, $r := -r$;
  2.2 выполняем шаги 3-4 алгоритма div1;
  2.3 умножаем каждое из двух чисел результата шага 2.2 на -1 и полученную пару чисел выдаем в качестве результата алгоритма div1.
3 Если $r < n - r$, то числа q, r выдаем в качестве результата алгоритма.
4 Если $r > n - r$, то в качестве результата выдаем числа $q + 1$ и $r - n$.

Первый шаг алгоритма осуществляется перед вызовом функции div1. Функция div1 получает в качестве аргументов целое число a, пару чисел q, r и число n.

```
div1( (-&a) &q &r &n) = minus(div1(&a SUB(0 &q)
                                        SUB(0 &r) &n));
div1(&a &q &r &n) = sgn(LESS(&r SUB(&n &r)) &q &r &n);
```

Первое предложение функции div1 отвечает шагу 2. алгоритма div1. Шаг 2.3 алгоритма div1 выполняет функция minus:

```
minus(&q &r) = SUB(0 &q) SUB(0 &r);
```

шаги 3-4 алгоритма div1 выполняет функция sgn.

```
sgn(TRUE &q &r &n)  = &q &r ;                  /*  шаг 3. */
sgn(FALSE &q &r &n) = ADD(&q 1) SUB(&r &n);    /*  шаг 4. */
```

Теперь остается только добавить, что все функции, здесь описанные, нужно оформить в модуль ring, аналогично п. 2.2.1 и п. 2.2.2.

§ 3. Алгоритм решения системы линейных уравнений

Описание алгоритма SOLVS решения системы линейных уравнений разобьем на 4 части:

- верхний уровень алгоритма SOLVS. Алгоритмы этой части сводят решение системы уравнений к решению одного уравнения.
- алгоритм QUOTN решения одного уравнения.
- алгоритм Tr_M приведения системы к верхнетреугольному виду.
- дополнительные алгоритмы.

3.1. Верхний уровень алгоритма SOLVS.

Пусть исходная система уравнений имеет вид:

$$\begin{cases} a_{1,1} * x_1 + \ldots + a_{1,n} * x_n = b_1 \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ a_{m,1} * x_1 + \ldots + a_{m,n} * x_n = b_m \end{cases} \qquad \textbf{(I)}$$

Систему линейных уравнений будем записывать в матричном виде

$$A\vec{x} = \vec{b}.$$

Алгоритм SOLVS:

1 Приводим матрицу системы к верхнетреугольному виду с помощью алгоритма Tr_M.

2 Если у полученной матрицы имеются нулевые строки и соответствующие им свободные члены b_i тоже равны нулю, то удаляем эти строки из системы. После этого шага можно считать, что система уравнений имеет вид:

$$\begin{cases} a_{1,1} * x_1 + \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots + a_{1,n} * x_n = b_1 \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ a_{i,s_i} * x_{s_i} + \ldots\ldots\ldots\ldots\ldots\ldots\ldots + a_{i,n} * x_n = b_i \\ a_{i+1,s_{i+1}} * x_{s_{i+1}} + \ldots\ldots\ldots + a_{i+1,n} * x_n = b_{i+1} \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ a_{m,s_m} * x_{s_m} + \ldots + a_{m,n} * x_n = b_m \end{cases}$$

3 Решаем последнее уравнение системы с помощью алгоритма QUOTN. Положим $i := n - 1$;

$\vec{sol}$ – частное решение системы всех последних уравнений, начиная с $(i+1)$-го;

base – базис решений системы соответствующих однородных уравнений, начиная с $(i+1)$-го. Таким образом,

$\vec{sol}$ – это вектор $\begin{pmatrix} x^0_{s_{(i+1)}} \\ \vdots \\ x^0_n \end{pmatrix}$ частного решения, а

base – матрица, столбцы которой суть векторы базиса решений однородной системы.

В матричном виде общее решение системы последних уравнений, начиная с $(i+1)$-го, выглядит так:

$$\begin{pmatrix} x_{s_{(i+1)}} \\ \vdots \\ x_n \end{pmatrix} = \vec{sol} + base \begin{pmatrix} C_1 \\ \vdots \\ C_k \end{pmatrix} \qquad \textbf{(II)}$$

4 Подставляем решение $\vec{sol} + base * \vec{C}$ в i-е уравнение системы. В матричном виде i-е уравнение имеет вид:

$$(a_{i,s_i}, \ldots, a_{i,n}) \begin{pmatrix} x_{s_i} \\ \vdots \\ x_n \end{pmatrix} = b_i$$

После подстановки получим:

$$(a_{i,s_i}, \ldots, a_{i,n}) \begin{pmatrix} x_{s_i} \\ \vdots \\ x_{(s_{(i+1)}-1)} \\ \vec{sol} + base \begin{pmatrix} C_1 \\ \vdots \\ C_k \end{pmatrix} \end{pmatrix} = b_i$$

Это уравнение на «новые» переменные $x_{s_i}, \ldots, x_{(s_{(i+1)}-1)}$ и «старые» константы $C_1, \ldots, C_k$.

Строка коэффициентов этого уравнения имеет вид:

$$(a_{i,s_i}, \ldots, a_{i,(s_{(i+1)}-1)}, (a_{i,s_{i+1}}, \ldots, a_{i,n}) * base)$$

Свободный член этого уравнения равен

$$b_i - (a_{i,s_{i+1}}, \ldots, a_{i,n}) * \vec{sol}$$

Полученное уравнение решаем с помощью `QUOTN`. Решение представим в виде:

$$\begin{pmatrix} x_{s_i} \\ \vdots \\ x_{(s_{(i+1)}-1)} \end{pmatrix} = \begin{pmatrix} x^0_{s_i} \\ \vdots \\ x^0_{(s_{(i+1)}-1)} \end{pmatrix} + B \begin{pmatrix} C'_1 \\ \vdots \\ C'_{k'} \end{pmatrix}$$

$$\begin{pmatrix} C_1 \\ \vdots \\ C_k \end{pmatrix} = \begin{pmatrix} C^0_1 \\ \vdots \\ C^0_k \end{pmatrix} + constbas \begin{pmatrix} C'_1 \\ \vdots \\ C'_{k'} \end{pmatrix}$$

Подставляя последнюю формулу в **(II)**, получим общее решение системы всех уравнений, начиная с i-го

$$\begin{pmatrix} x_{s_i} \\ \vdots \\ x_{(s_{(i+1)}-1)} \\ x_{s_{(i+1)}} \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} x^0_{s_i} \\ \vdots \\ x^0_{(s_{(i+1)}-1)} \\ \vec{sol} + base \begin{pmatrix} C^0_1 \\ \vdots \\ C^0_k \end{pmatrix} \end{pmatrix} + \begin{pmatrix} B \\ constbas \end{pmatrix} \begin{pmatrix} C'_1 \\ \vdots \\ C'_{k'} \end{pmatrix}$$

Обозначим полученное частное решение через $\vec{sol}$, а базис решений однородных уравнений – через $base$, и $\vec{C}'$ – через $\vec{C}$.

5 Если $i = 1$, то $\vec{sol} + base * \vec{C}$ – результат алгоритма `SOLVS`, иначе положим $i := i - 1$ и переходим к п.4 алгоритма `SOLVS`.

3.2. Алгоритм `QUOTN` решения одного уравнения.
Пусть решаемое уравнение имеет вид:

$$a_1 * x_1 + \ldots + a_n * x_n = b.$$

Алгоритм `QUOTN`:

1 Удаляем первые нулевые коэффициенты нашего уравнения. При этом появляются базисные векторы решений однородного уравнения.

2 Если все $a_i = 0$ и $b = 0$, то система имеет нулевое частное решение и соответствующий базис решений.
Если все $a_i = 0$ а $b \neq 0$, то решения не существует.
Если только один коэффициент a_n отличен от нуля, то решаем уравнение $a_n * x = b$.
После этого шага можно считать, что в уравнении

$$a_1 * x_1 + \ldots + a_n * x_n = b$$

коэффициент a_1 не равен нулю.

3 Вычисляем НОД$(a_1, \ldots, a_n)$. Обозначим НОД$(a_1, \ldots, a_n)$ через d_n.
Если b не делится на d_n, то уравнение не имеет решений.
Если b делится на d_n, то решаем уравнение

$$\frac{a_1}{d_n} * x_1 + \ldots + \frac{a_n}{d_n} * x_n = 1.$$

Это уравнение решается применением алгоритма `PRIME`.
Полученное частное решение умножаем на b/d_n и вместе с соответствующим базисом решений подаем на выход алгоритма `QUOTN`.

Описываемый ниже алгоритм `PRIME` автор узнала от В. А. Швачко. Это рекурсивный алгоритм. Рекурсия проводится по числу n переменных. Через d_i будем обозначать НОД$(a_1, \ldots, a_i)$. Алгоритм `PRIME` решает уравнение:

$$a_1 * x_1 + \ldots + a_n * x_n = d_n$$

Алгоритм `PRIME`:

1 Положим $s := 2, d_1 := a_1$, $\vec{sol} := (x_1^0 := 1)$ – частное решение уравнения $a_1 * x_1 = d_1$.

2 Решаем уравнение $a_1 * x_1 + \ldots + a_s * x_s = d_s$. Рассмотрим частное решение $(x_1^0, \ldots, x_{(s-1)}^0)$ уравнения

$$a_1 * x_1 + \ldots + a_{(s-1)} * x_{(s-1)} = d_{(s-1)}$$

Пусть k и y – частное решение уравнения $d_{(s-1)} * k + a_s * y = d_s$. Тогда положим $\vec{sol} = (x_1^0 * k, \ldots, x_{(s-1)}^0 * k, y)$ – это частное решение уравнения $a_1 * x_1 + \ldots + a_s * x_s = d_s$.

Далее, если $\vec{e}_1, \ldots, \vec{e}_k$ – имеющийся базис решений для уравнения $a_1 * x_1 + \ldots + a_{(s-1)} * x_{(s-1)} = 0$, то базис решений для уравнения $a_1 * x_1 + \ldots + a_s * x_s = 0$ составляют следующие векторы

$$(\vec{e_1}, 0), \ldots, (\vec{e_k}, 0), (x_1^0 * \frac{(-a_s)}{d_s}, \ldots, x_{(s-1)}^0 * \frac{(-a_s)}{d_s}, \frac{d_{(s-1)}}{d_s})$$

Положим $s := s + 1$.

3 Если $s = n + 1$, то построенное частное решение $\vec{sol}$ и базис решений однородного уравнения подаем на выход алгоритма `PRIME`;
иначе переходим к п.2.

3.3. Алгоритм приведения к треугольному виду.

`Tr_M` – это алгоритм Гаусса приведения к треугольному виду. Это рекурсивный алгоритм. Рекурсия проводится по числу строк и столбцов матрицы, поданной на вход алгоритма.

Алгоритм `Tr_M`:

1 Если все элементы первого столбца равны нулю, то запоминаем первую строку матрицы; удаляем первые строку и столбец матрицы; применяем к полученной матрице алгоритм `Tr_M`.

2 Находим самую верхнюю строку с ненулевым первым элементом.

3 Запоминаем эту строку.

4 К этой строке и всем оставшимся строкам применяем алгоритм `ZERO`, устанавливающий нули в первом столбце матрицы.

5 У матрицы, полученной алгоритмом `ZERO`, удаляем первый столбец.

6 Если у полученной матрицы исчерпаны строки или столбцы,
то все запомненные ранее строки выдаем в качестве результата.
Иначе переходим к п.1.

Алгоритм `ZERO`:

На вход алгоритма подается строка `str` и матрица. С помощью этой строки делаем нулевым первый столбец матрицы.

1 Если первая строка матрицы начинается с нуля, то запоминаем эту строку и применяем алгоритм `ZERO` к матрице без первой строки.

2 Пусть
$str1$ – первая строка входной матрицы;
el – первый элемент str;
$el1$ – первый элемент $str1$;
lcm – НОК$(el, el1)$.

3 Запоминаем строку $str1 * (lcm/el1) - str * (lcm/el)$, где через $*$ обозначено покомпонентное умножение строки на элемент кольца, а через `"-"` – покомпонентное вычитание строк.

4 Удаляем у матрицы строку $str1$.

```
._  SOLVS( ( (1 2 3 4) (0 0 2 6) (1 2 3 4) ) );
@: -5 0 3 ((-2 1 0))

._
```

Рис. 1. Пример решения системы.

Если все строки матрицы исчерпаны, то все ранее запомненные строки подаем на выход алгоритма ZERO.

Иначе переходим к п.1.

3.4. Дополнительные алгоритмы.

К дополнительным алгоритмам мы относим следующее: умножение двух матриц, транспонирование матрицы, покомпонентное применение заданной операции к строке и заданному элементу, НОД двух элементов, НОД всех элементов строки и операция деления в евклидовом кольце. Описание соответствующих алгоритмов мы не приводим ввиду их тривиальности. См. п.4.4 и п.5 (`module service`).

§ 4. Описание программы

Решение системы **(I)** осуществляется вызовом функции SOLVS. В качестве аргумента функции SOLVS нужно подать следующий аппликативный терм:

$$\begin{array}{l}\texttt{(}\ \texttt{(}\ \mathtt{a}_{1,1}\ \ldots\ \mathtt{a}_{1,n}\quad \mathtt{b}_1\ \texttt{)}\\ \quad\ldots\ldots\ldots\ldots\ldots\\ \quad\texttt{(}\ \mathtt{a}_{m,1}\ \ldots\ \mathtt{a}_{m,n}\ \mathtt{b}_m\ \texttt{)}\\ \texttt{)}\end{array}$$

Программа представит решение в виде списка

$$\mathtt{x}_1^0\ \ldots\ \mathtt{x}_n^0\ \texttt{(}\ \mathtt{e}_1\ \ldots\ \mathtt{e}_k\ \texttt{)}\ ,$$

где $\mathtt{x}_1^0$, ..., $\mathtt{x}_n^0$ – частное решение системы, а $\mathtt{e}_i$ – базис решений однородной системы уравнений. Каждое $\mathtt{e}_i$ имеет вид

$$\texttt{(}\ \mathtt{e}_{i,1}\ \ldots\ \mathtt{e}_{i,n}\ \texttt{)}\ .$$

Если система имеет единственное решение $\mathtt{x}_1^0$, ..., $\mathtt{x}_n^0$ (т.е. соответствующая однородная система не имеет ненулевых решений), то результатом программы будет список $\mathtt{x}_1^0\ \ldots\ \mathtt{x}_n^0\ \texttt{()}$.

На рис.1 приведен фрагмент экрана дисплея при решении системы уравнений

$$\begin{cases} x_1 + 2x_2 + 3x_3 = 4\\ \qquad\quad 2x_3 = 6\\ x_1 + 2x_2 + 3x_3 = 4\end{cases}$$

над кольцом целых чисел.

Вся программа для решения системы уравнений состоит из четырёх модулей в соответствии с четырьмя частями алгоритма п.3. На рис. 2-5 схематически представлены зависимости между основными функциями программы.

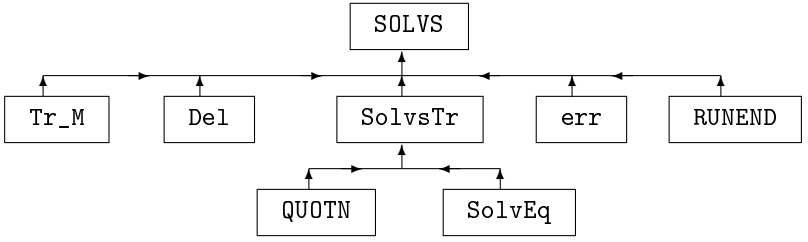


Рис. 2. Головные функции программы.

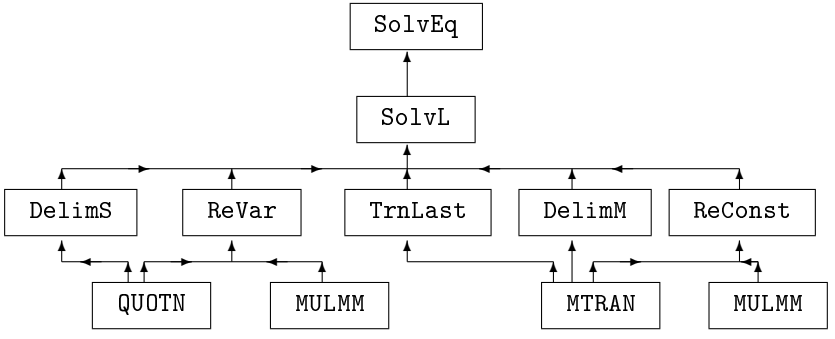


Рис. 3. Основные функции реализации верхнего уровня алгоритма.

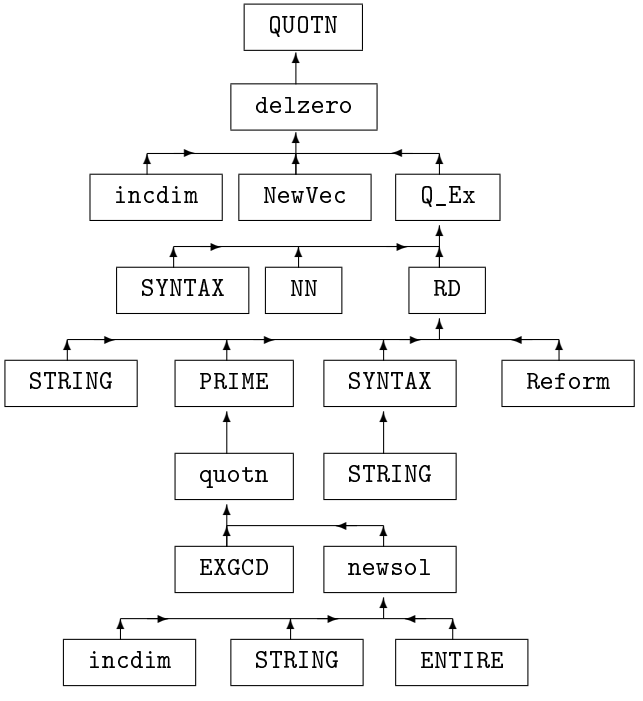


Рис. 4. Основные функции реализации алгоритма QUOTN.

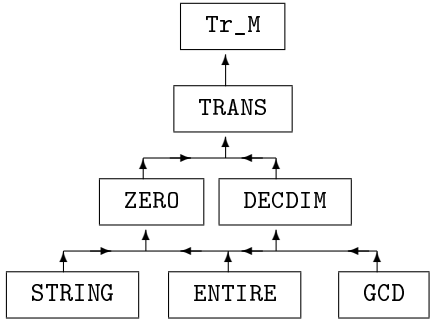


Рис. 5. Основные функции реализации алгоритма Tr_M.

Ниже для каждой из основных функций мы опишем входные и выходные данные и действия, выполняемые этой функцией.

4.1. Основные функции верхнего уровня алгоритма SOLVS.

Функция SOLVS – головная функция программы.

Функция RUNEND – встроенная функция системы FLAC. Если в системе во время выполнения программы возникает ошибка, то прекращается работа функции, в которой возникла ошибка, и начинает работать ближайшая (по стеку) функция RUNEND. Если возвращение в функцию RUNEND произошло без ошибки, то результат RUNEND – список термов:

$$0\ \texttt{N(}\ \ldots\ \texttt{)}\ ,$$

где «внутри» стоит аргумент функции RUNEND. Если возвращение происходит с ошибкой, то результат RUNEND – список термов:

$$n\ \texttt{ERR(}\ \ldots\ \texttt{)}\ ,$$

где n – код ошибки, первый аргумент ERR – это вызов функции, приведшей к ошибке, а остальные – аргументы RUNEND в момент ошибки. Кроме функции RUNEND, в системе имеется функция SYNTAX(), генерирующая синтаксическую ошибку (и тем самым приводящая к вызову функции RUNEND). Пара RUNEND-SYNTAX используется, когда по каким-либо причинам нужно срочно прекратить работу программы. В нашей программе это применяется в том случае, когда оказывается, что система не имеет решения. В этом случае нужно закончить работу программы.

Функция err обрабатывает результат функции RUNEND.

Функция Del удаляет из матрицы системы строки из нулевых элементов. Реализует шаг 2 алгоритма SOLVS.

Функция SolvTr решает систему уравнений с верхнетреугольной матрицей.

Функция SolvEq выполняет шаги 3-5 алгоритма SOLVS. Рекурсивная функция. На каждом шагу рекурсии имеем:
входные данные:

- первые $(i+1)$ строк системы уравнений – (#M &str &str1);
- общее решение системы уравнений, начиная с $(i+1)$-го – #sol &base;

выходные данные:

- первые i строк системы;
- общее решение системы уравнений, начиная с i-го.

Первый раз функция `SolvEq` вызывается со следующими аргументами: матрица всей системы; общее решение последнего уравнения системы. Результат работы этой функции – общее решение исходной системы уравнений.

Функция `SolvL` выполняет шаги 4-5 алгоритма `SOLVS`. Эта функция выбирает решения i-го уравнения среди всех решений системы уравнений, начиная с $(i+1)$-го.

Входные данные:

- строка `( a`$_{i,s_i}$ ... `a`$_{i,n}$ `b`$_i$ `)` i-го уравнения – `&str`;
- строка `( a`$_{(i+1),s_{(i+1)}}$ ... `a`$_{(i+1),n}$ `b`$_{(i+1)}$ `)` $(i+1)$-го уравнения – `&str1`;
- общее решение системы всех уравнений, начиная с $(i+1)$-го – `#sol &base`.

Выходные данные:

- общее решение системы уравнений, начиная с i-го.

Функция `DelimS` делит строку коэффициентов i-го уравнения на две части. Во вторую часть попадают ровно столько последних коэффициентов i-го уравнения, сколько коэффициентов у $(i+1)$-го уравнения. В первую часть попадают все первые коэффициенты i-го уравнения, не попавшие во вторую часть.

Выходные данные:

- `( a`$_{i,s_i}$ ... `a`$_{i,(s_{(i+1)}-1)}$ `)` – `&str`;
- `( a`$_{i,s_{(i+1)}}$ ... `a`$_{i,n}$ `b`$_i$ `)` – `&old`.

Функция `ReVar` подставляет в i-е уравнение решение системы, начиная с $(i+1)$-го уравнения, и решает полученное уравнение. Эта функция вместе с функцией `DelimS` реализует шаг 4 алгоритма `SOLVS`.

Входные данные:

- строка коэффициентов i-го уравнения, разбитая на две части (см. функцию `DilimS`):
  `( a`$_{i,s_i}$ ... `a`$_{i,(s_{(i+1)}-1)}$ `)` – `(#str)`;
  `( a`$_{i,s_{(i+1)}}$ ... `a`$_{i,n}$ `b`$_i$ `)` – `(#old &right)`;
- решение системы уравнений, начиная с $(i+1)$-го – `#sol &base`.

Работа функции: общее решение системы уравнений, начиная с $(i+1)$-го, представлено в виде частного решения `#sol` $= x^0_{i+1} \dots x^0_n$ и линейной комбинации векторов из `&base` с произвольными коэффициентами $C_1, \dots, C_k$. Подставляя это общее решение в i-е уравнение, получаем уравнение, на «новые» переменные

$$x_{s_i}, \dots, x_{(s_{(i+1)}-1)}$$

и «старые» константы $C_1, \dots, C_k$. Строка коэффициентов этого уравнения получается умножением матрицы `( (#old) )` на матрицу `&base`. Свободный член получается аналогичным умножением матрицы `( (#old) )` на «старое» частное решение `#sol`.

Выходные данные – общее решение полученного уравнения.

Это общее решение выражает «новые» переменные

$$x_{s_i}, \dots, x_{(s_{(i+1)}-1)}$$

и «старые» константы $C_1, \dots, C_k$ через «новые» константы.

Функция `Trn_Last` записывает по строкам матрицу `&base` базиса решений однородного уравнения, полученного функцией `ReVar`.

Функция `DelimM` разделяет решение уравнения из функции `ReVar` на две части.

Выходные данные:

- «новые» координаты частного решения $x^0_{s_i}, \ldots, x^0_{(s_{(i+1)}-1)}$ – `(#sol)`;
- «старые» константы в частном решении $C^0_1, \ldots, C^0_k$ – `&const`;
- строки матрицы базиса решений однородного уравнения, отвечающие «новым» переменным (матрица B) – `&base`;
- строки матрицы базиса решений однородного уравнения, отвечающие «старым» константам – `&constbas`.

Функция `ReConst` выражает решение системы уравнений, начиная с i-го, через «новые» константы. Часть решения уравнения из `ReVar`, отвечающая «новым» переменным, остается без изменений. А часть решения, выражающая «старые» константы через «новые», подставляется в решение системы, начиная с $(i+1)$-го уравнения. При вычислении частного решения используются вспомогательные функции `NB` и `ADDS`. При вычислении базиса решений важно различать три случая:

- системы уравнений, начиная с $(i+1)$-го и с i-го, имеют единственные решения.
- система уравнений, начиная с $(i+1)$-го, имеет единственное решение, а система уравнений, начиная с i-го, имеет непустой базис решений однородной системы («старых» констант нет, а «новые» есть).
- система уравнений, начиная с $(i+1)$-го, имеет непустой базис решений однородной системы.

Эти три случая разбираются вспомогательной функцией `NM`.

4.2. Основные функции реализации алгоритма `QUOTN`.

Функция `QUOTN` находит общее решение одного уравнения.

Функция `delzero` удаляет лишние нули из коэффициентов уравнения, увеличивая при этом базис пространства решений однородного уравнения. Это шаг 1 алгоритма `QUOTN`.

Функция `incdim` добавляет к каждому базисному вектору нулевую координату.

Функция `NewVec` порождает новый базисный вектор.

Функция `Q_Ex` реализует шаг 2 алгоритма `QUOTN`. Эта функция разбирает следующие ситуации:

- все коэффициенты (включая свободный член) равны нулю. В этом случае частное решение – нулевое, а матрица базиса – единичная;
- коэффициенты уравнения равны нулю, а свободный член – нет. В этом случае решения не существует.
- уравнение имеет ровно один ненулевой коэффициент. В этом случае решение существует, если этот коэффициент делит свободный член.
- в остальных случаях запускается функция `RD`.

Функция `SYNTAX` – встроенная функция системы FLAC. Функция генерирует синтаксическую ошибку и приводит к прекращению работы программы.

Функция RD проверяет, делится ли свободный член уравнения на НОД(всех коэффициентов) – &gcd. Если не делится, то решений нет. Если делится, то разделим все коэффициенты и свободный член на &gcd и к полученному уравнению применим функцию PRIME.

Функция Reform умножает частное решение, полученное функцией PRIME, на свободный член, деленный на &gcd. Тем самым получаем общее решение исходного уравнения. Эта функция вместе с функцией RD реализует шаг 3 алгоритма QUOTN.

Функция PRIME решает уравнение вида

$$a_1 * x_1 + \ldots + a_n * x_n = \text{НОД}(a_1 \ \ldots \ a_n) \ ,$$

причем a_1 не равно нулю.

Функция quotn реализует шаги 2-3 алгоритма PRIME. Рекурсивная функция. На каждом шагу рекурсии имеем:
входные данные:

- НОД$(a_1 \ \ldots \ a_{s-1})$ – &D;
- частное решение уравнения
$$a_1 * x_1 + \ldots + a_{s-1} * x_{s-1} = \text{НОД}(a_1 \ \ldots \ a_{s-1}) - \text{\#sol};$$
- базис решений уравнения
$$a_1 * x_1 + \ldots + a_{s-1} * x_{s-1} = 0 - \text{\&base};$$
- строка оставшихся коэффициентов $(a_s, \ldots, a_n)$ – (&el #str);

выходные данные:

- НОД$(a_1 \ \ldots \ a_s)$;
- частное решение уравнения
$$a_1 * x_1 + \ldots + a_s * x_s = \text{НОД}(a_1 \ \ldots \ a_s) \ ;$$
- базис решений уравнения
$$a_1 * x_1 + \ldots + a_s * x_s = 0 \ ;$$
- строка коэффициентов $(a_{s+1}, \ldots, a_n)$.

Первый раз эта функция вызывается с аргументами: a_1, единица кольца, соответствующий базис, $(a_2, \ldots, a_n)$. Оканчивая работу, функция выдаст результат, который и есть общее решение уравнения

$$a_1 * x_1 + \ldots + a_n * x_n = \text{НОД}(a_1 \ \ldots \ a_n).$$

Функция EXGCD – расширенный алгоритм Евклида. Функция от двух аргументов D и el. Эта функция находит частное решение уравнения

$$D*\text{\&x}+el*\text{\&y = НОД}(D, el).$$

Ее результат

$$\text{НОД}(D, el) \ \text{\&x \&y}.$$

Функция newsol реализует шаг 2 алгоритма PRIME.
Входные данные:

- результат EXGCD(НОД$(a_1 \ \ldots \ a_{s-1})$ a_s) – &D &K &Y ;
- частное решение $x_1^0, \ldots, x_{s-1}^0$ уравнения
$$a_1 * x_1 + \ldots + a_{s-1} * x_{s-1} = \text{НОД}(a_1 \ \ldots \ a_{s-1}) - \text{\#sol};$$
- a_s – &el ;
- НОД$(a_1 \ \ldots \ a_{s-1})$ – &oldD ;
- базис решений уравнения
$$a_1 * x_1 + \ldots + a_{s-1} * x_{s-1} = 0 - \text{(\#base)};$$

Выходные данные:

- НОД($a_1 \ldots a_s$);
- частное решение уравнения
$$a_1 * x_1 + \ldots + a_s * x_s = \text{НОД}(a_1 \ldots a_s);$$
- базис решений уравнения
$$a_1 * x_1 + \ldots + a_s * x_s = 0;$$

4.3. Основные функции алгоритма TrM.

Функция Tr_M приведения матрицы к треугольному виду. Исходная матрица задается списком по строкам. В результате получаем матрицу, в которой список, отвечающий i-й строке, не содержит первые i нулей. Т.е. формат выходной матрицы – это список аппликативных термов, причем каждый аппликативный терм содержит на один элемент меньше чем предыдущий.

Функция TRANS реализует шаги 1-3 алгоритма Tr_M.

Функция DECDIM удаляет первый столбец матрицы.

Функция ZERO реализует алгоритм ZERO. Эта функция использует вспомогательные функции LCMS и EXCL, вычисляющие НОК первых элементов строк и разность $str * (lcm/el1) - str * (lcm/el)$ из 3-го шага алгоритма ZERO.

4.4. Дополнительные функции.

Дополнительные функции: MULMM, MTRAN, STRING, GCD, NN, ENTIRE. Работа всех этих функций понятна из их описания на FLACe (см. п.5 module service). Отметим только, что в функции умножения матриц MULMM требуется, чтобы первая матрица была задана в виде списка по строкам, а вторая – в виде списка по столбцам.

§ 5. Программа на FLACe

```
      /*****************************
      **                           **
      **      module SOLVS         **
      **                           **
      *****************************/

 module SOLVS;

 PORT(SOLVS
      QUOTN Tr_M
      zero  add sub
      MTRAN MULMM
     );

/*--------------- MAIN FUNCTION SOLVS -----------------------------*
 |                                                                  |
 |  FUNCTION SOLVS finds all solutions of a sytem of linear         |
 |    equations   (A1,1)X1 + ... + (A1,n)An = B1                    |
 |                ..............................                    |
 |                (Am,1)X1 + ... + (Am,n)Xn = Bm                    |
 |                                                                  |
 |  INPUT : ( (A1,1 ... A1,n B1) ... (Am,1 ... Am,n Bm) )           |
```

```
 |  OUTPUT:  X1 ... Xn  (e1 ... ek)                                    |
 |   where  ei - a basis of all solutions of the homogeneous system |
 *-------------------------------------------------------------------*/

  SOLVS((#M)) = SolvsTr(Del(Tr_M(#M))) err(RUNEND());

    err(#x ERR(SYNTAX(#y))) = absent;
    err(#y ERR(#x))         = #y |ERR(#x);
    err(#y N(#x))           = ;

/* -------------- Delete zero's rows ----------------- */

    Del(&str) = &str;
    Del(#M &str) = ISZERO( #M &str zero() () );

         ISZERO(#M (#str &zero) &zero (#old))
               = ISZERO(#M |(#str) &zero |(#old &zero));

         ISZERO(#M (#str &el) &zero (#old))
               = #M |(#str &el #old);

         ISZERO(#M () &zero &old) = Del(#M);

/* ------------------------------------------------- */

/* ---- Solution of a system in the triangular form ---- */

    SolvsTr(#M (#str)) = SolvEq( |(#M |(#str)) QUOTN(#str) );

       SolvEq((#M &str &str1) #sol &base)
             = SolvEq( |(#M &str) SolvL(&str &str1 #sol &base));

       SolvEq( &str #sol &base)       = #sol &base;

/* ------------------------------------------------- */

/* ---  Solution of a system of two equations with  --- */
     /* ---- the coefficients &str and &str1 -----*/

    SolvL(&str &str1 #sol &base)
         = ReConst(#sol &base
                   DelimM(() () &base
                          TrnLast(ReVar(DelimS(() &str &str1)
                                        #sol &base
                                       )
                                  )
                          )
                  );

    DelimS((#old) (#str &el) (#Cnt &Cnt))
          = DelimS(|(&el #old) |(#str) |(#Cnt));

    DelimS(&old &str ())  = &str &old;
```

```
 /*-------------------------------------------------------------*
  |     FUNCTION  ReVar substitutes a given variable from      |
  |               i-th equation  into (i-1)-th                 |
  *-------------------------------------------------------------*/

    ReVar((#str) (#old &right) #sol &base)
         = QUOTN(#str SBR(MULMM( |(|(#old)) &base ))
                 sub(&right SBR(MULMM( |(|(#old)) |(|(#sol)) )))
                );

        SBR((#X)) = #X;
        SBR() = ;

/* ------------------------------------------------------------ */

    TrnLast(#sol &base)  = #sol |(MTRAN(&base));

    DelimM((#const) (#constbas) (#Cnt &Cnt)
           #sol &sol (#base &base))
          = DelimM(|(&sol #const) |(&base #constbas)
                   |(#Cnt) #sol |(#base)
                  );

    DelimM(&const &constbas () #sol &base)
          = |(#sol) &const &base &constbas;

/* ------------------------------------------------------------ */

/*-------------------------------------------------------------------*
 |   FUNCTION ReConst expresses a solution through of new constants |
 |                                                                   |
 *-------------------------------------------------------------------*/

    ReConst(#oldsol &oldbase (#sol) &const (#base) &constbas)
           = #sol ADDS( |(#oldsol) NB(&oldbase &const) )
             |( MTRAN( |(#base NM( |(#oldsol) |(#base)
                                   &oldbase &constbas
                                 )
                       )
                     )
              );

      NB(&oldbase &const) = MULMM( |(&const) |(MTRAN(&oldbase)) );

      ADDS((#oldsol &el) (#const &el1))
                        = ADDS(|(#oldsol) |(#const)) add(&el &el1);
      ADDS((#oldsol) #x) = #oldsol;

      NM( &oldsol () () () ) = ;
      NM( (#oldsol) (#base) () () )
                  = Makestr(#oldsol |(Makestr(#base zero())) );
```

```
      NM( &oldsol &base &oldbase &constbas )
                  = MULMM(|(MTRAN(&oldbase)) |(MTRAN(&constbas)) );

      Makestr(&Cnt #Cnt &el) = &el Makestr(#Cnt &el);
      Makestr(&el) = ;

/* ------------------------------------------------------------- */

 end;


       /*****************************
       **                           **
       **      module QUOTN         **
       **                           **
       *****************************/

 module QUOTN;

 PORT(QUOTN STRING ENTIRE NN
      eualg mult zero unit sub

     );

/*-------------------------------------------------------------------*
 | FUNCTION QUOTN finds all solutions of a linear equation          |
 |     A1X1 + ... + AnXn = B                                        |
 |  INPUT : A1 ... An B                                             |
 |  OUTPUT: X1 ... Xn (e1 ... ek)                                   |
 |  where ei - a basis of all solutions of the homogeneous equation |
 *-------------------------------------------------------------------*/

 QUOTN(#str) = delzero(zero() |(#str) ());

 delzero(&zero (&zero #str &right) #zero (&b #base))
        = delzero(&zero |(#str &right) #zero &zero
                  |(incdim(&b #base) NewVec(&b unit()))
                 );
 delzero(&zero (&zero #str &right) ())
        = delzero(&zero |(#str &right) &zero |(|(unit())) );

 delzero(#x) = Q_Ex(#x);

    Q_Ex(&zero (&zero) #zero &base)   = #zero &base;
    Q_Ex(&zero (&right) #zero)        = SYNTAX();
    Q_Ex(&zero (&a &right) #zero &base) = QUOT1(&zero
                                                eualg(&right &a)
                                                #zero &base);
    Q_Ex(&zero (#str &right) #zero &base) = RD(NN(#str) #str &right
                                               |(#zero &base)
                                              );

          QUOT1(&zero &q &zero ())  = &q ();
```

```
          QUOT1(&zero &q &zero #zero (#base))
                                    = #zero &q |(incdim(#base));
          QUOT1(#x)  = SYNTAX();

    RD(&gcd #str &right &eq)
                       = Reform( absent(zero() eualg(&right &gcd))
                         PRIME(STRING(|(#str ) &gcd ENTIRE) &eq) );

          absent(&zero &q &zero) = &q;
          absent(#x)             = SYNTAX();

          Reform(&q #sol &base) = STRING(|(#sol) &q mult) &base;

/*-----------------------------------------------------------------*
 | FUNCTION PRIME solves a linear equation                         |
 |       with B = GCD(A1 ... An) = 1                               |
 *-----------------------------------------------------------------*/

  PRIME(&el #str (#zero (#base)))
       = #zero quotn(&el unit() |(incdim(#base)) |(#str));


/*-------------------------------------------------------------------*
 | FUNCTION  quotn realizes an algorithm solving an equation         |
 |           A1X1 + ... + AsXs = GCD(A1 ... As)                      |
 |                                                                   |
 |  Let Di = GCD(A1 ... Ai).                                         |
 |  Let X1, ..., Xn be a particular solution of the equation         |
 |             A1X1 + ... + As-1Xs-1 = Ds-1                          |
 |                                                                   |
 |   PARTICULAR  SOLUTION :                                          |
 |    Let K , Y  be a particular solution of the equation            |
 |              Ds-1K + AsY = Ds                                     |
 |    then X1*K , ... , Xn-1*K ,Y is a particular solution           |
 |    of the initial equation                                        |
 |                                                                   |
 |   BASIS :                                                         |
 |    Let e1 , ... , en-2 - a basis of all solutions of the equation |
 |            A1X1 , ... , As-1 Xs-1 = 0                             |
 |    then (e1 0) ... (en-2 0)                                       |
 |     (X1*(-As/Ds) ... Xs-1*(-As/Ds) Ds-1/Ds) is                    |
 |    a basis of all solutions of the equation A1X1 + ... + AsXs = 0 |
 *-------------------------------------------------------------------*/

 quotn(&D #sol &base (&el #str))
      = quotn( newsol(EXGCD(&D &el) #sol &el &D &base) |(#str) );

 quotn(&D #sol &base ()) = #sol &base;

    newsol(&D &K &Y #sol &el &oldD (#base))
          = &D STRING(|(#sol) &K mult) &Y
               |( incdim(#base)
                  |( STRING(|(#sol) MINUS(ENTIRE(&el &D)) mult)
```

```
                  ENTIRE(&oldD &D)
                )
            );

/* ---------------------------------------------------------- */

 incdim((#x) #y) = |(#x zero()) incdim(#y);
 incdim() = ;


 NewVec((#b) #x) = |( ListZero(#b) #x);

    ListZero(&b #b) = zero() ListZero(#b);
    ListZero()      = ;

 MINUS(&X)    = sub(zero() &X);

/*-------------------------------------------------------------*
 |  FUNCTION  EXGCD finds a particular solution of the equation |
 |                    Ax + By = GCD(A B)                       |
 |  INPUT : A B                                                |
 |  OUTPUT: GCD(A B) x y                                       |
 *-------------------------------------------------------------*/

  EXGCD(&A &B)
       = EXGCD1(zero() &A &B |(unit() zero()) |(zero() unit()) );

     EXGCD1(&zero &A &zero (#A12) &B12) = &A #A12;
     EXGCD1(&zero &A &B &A12 &B12)
           = EXGCD1( &zero EXGCD2(eualg(&A &B) &B &A12 &B12) );

        EXGCD2(&q &r &B (&A1 &A2) (&B1 &B2))
              = &B &r |(&B1 &B2) |(sub(&A1 mult(&q &B1))
                                   sub(&A2 mult(&q &B2))
                                  );

/* ---------------------------------------------------------- */

 end;

       /*****************************
       **                           **
       **      module  Tr_M         **
       **                           **
       *****************************/

 module Tr_M;

 PORT( GCD Tr_M STRING ENTIRE NN
       mult sub zero
     );

/*-------------------------------------------------------------------------*
```

```
|  FUNCTION Tr_M(#M) reduces a given matrix #M to the triangular form |
|  INPUT: (A11 ... A1n) ... (Am1 ... Amn)                            |
*--------------------------------------------------------------------*/

Tr_M()   = ;
Tr_M(#M) =  TRANS( () zero() #M);

TRANS( (#zerostr) &zero (&zero #str) #M )
     = TRANS( |( #zerostr |(&zero #str) ) &zero #M);

TRANS( (#zerostr) &zero (&x #str) #M )
     = |(&x #str)
       Tr_M( DECDIM(&zero #zerostr ZERO(&zero |(&x #str) #M)) );

TRANS( (&str #zerostr) &zero )
     = &str Tr_M( DECDIM(&zero #zerostr) );

TRANS( #x ) = ;

   DECDIM(&zero (&el #str))    = |(#str);
   DECDIM(&zero (&el #str) #M) = |(#str) DECDIM(&zero #M);
   DECDIM(&zero) = ;

ZERO(&zero &str (&zero #str1) #M)
    = |(&zero #str1) ZERO(&zero &str #M);
ZERO(&zero &str &str1 #M)
    = EXCL(&zero LCMS(&str &str1) &str &str1)
      ZERO(&zero &str #M);

ZERO(&zero &str) = ;

   LCMS((&Z #Z) (&W #W)) = &Z &W LCM(&Z &W);

      LCM(&X &Y)    = ENTIRE(mult(&X &Y) GCD(&X &Y));

   EXCL(&zero &Z &zero &ZW &X &Y) = &Y;
   EXCL(&zero &Z &W &ZW &X &Y)
                       = |(SUBS(
                               |(STRING( &Y ENTIRE(&ZW &W) mult))
                               |(STRING( &X ENTIRE(&ZW &Z) mult))
                               )
                          );

      SUBS((&X #X) (&Y #Y)) = sub(&X &Y)  SUBS(|(#X) |(#Y));
      SUBS(() ())           = ;

/*-------------------------------------------------------------*/

end;
```

```
      /****************************
      **                          **
      **      module service      **
      **                          **
      ****************************/

 module service;

 PORT(MULMM MTRAN STRING GCD NN ENTIRE
      zero add mult eualg
     );
/*-------------------------------------------------------------*
 |    FUNCTION MULMM  multiplies two given matrixes            |
 |      INPUT : ((A11 ... A1n) ... (Am1 ... Amn))              |
 |              ((B11 ... B1n) ... (Bm1 ... Bmn))              |
 |      OUTPUT:  (C11 ... C1n) ... (Cm1 ... Cmn)               |
 *-------------------------------------------------------------*/

 MULMM(&M ())    = ;
 MULMM((&str #M) &M1) = |(MULSM(&str &M1)) MULMM(|(#M) &M1);
 MULMM(() &M1)      = ;

    MULSM(&str (&str1 #M1))
                    = SCAL(zero() &str &str1) MULSM(&str |(#M1));
    MULSM(&str ())   = ;

        SCAL(&scal (&el #str) (&el1 #str1))
            = SCAL(add(&scal mult(&el &el1)) |(#str) |(#str1));
        SCAL(&scal () ()) = &scal;


/*-------------------------------------------------------------*
 |   FUNCTION   MTRAN  transpose matrix                        |
 |     INPUT : ((A11 ... A1n) ... (Am1 ... Amn))               |
 |     OUTPUT:  (A11 ... Am1) ... (A1n ... Amn)                |
 *-------------------------------------------------------------*/

 MTRAN(()) = ;
 MTRAN(&M) = COL(() () &M);

    COL( (#col) (#tail) ((&el #str) #M) )
        = COL(|(#col &el) |(#tail |(#str)) |(#M));

    COL( &col &tail () ) = &col COL( () () &tail);
    COL( &col &tail (() #M) ) = ;

/*-------------------------------------------------------------*/

 STRING((&X #X) &Y &F) = &F(&X &Y) STRING(|(#X) &Y &F);
 STRING(() &Y &F)      = ;

/*-------------------------------------------------------------*/
```

```
 GCD(&X &Y)    = GCD1(zero() &X &Y);

    GCD1(&Y &X &Y) = &X;
    GCD1(&Z &X &Y) = GCD1(&Z &Y REST(eualg(&X &Y)));

/*-------------------------------------------------------------*/

 NN(&X &Y #x) = NN(GCD(&X &Y) #x);
 NN(&X)       = &X;

/*-------------------------------------------------------------*/

 ENTIRE(&X &Y) = FIRST(eualg(&X &Y));

/*-------------------------------------------------------------*/

 end;
```

Список литературы

[1] К. А. Родосский, *Алгоритм Евклида*, Наука, Москва, 1988.

[2] В. Л. Кистлеров, "Принципы построения языка алгебраических вычислений FLAC", Препринт, ИПУ АН СССР, Москва, 1987.

[3] Е. А. Гайдар, И. Н. 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, **1**, Издательство Сборник, Переславль-Залесский, 2014(1988), 11–42.

[4] Е. А. Гайдар, И. Н. Игнатович, В. Ф. Козадой, А. П. Немытых, В. А. Пинчук, С. В. Чмутов, "Реализация системы программирования FLAC", Сборник трудов по функциональному языку программирования Рефал, **1**, Издательство Сборник, Переславль-Залесский, 2014(1988), 43–91.

[5] Э. Баннаи, Т. Ито, *Алгебраическая комбинаторика*, Мир, Москва, 1987.

[6] З. И. Боревич, И. Р. Шафаревич, *Теория чисел*, Наука, Москва, 1985.

[7] К. Айерленд, М. Роузен, *Классическое введение в современную теорию чисел*, Мир, Москва, 1987.

[8] М. М. Постников, *Введение в теорию алгебраических чисел*, Наука, Москва, 1982.

[9] Г. Эдварс, *Последняя теорема Ферма*, Мир, Москва, 1980.

[10] Ж. П. Серр, *Линейные представления конечных групп*, Мир, Москва, 1970. **Ниже дан список литературы, добавленный редактором сборника.**

[11] 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.

[12] S. V. Chmutov, E. A. Gaydar, I. M. Ignatovich, V. F. Kozadoy, A. P. Nemytykh, V. A. Pinchuk, *The symbol analytic transformation language FLAC: sources, executable modules*, ftp://www.botik.ru/pub/local/scp/flac/flac386.zip, 1991.

Нина А. Чмутова (Nina A. Chmutova)
Переславль-Залесский
E-mail: chmutova@yahoo.com

А. П. Немытых

Лекции по языку программирования Рефал

Мы представляем конспекты лекций по функциональному языку программирования Рефал. Лекции были прочитаны автором в 2006 году в г. Переславле-Залесском. Библ. 10 наим.

Содержание

§ 1. Данные языка программирования Рефал (Лекция №1)

РЕкурсивный Функциональный АЛгоритмический язык РЕФАЛ[1] был разработан В. Ф. Турчиным как универсальный метаязык программирования в 60-х годах. Первая реализация появилась в 1968 г. Рефал – строгий язык первого порядка, основанный на алгорифмах Маркова, имеет два неизоморфных конструктора; один из которых – приписывание (конкатенация) – ассоциативен, а другой – унарный – используется для построения структуры произвольного дерева. Ассоциативность приписывания превращает множество данных Рефала в моноид. Структура этого моноида отражается в синтаксисе программ и операционной семантике Рефала.

1.1. Уравнения в свободной полугруппе. *Моноидом* называется полугруппа с единицей.

Пусть M – непустое множество. Тогда *свободным моноидом*, порождённым M, называется множество всех конечных последовательностей (включая пустую последовательность) элементов из M с полугрупповой операцией, задаваемой приписыванием последовательностей, т. е.

$$\{a_1, \ldots, a_n\} \circ \{b_1, \ldots, b_m\} = \{a_1, \ldots, a_n, b_1, \ldots, b_m\}$$

Далее пустую последовательность будем обозначать *ничем* (отсутствием какого либо знака).

Рассмотрим свободный моноид, порождённый множеством букв Кириллицы и латинского алфавита. Назовём этот моноид *алфавитной полугруппой*. Элементы алфавитной полугруппы будем писать в одинарных кавычках. Например, `'ы'`. Слово, заключённое в одинарные кавычки, будем считать сокращением записи последовательности букв, входящих в это слово. Например,

`'полугруппа'` = `'п' 'о' 'л' 'у' 'г' 'р' 'у' 'п' 'п' 'а'`

Задача 1. Решить уравнения в алфавитной полугруппе (где x, y и z – переменные):

а). x `'п'` z = `'полугруппа'`;
б). $x^2 = xx =$ `'abababab'`;
в). x^2 `'a'` $y =$ `'aabaabaab'`;

Иногда удобно выделять последовательности, состоящие из одного элемента. Перед именами переменных с областью допустимых значений (О.Д.З.) – множеством одноэлементных последовательностей будем приписывать приставку «`s.`», перед именами переменных с О.Д.З., совпадающим со всей полугруппой, будем приписывать приставку «`e.`» Например: `s.x`, `e.y` .

[1]Название, данное В. Ф. Турчиным, – алгоритмический язык рекурсивных функций. Мы предпочитаем другую расшифровку этого сокращения, избегая перестановки. Не существует общепринятой традиции, какими буквами – прописными (и сколькими) или строчными – писать название языка. В. Ф. Турчин также использовал разные варианты синтаксиса этого названия. Мы будем писать Рефал, делая редкие исключения.

Задача 2. Решить уравнения в алфавитной полугруппе

а). `e.x s.y 'п' e.z = 'полугруппа'`;
б). `s.y e.x`^{3} `s.z = 'abababab'`;
в). `e.u s.z`^{2} `e.x`^{2} `e.y = 'aabaabaab'`;

Теперь рассмотрим уравнения с параметрами. Аналогично переменным, параметры будут двух типов, соответствующих введённым типам переменных. Чтобы отличать параметры от переменных, мы будем помечать параметры знаком диез «#». Например: `#s.p` – параметр типа «s» с именем «p», `#e.q` – параметр с именем «q» и О.Д.З., совпадающим со всей полугруппой.

Задача 3. Решить уравнения с параметрами в алфавитной полугруппе

а). `e.x s.y 'п' e.z = 'пол' #s.p 'группа'`;
б). `e.x 'a' = 'a' #e.p`;
в). `e.y 'a' e.x`^{2} `= 'ab' #s.p #s.p 'abaab'`;

Задача 4. Рассмотрим свободный моноид, порождённый множеством из `'0'` и `'1'`. Придумать в этом моноиде уравнение с параметрами `#s.p`, `#s.q`, `#s.r`

`... = #s.p #s.q #s.r`

такое, что одно из его переменных есть `s.d` и решение этого уравнения имеет вид

`..., s.d = '0', ...`

тогда и только тогда, когда последовательность `#s.p #s.q #s.r` содержит больше нулей, чем единиц.

1.2. Системы уравнений в свободной полугруппе. Рассмотрим систему из двух уравнений в алфавитной полугруппе

$$\begin{cases} \texttt{e.x e.y e.z = 'abbabaab'} \\ \texttt{e.y s.u = 'bab'} \end{cases}$$

Скобки (и) не принадлежат алфавиту. Алфавитная полугруппа ассоциативна, что позволяет нам опускать эти скобки, не указывая порядок исполнения операции приписывания в полугруппе. Таким образом, скобки (и) можно использовать как метасимволы – для иных целей. Воспользуемся этим замечанием, чтобы формально записать систему уравнений более кратко:

e.x e.y e.z (e.y s.u) = 'abbabaab' ('bab')

Аналогично можно записать любую систему уравнений в алфавитной полугруппе. Например, систему

$$\begin{cases} \texttt{e.y}_1\ \texttt{'н' e.n 'учи' e.y}_2 = \texttt{'Какая' 'ночь' 'Мороз' 'трескучий'} \\ \texttt{e.z}_1\ \texttt{'н' e.m 'учи' e.z}_2 = \texttt{'На' 'небе' 'ни' 'единой' 'тучи'} \\ \texttt{e.x}_1\ \texttt{s.k e.x}_2\ \texttt{s.k e.x}_3 = \texttt{'Как' 'шитый' 'полог' 'синий' 'свод'} \end{cases}$$

можно записать так

```
e.y1 'н' e.n 'учи' e.y2 (e.z1 'н' e.m 'учи' e.z2) (e.x1 s.k e.x2 s.k e.x3) =
'Какая' 'ночь' 'Мороз' 'трескучий'
('На' 'небе' 'ни' 'единой' 'тучи') ('Как' 'шитый' 'полог' 'синий' 'свод')
```

или так

```
e.y1 'н' e.n 'учи' e.y2 (e.z1 'н' e.m 'учи' e.z2 (e.x1 s.k e.x2 s.k e.x3))
=
'Какая' 'ночь' 'Мороз' 'трескучий'
('На' 'небе' 'ни' 'единой' 'тучи' ('Как' 'шитый' 'полог' 'синий' 'свод'))
```

Задача 5. Решите данные выше уравнения.

Пусть G – свободный моноид. Тогда *древовидной полугруппой* D, порождённой G, назовём наибольшее множество, которое можно получить из элементов G посредством конечного числа применений бинарной операции приписывания и унарной операции заключения в скобки. То есть, $G \subset D$ и

1. если $d_1 \in D$ и $d_2 \in D$, то $d_1 \circ d_2 = d_1 \ \ d_2 \in D$;
2. если $d \in D$, то $(d) \in D$

Задача 6. Докажите, что древовидная полугруппа является полугруппой с единицей относительно операции приписывания.

Пусть Ω – алфавитная полугруппа, тогда правая часть последнего, приведённого выше, уравнения принадлежит к древовидной полугруппе, порождённой Ω. Выражение

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

принадлежит любой древовидной полугруппе. Более того, любая правильная скобочная структура (выражение, состоящее только из скобок) принадлежит любой древовидной полугруппе.

Задача 7. *(*числа Каталана): Сколько существует различных правильных скобочных структур из n пар скобок?

1.2.1. *Леса корневых деревьев.* *Ориентированным графом* будем называть тройку $\Gamma = \{V, E, I\}$, состоящую из конечного множества *вершин* V, конечного (возможно, пустого) множества *рёбер* E и *отображения инцидентности* $I : E \to V \times V$, сопоставляющего каждому ребру упорядоченную пару вершин (*начало* и *конец* ребра), которые это ребро соединяет.

Последовательность ребер $e_1, \ldots, e_n$ ориентированного графа назовем *путем*, если начало каждого последующего ребра является концом предыдущего.

Корневым деревом назовем ориентированный граф Γ с выделенной вершиной r (*корнем*), в которую не входит ни одно ребро и из которой существует единственный путь в любую другую вершину. Кроме того, для каждой вершины из Γ множество исходящих из неё рёбер упорядочено.

Задача 8. Докажите, что в корневом дереве всякая вершина, отличная от корня, имеет ровно одно входящее в неё ребро.

Вершина называется *узлом*, если из неё выходит не менее одного ребра. Множество вершин, не являющихся узлами, разобьём на два непересекающихся множества; элементы одного из них назовём *листьями*, элементы другого – *срезами*. Каждому листу припишем некоторое имя.

Пусть дано множество имён $\mathcal{A}$. *Лесом* корневых деревьев $F_{\mathcal{A}}$ назовём множество всех конечных последовательностей (включая пустую последовательность) корневых деревьев, листья которых поименованы элементами из $\mathcal{A}$.

Деревья будем изображать растущими от корня вниз. Каждому выражению с правильной скобочной структурой можно сопоставить лес корневых деревьев. Например,

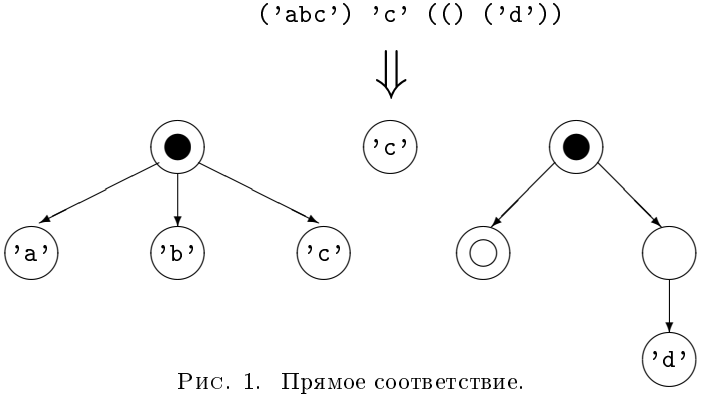


Рис. 1. Прямое соответствие.

Обратно, каждому лесу корневых деревьев можно сопоставить выражение с правильной скобочной структурой.

Задача 9. Опишите обратное соответствие.

1.3. Домашнее задание.

Задача 10. Решить уравнения в алфавитной полугруппе (где x и y переменные)

а). y `'a'` $x^2 =$ `'abbabaab'`;
б). x `'a'` $=$ `'a'` x.

Задача 11. Решить уравнения в алфавитной полугруппе

а). `e.y s.z e.x`^{2} `= 'abbabaab'`;
б). `s.x 'a' = 'a' s.x e.y`.

Задача 12. Решить уравнения с параметрами в алфавитной полугруппе

а). `s.y e.x`^{3} `s.z = 'abab' #e.p`;
б). `e.x`^{2} `'a' e.y = #e.p #e.p 'aab'`.

Задача 13. Дан свободный моноид G, порождённый множеством из букв `'a'` и `'b'`. В G рассматривается уравнение с параметрами:

`e.y e.x`^{2} `e.z = #s.p #s.q #s.r #e.u`

Докажите, что это уравнение имеет решение такое, что длина последовательности `e.x` больше нуля.

Задача 14. Последовательность w_n определяется индуктивно по правилу:

$$w_0 = \texttt{'a'}, w_1 = \texttt{'ab'}, w_{n+2} = w_{n+1}\ \ w'_{n+1},$$

где w'_{n+1} получается из w_{n+1} заменой всех букв `'a'` на `'b'` и всех букв `'b'` на `'a'`. Например, $w_2 =$ `'abba'`, $w_3 =$ `'abbabaab'`. Пусть $u = w_k$ для некоторого k. Решить уравнения: 1) `e.y` `'a'`^{3} `e.z =` u ; 2) `e.y` `'b'`^{3} `e.z =` u .

Задача 15. Установите на вашем компьютере одну из реализаций диалектов языка Рефал.

§ 2. Функции индуктивные по Кушниренко (Лекция №2)

РЕФАЛ – РЕкурсивный Функциональный АЛгоритмический, по определению, есть рекурсивный язык программирования. Это означает, что основным выразительным инструментом программирования на Рефале является рекурсия (от recursion – повторение). Понятие рекурсии тесно связано с понятием индукции по структуре данных: Рефал-программу можно часто понимать как доказательство некоторого утверждения о данных методом математической индукции. Мы рассмотрим индукцию по данным, построенным с помощью конструктора приписывания.

2.1. Функции индуктивные по Кушниренко. Скажем, что функция $F : M \to K$ *конструктивна*, если существует алгоритм, который $\forall x \in M$ вычисляет значение $F(x)$. Ниже под функциями мы будем понимать только конструктивные функции.

Понятие алгоритма является *начальным* (*неопределяемым*), как, например, понятия точки и множества в геометрии.

Рассмотрим свободный моноид G, порождённый некоторым множеством M. Пусть N – некоторое множество. Функция $f : G \to N$ называется *индуктивной*, если её значение на последовательности $x_1\ \ldots\ x_n$ можно вычислить по её значению на последовательности $x_1\ \ldots\ x_{n-1}$ и по x_n, то есть существует функция $F : N \times G \to N$, для которой

$$f(x_1\ \ldots\ x_n) = F(f(x_1\ \ldots\ x_{n-1}), x_n)$$

Отметим, что 1) f есть функция от одного аргумента, который является конечной последовательностью, и 2) длина n этой последовательности не дана, хотя и дан её последний элемент x_n (если он существует).

Таким образом, мы получаем конкретный алгоритм вычисления значений функции f, если известно её значение $f()$ на пустой последовательности.

Например, $\sum_{i=1}^{n} x_i$ – сумма всех членов конечной числовой последовательности является индуктивной функцией. Мы считаем, что сумма членов пустой последовательности равна 0.

Не все функции $f : G \to N$ являются индуктивными.

Индуктивным расширением функции $f : G \to N$ называется такая *индуктивная* функция $g : G \to K$, что существует функция $Q : K \to N$ такая, что

$$f(x_1 \ \ldots \ x_n) = Q(g(x_1 \ \ldots \ x_n))$$

Мы снова получаем конкретный алгоритм вычисления значений функции f.

Задача 1. Определить, являются ли следующие функции индуктивными, доопределив их, если необходимо, естественным образом на пустой последовательности. Указать индуктивные расширения для неиндуктивных функций:

а). Функция, сопоставляющая данной последовательности $x_1 \ \ldots \ x_n$ перевёрнутую последовательность: $x_n \ \ldots x_1$.

б). Дана последовательность $y_1 \ \ldots \ y_k$ десятичных цифр, где $y_1 \neq 0$, если $k > 1$. Функция, сопоставляющая последовательности $x_1 \ \ldots \ x_n$ десятичных цифр, где $x_1 \neq 0$ (если $n > 1$), произведение целых чисел $\overline{x_1 \ldots x_n} * \overline{y_1 \ldots y_k}$.

в). Среднее арифметическое последовательности целых чисел.

г). Число элементов последовательности целых чисел, равных её максимальному элементу.

д). Максимальная длина монотонного (неубывающего или невозрастающего) отрезка из идущих подряд элементов в последовательности целых чисел.

е). Максимальная длина возрастающей подпоследовательности данной последовательности целых чисел.

2.2. Примеры программ на языке Рефал. *Языком программирования* L называется тройка $\{D, P, S\}$, где D – множество *данных*, P – множество *программ* и функция *семантики* $S : P \times D \to D_\perp$ (где $\perp \notin D$ и $D_\perp ::= D \cup \{\perp\}$), описывающая алгоритм вычисления конкретной программы $p \in P$ на конкретном данном $d \in D$, то есть $S(p, d) = p(d)$. Причём $p(d) = \perp$ тогда и только тогда, когда p попадает в состояние аварийной остановки или в бесконечный цикл.

Пусть фиксирован некоторый алфавит $\mathcal{A}$, элементы которого мы будем называть символами. Множество данных Рефала D может быть определено индуктивно:

- если $\mathtt{d}_1 \in D$, тогда $\mathtt{(d}_1\mathtt{)} \in D$;
- если $\mathtt{d}_1 \in D$ и $\mathtt{d}_2 \in D$, тогда $\mathtt{d}_1 \ \mathtt{d}_2 \in D$;
- пустая последовательность является данным;
- $\forall a \in \mathcal{A} \Rightarrow a \in D$.

Подобные определения принято кратко записывать так:

$\mathtt{d} \ ::= \ \mathtt{d}_1 \ \mathtt{d}_2 \ | \ \mathtt{(d}_1\mathtt{)} \ | \ a \ | \ \mathit{empty}$

$\mathit{empty} \ ::=$

, где знак «::=» читается «по определению равно», а знак «|» есть логическая связка «или».

Алфавит Рефала (множество *символов*) мы будем описывать по ходу дела – постепенно расширяя его. Данные Рефала принято называть *объектными выражениями*. *Объектным термом* Рефала называется символ или объектное выражение, заключённое в скобки. Таким образом, любое объектное

Рефал-выражение есть конечная последовательность объектных Рефал-термов (включая пустую последовательность).

Любой читабельный символ, который позволяет ввести клавиатура компьютера, заключённый в одинарные кавычки, есть символ Рефала. Например: `';'`, `' '`, `'b'`. Исключение составляет сама одинарная кавычка; её необходимо экранировать: `'\''`.

Слово называется *палиндромом*, если европейское прочтение этого слова (слева направо) совпадает с его арабским прочтением (справа налево). Слово «палиндром» в переводе с греческого языка означает бегущий вспять.

Предикатом на множестве M называется функция φ со значениями в двухэлементном множестве $\varphi : M \to \{$`'T', 'F'`$\}$. Здесь `'T'` означает истину (True), а `'F'` – ложь (False).

Ниже дано рекурсивное определение на Рефале предиката на множестве слов: его значение на данном конкретном слове-аргументе w_0 равно `'T'` тогда и только тогда, когда это слово есть палиндром.

```
Palindrome {
      =  'T';
 s.x  =  'T';
 s.x e.middle s.x = <Palindrome e.middle>;
 e.word = 'F';
}
```

Третье предложение сравнивает на равенство первую и последнюю буквы слова w_0 и, если они совпадают, w_0 есть палиндром тогда и только тогда, когда палиндромом является слово, полученное из w_0 отбрасыванием первой и последней буквы. Первые два предложения описывают *базисные случаи индукции*: первое – для слов чётной длины, второе – для слов нечётной длины. Процесс проверки заканчивается, так как на каждом шаге длина слова уменьшается и она ограничена снизу нулём.

Задача 2. Определить предикат палиндрома на языке Haskell и сравнить это определение с данным выше.

Понятие рекурсии тесно связано с понятием математической индукции.

Рассмотрим рекурсивное определение функции, переворачивающей данную последовательность термов. О.Д.З. переменной типа «`t.`» есть множество объектных Рефал-термов.

```
Reverse {
            = ;
 t.x e.rest = <Reverse e.rest> t.x;
}
```

Мы 1) переворачиваем последовательность, полученную из данной последовательности отбрасыванием первого элемента, и 2) приписываем к результату переворачивания сзади этот первый терм-элемент. Длина сокращённой последовательности на единицу меньше длины исходной, следовательно, повторяя процесс шаг за шагом, мы придём к пустой последовательности, результат переворачивания которой есть она сама.

Задача 3. Определить аналогичную функцию на языке Haskell и сравнить это определение с данным выше.

Задача 4. Привести пример программы, которая работает бесконечное время.

Решим на Рефале следующую задачу. Кузнечик прыгает по прямой линии на один метр, каждый раз случайно выбирая направление прыжка (вправо или влево). Будем наблюдать за ним, записывая его выбор последовательностью букв `'L'` (влево) и `'R'` (вправо). Определить, на каком расстоянии от начальной позиции будет находиться кузнечик, когда он остановится.

Если в последовательности прыжков есть рядом стоящие символы `'LR'` или `'RL'`, то они не влияют на ответ и их можно выкинуть из последовательности – далее повторить сокращения, в противном случае вся последовательность состоит из одинаковых букв; их число есть расстояние кузнечика от стартовой позиции – вправо или влево.

Буквальное повторение этого предписания на Рефале выглядит так:

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

Здесь в качестве имён переменных мы использовали числа. Отметим, что второе предложение можно переставить выше первого – определение останется правильным. Третье же предложение переставить нельзя.

Задача 5. Решить задачу про кузнечика на языке Haskell и сравнить это решение с данным выше.

2.2.1. *Как выполнить программу на языке Рефал.* Пусть дана программа p. Функция семантики $S(p, d) = p(d)$, по определению, может быть вычислена только тогда, когда фиксированы оба её аргумента. Программу мы уже фиксировали. Данное – второй аргумент функции семантики в Рефале – можно задать, например, посредством Рефал-определения с выделенным именем **Go**; именно с этого определения алгоритм, реализующий функцию семантики, начинает рассматривать Рефал-программу (принято говорить, что **Go** является *входной точкой* программы).

Чтобы подчеркнуть, что мы будем общаться с программой именно через эту функцию – явно укажем это в синтаксисе *ключевым словом* `$ENTRY`:

```
$ENTRY Go {
  = <Palindrome 'долог голод'>;
}
```

Мы зафиксировали аргумент предиката `Palindrome`, и теперь его значение может быть вычислено на вычислительной машине, но результат вычисления для нас останется неизвестным. Чтобы компьютер напечатал результат на экране, необходимо попросить его об этом:

```
$ENTRY Go {
  = <Prout <Palindrome 'долог голод'>>;
}
```

Здесь в правой части предложения стоит композиция двух действий: вычислить значение `Palindrome` и напечатать результат вычисления. В результате исполнения этой программы на экране компьютера появится строка: `T`.

2.3. Домашнее задание.

Задача 6. Определить, являются ли следующие функции индуктивными, доопределив их, если необходимо, естественным образом на пустой последовательности. Указать индуктивные расширения для неиндуктивных функций. Дать определение этих функций на Рефале.

а). Максимальный отрезок идущих подряд одинаковых элементов последовательности.

б). Даны две последовательности $x_1 \ \ldots \ x_n$ и $y_1 \ \ldots \ y_k$. Функция для всех $m \leqslant n$ сопоставляющая последовательности $x_1 \ \ldots \ x_m$ максимальное число $j \leqslant k$ такое, что последовательность $y_1 \ \ldots \ y_j$ есть подпоследовательность последовательности $x_1 \ \ldots \ x_m$.

Задача 7. Будем все слова записывать только строчными буквами. *Обобщённым палиндромом* назовём фразу (или последовательность фраз), которая становится палиндромом, если из неё выкинуть все знаки препинания и пробелы. Определить на Рефале предикат на множестве текстов, множество истинности которого совпадает с множеством обобщённых палиндромов.

Задача 8. Даны текст и строка. Определить на Рефале функцию, значение которой есть отрезок данного текста, непосредственно следующий за *первым вхождением* данной строки, если такое вхождение существует, и сам текст, если данная строка в него не входит.

Задача 9. Даны текст и строка. Определить на Рефале функцию, значение которой есть отрезок данного текста, непосредственно следующий за *последним вхождением* данной строки, если такое вхождение существует, и саму строку, если она в данный текст не входит.

Задача 10. В унарной системе счисления натуральные числа представляются счётными палочками. Например, `'III'` = 3. Определить на Рефале функции унарной арифметики: сложения, вычитания, умножения и деления с остатком двух натуральных чисел.

§ 3. Унарный конструктор построения дерева. Рефал-машина. (Лекция №3)

Рефал – строгий язык первого порядка – имеет два неизоморфных конструктора; один из которых – приписывание – ассоциативен, а другой – унарный –

используется для построения структуры произвольного дерева. Нас будет интересовать базисный Рефал (подмножество Рефала), в котором процесс вычисления программы может быть представлен в виде композиции простых действий, называемых шагами Рефал-машины.

Язык программирования называется *строгим* (или *аппликативным*), если значение функции $F(x, \ldots, y)$ не вычисляется, пока не вычислены все её аргументы $x, \ldots, y$. Примером нестрогого («*ленивого*») языка является язык программирования Haskell.

Задача 1.

а). Приведите пример программы p на языке Haskell, формальный перевод которой на Рефал (то есть изменяется только синтаксис, но не изменяется структура программы) существенно повышает её эффективность: позволяет вычислить значение $p(d_0)$ существенно быстрее и для этого вычисления понадобится существенно меньше оперативной памяти вычислительной машины.

б). Приведите пример программы p на языке Рефал, формальный перевод которой на Haskell существенно повышает её эффективность.

3.1. Парные скобки как унарный конструктор построения дерева. *Унарный конструктор* парных круглых скобок ($\texttt{data}_0$) является средством структурирования данных. Удобно думать о действии этого конструктора, как о помещении его аргумента (внутренности скобок) внутрь матрёшки: мы можем переносить матрёшку из одного места в другое как единое целое – не заботясь о том, что спрятано внутри неё; как и содержимое матрёшки, внутренность скобки имеет конечную *глубину*.

Хотя формально в Рефале все функции, по определению, одного аргумента (*унарны*) – конечной последовательности, конструктор скобок позволяет имитировать произвольное число аргументов. Например, функцию `F` от двух аргументов можно синтаксически представить так

$$\texttt{<F (}x_1 \ldots x_n\texttt{) (}y_1 \ldots y_m\texttt{)>}$$

или так

$$\texttt{<F }x_1 \ldots x_n\ \texttt{(}y_1 \ldots y_m\texttt{)>}$$

Функция сложения двух натуральных чисел, представленных в унарной системе счисления, может быть определена так:

```
Summa {
 (e.n) (e.m) = e.n e.m;
}
```

Аналогичным образом можно имитировать отображение из D в $D \times D$:

```
F {
.............
... = (x1  ... xn) (y1  ... ym);
}
```

Как мы знаем, каждое Рефал-данное представляет лес корневых ориентированных деревьев.

Задача 2. Определить на Рефале функцию, которая по данному лесу:

а). строит последовательность всех листьев этого леса, не изменяя порядок их следования;

б). строит последовательность всех листьев, пропуская повторные вхождения равных листьев, но не изменяя порядок их следования.

Расширим алфавит Рефал-символов: каждое *слово*, написанное в латинице, будем считать символом, если это слово не заключено в одинарные кавычки. Например,

- `refal` – символ языка Рефал, то есть входит в О.Д.З. `s`-переменной;
- `'refal'` – последовательность из пяти символов.

Каждое слово, состоящее из букв Кириллицы и латиницы, *заключённое в двойные кавычки*, есть, по определению, Рефал-символ. Причём `'b'` $\neq$ `b`, но `"b" = b`.

После нашего расширения алфавита Рефала, следующее стихотворение Валерия Брюсова будет обобщенным палиндромом (см. предыдущую лекцию №2).

```
"жестоко"  "раздумье"  '.'  "ночное"  "молчанье"
"качает"  "виденья"  "былого" ','
"мерцанье"  "встречает"  "улыбки"  "сурово" ','
                                    "страданье"
                    "глубоко"  '-'  "глубоко"  '!'
"страданье"  "сурово"  "улыбки"  "встречает"  '...'
"мерцанье"  "былого"  "виденья"  "качает"  '...'
"молчанье"  '.'  "ночное"  "раздумье"  "жестоко" '.'
```

Задача 3. Дано объектное Рефал-выражение d. Написать программу, заменяющую в d каждое слово `Haskell` на слово `Refal`.

Скелетом Рефал-выражения называется его скобочная структура. Например, скелет выражения `((refal) (()) ("выражение"))` есть `(() (()) ())`.

Задача 4. По данному объектному Рефал-выражению построить его скелет.

3.2. Абстрактная Рефал-машина. В *базисном* подмножестве Рефала программы представляют собой множество *определений-функций*. Каждое Рефал-определение есть непустая последовательность *предложений*, которые разделяются точкой с запятой. Каждое предложение имеет вид:

```
образец = правая_часть ;
```

Образец и *правую часть* определим индуктивно посредством формальной грамматики:

```
образец ::= терм-образец образец1 | empty
терм-образец ::= (образец) | СИМВОЛ | переменная
переменная ::= e.NAME | t.NAME | s.NAME
empty ::=

правая_часть ::= терм-правая_часть правая_часть1 | empty
```

```
терм-правая_часть ::= вызов_функции | (правая_часть) | СИМВОЛ
                    | переменная

вызов_функции ::= <NAME аргументы>
аргументы ::= правая_часть
```

Здесь *базисные понятия* мы написали прописными буквами. Существует дополнительное требование: множество переменных правой части предложения является подмножеством переменных образца этого предложения.

Все ранее рассмотренные нами программы принадлежат базисному Рефалу.

Рефал-машина реализует функцию семантики языка Рефал. Опишем *абстрактную машину базисного Рефала*. (Повсюду ниже, если не оговаривается иное, мы будем под словом Рефал иметь в виду базисный Рефал.)

Рабочая память Рефал-машины называется *полем зрения*.

Перед началом работы в поле зрения помещается вызов функции **Go**:

```
<Go >
```

Процесс работы Рефал-машины есть исполнение конечной последовательности её шагов.

Шагом Рефал-машины называется следующая последовательность действий:

1. Выбираем в поле зрения активный вызов функции `<F` $\mathtt{d}_0$`>` – алгоритм этого выбора мы дадим ниже (стартовый активный вызов есть `<Go >`); `i := 1`.
2. Пусть «$\mathtt{образец}_i$ `=` $\mathtt{правая\_часть}_i$» есть i-тое предложение функции `F`, тогда решается уравнение «$\mathtt{образец}_i$ `=` $\mathtt{d}_0$», если это уравнение несовместно, то переходим к решению уравнения «$\mathtt{образец}_{(i+1)}$ `=` $\mathtt{d}_0$». Если же i-ое предложение последнее, то значение функции `F` на аргументе $\mathtt{d}_0$ не определено и Рефал-машина переходит в состояние *аварийной остановки*.
3. Пусть уравнение «$\mathtt{образец}_i$ `=` $\mathtt{d}_0$» имеет решение. Если решение единственно, тогда переходим к пункту 4, иначе существует `e`-переменная, которая принимает более одного значения. Такие `e`-переменные называются *открытыми* в данном образце. Рассмотрим самую левую открытую переменную `e.y`; среди всех возможных значений `e.y` – конечных последовательностей $x_1^0 \ \ldots \ x_n^0$ выбираем последовательность с наименьшей длиной n и считаем только её значением переменной `e.y`, отбрасывая все решения уравнения «$\mathtt{образец}_i$ `=` $\mathtt{d}_0$», которые не удовлетворяют данному условию. Таким образом, мы сузили множество решений M. Если в образце остались другие открытые переменные, то выбираем среди них самую левую и выбираем из множества её значений самое короткое и т. д., пока в образце не останется открытых переменных.
4. Имеем выбранное предложение «$\mathtt{образец}_i$ `=` $\mathtt{правая\_часть}_i$» и выбранное нами его единственное решение. Делаем подстановку значений переменных в правую часть i-ого предложения и заменяем в поле зрения активный вызов `<F` $\mathtt{d}_0$`>` результатом этой подстановки.
5. Если в изменённом поле зрения нет вызовов функций, то машина останавливается, и поле зрения считается результатом её работы.

Заметим, что реальный (не абстрактный) алгоритм шага Рефал-машины может быть иным. Например, в Рефале-5 пункты 2 и 3 шага делаются одновременно и ищется сразу необходимое решение уравнения, а не всё множество решений.

Поле зрения, по определению, есть

```
поле_зрения ::= терм-поле_зрения поле_зрения1 | empty
терм-поле_зрения ::= <NAME поле_зрения> | (поле_зрения) | СИМВОЛ
```

Активным вызовом функции в поле зрения называется вызов, правая закрывающая угловая скобка которого является самой левой из всех правых угловых скобок.

Задача 5. Дана последовательность, элементами которой являются скобки `'('`, `')'`, `'<'`, `'>'`. Написать программу, выдающую сообщение об ошибке, если существует скобочный уровень одного типа скобок, на котором нарушена правильность скобочной структуры любого типа скобок. Если же на вход программы подана правильная скобочная структура, то преобразовать каждую пару `'(' ... ')'` в структурные скобки `( ... )`, а каждую пару `'<' ... '>'` в `(Call ... )`. Например,

$$\texttt{'(<()>)'} \Rightarrow \texttt{((Call ())).}$$

3.3. Домашнее задание.

Задача 6. Скобочной унарной системой счисления назовём систему счисления, в которой натуральные числа представляются парными вложенными скобками. Например, $((())) = 3$. Определить на Рефале функции скобочной унарной арифметики: сложения, вычитания, умножения и деления с остатком двух натуральных чисел.

Задача 7. Дана правильная скобочная структура d, содержащая только скобки. Требуется написать программу, результатом которой является последовательность всех различных объектных Рефал-выражений, скелеты которых совпадают с d и которые содержат ровно один символ `b`. Например,

```
(()) () ⇒ '[' b (()) () ']' '[' (b ()) () ']' '[' ((b)) () ']'
          '[' (() b) () ']' '[' (()) b () ']' '[' (()) (b) ']'
          '[' (()) () b ']'
```

Задача 8. Определить, может ли быть получен палиндром из данного слова путём перестановки букв.

Задача 9. (*Примитивный корень последовательности*). На входе конечная последовательность объектных Рефал-термов. Найти такую последовательность минимальной длины, повторением которой несколько раз можно получить входную последовательность.

Задача 10. *Анаграмма* – литературный прием, состоящий в перестановке букв определенного слова (или словосочетания), что в результате дает другое слово или словосочетание. Преобразовать текст в анаграмму, где одинаковые символы стоят подряд (так шифровали открытия средневековые ученые).

§ 4. Арифметика (Лекция №4)

Целочисленная арифметика в Рефале является точной. Другими словами: ограничение на разрядность (длину) целых чисел отсутствует. Представление целых чисел зависит от конкретного диалекта Рефала. Мы рассмотрим, каким образом целые числа представлены в Рефале-5.

4.1. Представление целых чисел. Прежде всего, снова расширим алфавит Рефал-символов: натуральное число n принадлежит алфавиту тогда и только тогда, когда

$$0 \leqslant n < 2^{32}$$

Целые числа в Рефале представляются в системе счисления по основанию 2^{32}. Таким образом, по определению, целое число принадлежит О.Д.З. `s`-переменной тогда и только тогда, когда оно является цифрой в этой системе счисления. Чтобы избежать путаницы с десятичными цифрами, мы будем называть цифры в системе счисления по основанию 2^{32} макро-цифрами.

Задача 1. Используя школьные алгоритмы, определить на Рефале функции арифметики в двоичной системе счисления: сложения, вычитания, умножения и деления с остатком двух натуральных чисел.

Как и в десятичной системе счисления, произвольное натуральное число есть последовательность макро-цифр. Например,
$2^{32} = 1\ 0$
$2^{32} - 1 = 4294967295$
Заметим, что `'1'` $\neq 1$ и `'4294967295'` $\neq 4294967295$.
Отрицательные числа начинаются с Рефал-символа `'-'`. Например,
$-(2^{32}+1) =$ `'-' 1 1` .
Аналогично разрешается приписывать символ `'+'` перед неотрицательным числом.

4.2. Встроенные функции арифметики. Некоторые функции существуют в Рефал-машине изначально – они называются встроенными.

Все встроенные функции арифметики целых чисел (сложение – `Add` (или `+`), вычитание – `Sub` (или `-`), умножение – `Mul` (или `*`), деление – `Div` (или `/`), деление с остатком – `Divmod`, остаток от деления – `Mod`, сравнение – `Compare`) логически имеют два аргумента, первый из которых заключён в структурные скобки. Например,

```
<Add (1 0) 1> = 1 1
```

Результат деления с остатком есть пара:

```
<Divmod (e.n) e.m> = (e.q) e.r
```

где `e.n` = `e.q`*`e.m` + `e.r`, $0 \leqslant |$`e.r`$| < |$`e.m`$|$ и знак остатка `e.r` совпадает со знаком делимого `e.n`.

Задача 2. Дана последовательность цифр натурального числа n, данного в системе счисления по основанию 2^{32}. Построить последовательность цифр числа n в десятичной системе счисления.

Функция сравнения двух целых чисел `Compare` возвращает: `'+'`, если первое больше второго; `'-'`, если первое меньше второго, и `'0'`, если числа равны.

В том случае, когда не возникает неоднозначности, структурные скобки, заключающие первый аргумент встроенных функций арифметики, можно опускать.

Задача 3. Реализовать на Рефале алгоритм умножения столбиком двух натуральных чисел, данных в в системе счисления по основанию 2^{32}. Встроенные функции разрешается использовать только тогда, когда их аргументы являются макро-цифрами.

4.3. Два типа рекурсии. На примере функции $n!$ покажем два вида рекурсии.

Рекурсия первого вида соответствует умножению чисел от 1 до n: $n! = 1 * \ldots * n$.

```
Fact {
 1   =  1;
 s.n = <Mul (s.n) <Fact <Sub s.n 1>>>;
}
```

Задача 4. Объясните, почему аргументом функции `Fact` выбрана макроцифра, а не произвольное натуральное число. Как будет работать Рефал-машина при попытке вычислить вызов `<Fact 0>`?

Рекурсия второго вида соответствует умножению чисел в обратном порядке: $n! = n * \ldots * 1$.

```
Fact1 {
 s.n = <Fact2 s.n 1>;
}

Fact2 {
 s.n s.n =  s.n;
 s.n s.m = <Mul (s.m) <Fact2 s.n <Add s.m 1>>>;
}
```

По способу завершения рекурсии, рекурсию первого вида назовём рекурсией с выходом по исчерпанию; рекурсию второго вида – рекурсией с выходом по наполнению.

Задача 5. Дано натуральное число n. Вычислить на Рефале n-ое число Каталана c_n (см. лекцию 1, задачу 7).

4.4. Фибоначчиева система счисления. Числа Фибоначчи определяются индуктивно: $f_1 = f_2 = 1, f_{n+2} = f_n + f_{n+1}$.

Задача 6. Дано натуральное число n. Вычислить на Рефале n-ое число Фибоначчи f_n.

Докажем, что любое натуральное число $m > 0$ можно однозначным образом представить в виде: $m = a_n * f_n + \ldots + a_2 * f_2$, где для всех $i < n$ верно, что a_i равно нулю или единице и $a_n = 1$, для всех i верно $a_i * a_{i+1} = 0$. Таким

образом, получаем фибоначчиеву систему счисления: a_i – цифры числа m в этой системе счисления.

Доказательство будет конструктивно и индуктивно.

Вычтем из $m_0 = m$ наибольшее из не превосходящих его чисел Фиббоначи f_n (с наибольшим номером) и рассмотрим $m_1 = m_0 - f_n$, $a_n = 1$.

Предположим, что k шагов уже выполнены, и мы имеем некоторое число m_k и последовательность фибоначчиевых цифр $a_n, \ \ldots, a_{n-k}$, состоящую из нулей и единиц. Тогда на $k+1$ шаге построим фибоначчиеву цифру $a_{n-(k+1)}$ и число m_{k+1}.

Если $m_k < f_{n-k}$, то $a_{n-(k+1)}$:= 0 и m_{k+1} := m_k,
иначе a_{k+1} := 1 и m_{k+1} := $m_k - f_{n-k}$.

Покажем корректность приведённого алгоритма.

На каждом шаге k уменьшается номер $n-k$ числа Фибоначчи, сравниваемого с $m_k \geqslant 0$ и, следовательно, уменьшается f_{n-k}.

Индукцией по k покажем, что $\forall k.(m_k \geqslant f_{n-k}) \Rightarrow (m_{k+1} < f_{n-(k+1)})$.

1. Базис индукции: $f_{n+1} > m_0 \geqslant f_n \Rightarrow m_1 = (m_0 - f_n) < (f_{n+1} - f_n) = f_{n-1}$. Базис индукции доказан.
2. Шаг индукции: пусть утверждение верно $\forall i \leqslant k$, тогда либо $m_{k+1} < f_{n-(k+1)}$ и посылка нашего утверждения для $k+1$ ложна; либо $m_{k+1} \geqslant f_{n-(k+1)}$ и тогда, по определению алгоритма, $m_{k+2} = m_{k+1} - f_{n-(k+1)}$.
   Кроме того, по предположению индукции, неравенство $m_{k+1} \geqslant f_{n-(k+1)}$ влечёт $m_k < f_{n-k}$ и, следовательно, $m_{k+1} = m_k$. Таким образом, $m_{k+2} = m_{k+1} - f_{n-(k+1)} = (m_k - f_{n-(k+1)}) < (f_{n-k} - f_{n-(k+1)}) = f_{n-(k+2)}$.
   Шаг индукции доказан.

Следствие 1. Алгоритм разложения натурального числа к виду $a_n * f_n + \ldots + a_2 * f_2$, где $\forall i < n$. a_i равно нулю или единице и $a_n = 1$, заканчивает свою работу за конечное число шагов.

Действительно, так как $f_{n-(k+1)} \leqslant f_{n-k}$ и $\min(m_k, m_{k+1}) = m_{k+1}$, то доказанное утверждение даёт $0 \leqslant m_{k+1} < f_{n-k}$, а последовательность f_{n-k} при $k < n-1$ строго убывает и $f_{n-(n-2)} = f_2 = 1$.

Пример:

$$m = m_0 = 19, m_1 = 19 - 13 = 19 - f_7 = 6; a_7 = 1;$$
$$m_1 < f_6 = 8 \Rightarrow a_6 = 0; m_2 = m_1;$$
$$m_2 \geqslant f_5 = 5 \Rightarrow a_5 = 1; m_3 = m_2 - f_5 = 6 - 5 = 1;$$
$$m_3 < f_4 = 3 \Rightarrow a_4 = 0; m_4 = m_3;$$
$$m_4 < f_3 = 2 \Rightarrow a_3 = 0; m_5 = m_4;$$
$$m_5 \geqslant f_2 = 1 \Rightarrow a_2 = 1; m_6 = m_5 - f_2 = 1 - 1 = 0;$$

и имеем $19_{10} = 101001_f = f_7 + f_5 + f_2 = 13_{10} + 5_{10} + 1_{10}$.

Задача 7. Напишите программу:

а). переводящую числа из фибоначчиевой системы счисления в десятичную;
б). переводящую числа из десятичной системы счисления в фибоначчиеву;
в). сложения двух натуральных чисел в фибоначчиевой системе счисления;
г). умножения двух натуральных чисел в фибоначчиевой системе счисления.

4.5. Домашнее задание.

Задача 8. В шестнадцатеричной системе счисления цифры представлены символами: `'0'`, `'1'`, `'2'`, `'3'`, `'4'`, `'5'`, `'6'`, `'7'`, `'8'`, `'9'`, `'A'`, `'B'`, `'C'`, `'D'`, `'E'`, `'F'`. Используя школьные алгоритмы, определить на Рефале функции арифметики в шестнадцатеричной системе счисления: сложения, вычитания, умножения и деления с остатком двух натуральных чисел.

Задача 9. Написать программу, вычисляющую наибольший общий делитель двух натуральных чисел методом Евклида.

Задача 10. Написать программу, вычисляющую наименьшее общее кратное двух натуральных чисел.

Задача 11. Придумать представление рациональных чисел посредством данных Рефала. Реализовать рациональную арифметику (функции сложения, вычитания, умножения, деления двух рациональных чисел).

Задача 12. Дано натуральное число n. Вычислить на Рефале n-ое число Фибоначчи f_n, если требуется, чтобы число арифметических операций при этом вычислении было пропорционально $\log n$.

§ 5. Функциональность. Ввод-вывод. (Лекция №5)

Второе определяющее слово реФала есть *функциональный*. Язык программирования $\{D, P, S\}$ называется функциональным, если для всех $p_0 \in P$ $S(p_0, d)$ есть *функция* из D в $D_\perp$ (см. лекцию №2). Другими словами, программа p_0 представляет собой определение некоторого *отображения* из D в $D_\perp$.

Функциональным языком, в строгом смысле приведённого выше определения, может быть только чисто теоретический язык программирования.

Задача 1. Какие из рассмотренных в предыдущих семинарах программы не являются определениями функций из D в $D_\perp$, где D – множество данных Рефала?

Причина состоит в том, что в *реальных* языках программирования всегда существуют программы с *побочными действиями*. И без таких программ мы не сможем узнать, что же вычислил компьютер, и вычислил ли вообще что-нибудь. Мы рассмотрим некоторые из подобных программ ниже.

Среди *реальных* (не чисто теоретических) языков программирования функциональными принято называть те языки, в которых число выразительных синтаксических средств, позволяющих конструировать программы с побочными действиями, сведено до минимума. *Типичным свойством* функционального языка программирования является отсутствие в нём глобальных переменных.

Задача 2. Приведите пример языка программирования, не являющегося функциональным.

5.1. Встроенный ввод-вывод. Мы уже использовали встроенную «функцию» печати `Prout`: её значением всегда является пустое выражение; а *побочным действием* – вывод на терминал своего аргумента.

Часто бывает необходимо вывести результат вычислений Рефал-машины в файл: с целью длительного хранения этого результата, другой причиной может быть большой объём выводимых данных.

Организация работы Рефал-машины с файлами зависит от конкретного диалекта Рефала. Мы будем интересоваться Рефалом-5.

File, в переводе с английского языка, – папка бумаг; она может быть пустая или содержать некоторую текстовую информацию. На обложке папки написано её имя.

Чтобы работать с содержимым файла, его, как и папку, нужно сначала *открыть*. При завершении работы с содержимым файла, файл необходимо *закрыть*. Цель работы может быть разная: чтение содержимого файла; добавление текста; уничтожение всей информации как устаревшей и запись в файл новой информации. У папок, взятых с разных книжных полок, могут быть совпадающие имена; чтобы не возникала путаница при работе с ними, принято прикреплять к ним (в момент открытия) *уникальные этикетки*, которые являются макро-цифрами – описателями (дескрипторами – descriptors). Посредством этих описателей и происходит реальная работа с файлами.

Файл с именем `'name'` открывает вызов

```
<Open s.Mode s.D 'name'>
```

где `s.D` – дескриптор файла, который задаёт программист, а `s.Mode` указывает цель работы с содержимым файла: `'r'` – открыть для чтения, `'a'` – открыть для дополнения, `'w'` – открыть для записи, предварительно *уничтожив его содержимое*. Если происходит попытка открыть для записи несуществующий файл, то он автоматически создаётся.

Закрываем файл посредством вызова

```
<Close s.D>
```

Значения «функций» `Open` и `Close` суть пустые выражения.

«Функция» `Putout` записывает (в виде одной строки) значение `e.Expr` в файл, связанный с дескриптором `s.D`:

```
<Putout s.D e.Expr>
```

Следующий вызов этой «функции» будет производить запись с начала следующей строки.

Задача 3. (*Соревнование*) Написать как можно более короткую программу, записывающую в файл `turnip.txt` русскую народную сказку «Репка».

Аналогично выводу, чтение (ввод) в Рефале-5 также построчный: n-ый вызов

```
<Get s.D>
```

выдаёт n-ую строку файла, связанного с `s.D`.

Чтение с клавиатуры производится вызовом

```
<Card >
```

который завершает чтение по нажатию клавиши `Enter`.

Задача 4. Написать программу, переставляющую строки в данном файле в обратном порядке.

Значение вызова `<Arg s.n>` есть `s.n`-ый аргумент командной строки операционной системы при запуске Рефал-машины.

Задача 5. Написать программу, значением которой является последовательность всех аргументов Рефал-машины, передаваемых командной строкой операционной системы.

5.2. Пример А. В. Корлюкова. Рассмотрим пример Александра Корлюкова, показывающий выразительные возможности ассоциативной конкатенации.

Пусть $B = \{\texttt{'0'}, \texttt{'1'}\}$, $M \supset B^3$. Требуется определить на Рефале функцию $F : M \to B^2$ такую, что $\forall x \in B^3$. $F(x)$ есть число n единиц в x, где n представлено в двоичной системе счисления и, возможно, содержит в своём представлении лидирующие нули.

Задачу начнём решать с полного перебора:

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

В этом определении предложения перестановочны. Объединяя первое и последнее предложение, получаем

```
s.d s.d s.d = s.d s.d;
```

Аналогично объединяем все предложения с совпадающими правыми частями. Наше определение теперь выглядит нижеследующим образом:

```
F {
  s.d s.d  s.d  =  s.d s.d;
  e.x  '1' e.y  '1' e.z  = '10';
  e.x  '0' e.y  '0' e.z  = '01';
}
```

Заметим, что во втором и в третьем предложении значения двух переменных равны пустому выражению; значение третьей переменной во втором предложении равно `'0'`, а в третьем – `'1'`. Таким образом, второе и третье предложение также можно объединить:

```
F {
  s.d s.d s.d  =  s.d s.d;
  e.x s.d e.y s.d e.z  =  s.d e.x e.y e.z;
}
```

Теперь нетрудно видеть, что первое предложение лишнее. Действительно, в результате отождествления трёх равных цифр с образцом `e.x s.d e.y s.d e.z`:
`e.x s.d e.y s.d e.z = d0 d0 d0`
имеем
`s.d = d0`
`e.x =`
`e.y =`
`e.z = d0`
и правая часть второго предложения становится равной `d0 d0`, то есть совпадает с правой частью первого предложения.

Окончательно имеем определение:

```
F {
  e.x s.d e.y s.d e.z = s.d e.x e.y e.z;
}
```

В результате наших преобразований область определения функции, данной стартовым определением, была расширена.

Задача 6. Написать программу, стирающую двойные (лишние) пробелы в данном тексте, то есть заменяющую каждую группу стоящих подряд пробелов единственным пробелом.

5.3. Домашнее задание.

Задача 7. Придумать представление понятия множества посредством данных Рефала. Определить функции объединения, пересечения и разности двух множеств.

Задача 8. (*Соревнование*) Написать как можно более короткую программу, записывающую в файл `Jack.txt` нижеследующее стихотворение Джонатана Свифта:

`The House That Jack Built`
`Jonathan Swift`

This is the house that Jack built.

This is the malt
That lay in the house that Jack built.

This is the rat,
That ate the malt
That lay in the house that Jack built.

This is the cat,
That killed the rat,
That ate the malt
That lay in the house that Jack built.

This is the dog,
That worried the cat,
That killed the rat,
That ate the malt

That lay in the house that Jack built.

This is the cow with the crumpled horn,
That tossed the dog,
That worried the cat,
That killed the rat,
That ate the malt
That lay in the house that Jack built.

This is the maiden all forlorn,
That milked the cow with the crumpled horn,
That tossed the dog,
That worried the cat,
That killed the rat,
That ate the malt
That lay in the house that Jack built.

This is the man all tattered and torn,
That kissed the maiden all forlorn,
That milked the cow with the crumpled horn,
That tossed the dog,
That worried the cat,
That killed the rat,
That ate the malt
That lay in the house that Jack built.

This is the priest all shaven and shorn,
That married the man all tattered and torn,
That kissed the maiden all forlorn,
That milked the cow with the crumpled horn,
That tossed the dog,
That worried the cat,
That killed the rat,
That ate the malt
That lay in the house that Jack built.

This is the cock that crowed in the morn,
That waked the priest all shaven and shorn,
That married the man all tattered and torn,
That kissed the maiden all forlorn,
That milked the cow with the crumpled horn,
That tossed the dog,
That worried the cat,
That killed the rat,
That ate the malt
That lay in the house that Jack built.

This is the farmer sowing the corn,
That kept the cock that crowed in the morn,
That waked the priest all shaven and shorn,
That married the man all tattered and torn,
That kissed the maiden all forlorn,
That milked the cow with the crumpled horn,

That tossed the dog,
That worried the cat,
That killed the rat,
That ate the malt
That lay in the house that Jack built.

Задача 9. Сонет -– стихотворение из 14 строк, которые образуют особым образом построенную строфу. Венок сонетов состоит из 15 стихотворений, где последняя строка каждого из 14 сонетов является началом следующего. Заключительный сонет воспроизводит первые строки всех 14 предыдущих сонетов (см. [9]).

Написать программу, проверяющую, является ли данное в файле стихотворение венком сонетов (см. определение венка сонетов на странице [9]). Программу протестировать на следующем стихотворении Владимира Солоухина (см. также файл [10]).

1

Венок сонетов – давняя мечта.
Изведать власть железного канона!
Теряя форму, гибнет красота,
А форма чётко требует закона.

Невыносима больше маета
Аморфности, неряшливости тона,
До скрежета зубовного, до стона,
Уж если так, пусть лучше немота.

Прошли, прошли Петрарки времена.
Но в прежнем ритме синяя волна
Бежит к земле из дали ураганной.

И если ты всё – мастер и поэт,
К тебе придёт классический сонет –
Вершина формы строгой и чеканной.

2

Вершина формы строгой и чеканной –
Земной цветок: жасмин, тюльпан, горлец,
Кипрей и клевер, лилии и канны,
Сирень и роза, ландыш, наконец.

Любой цветок сорви среди поляны –
Тончайшего искусства образец,
Не допустил Ваятеля резец
Ни одного малейшего изъяна.

Как скудно мы общаемся с цветами.
Меж красотой и суетными нами
Лежит тупая жирная черта.

Но не считай цветенье их напрасным,
Мы к ним идём, пречистым и прекрасным,
Когда невыносима суета.

3

Когда невыносима суета,
И возникает боль в душе глубоко,
И складка горькая ложится возле рта,
Я открываю том заветный Блока.

Звенит строка, из бронзы отлита,
Печального и гордого пророка.
Душа вольна, как дальняя дорога,
И до звезды бездонна высота.

О Блок! О Блок! Мертвею, воскреси!
Кидай на землю, мучай, возноси
Скрипичной болью, музыкой органной!

Чисты твоей поэзии ключи.
Кричать могу, молчанью научи,
К тебе я возвращаюсь в день туманный.

4

К тебе я возвращаюсь в день туманный,
О Родина, ужели это сны?
Кладу букет цветов благоуханный
На холмик глины около сосны.

И около берёзы. И в Тарханах.
И у церковной каменной стены.
Поэты спят; те стойкой ресторанной,
Те пошлостью, те пулей сражены.

А нас толпа. Мы мечемся. Мы живы.
Слова у нас то искренни, то лживы.
И без звезды живём и без креста.

Но есть дела. Они первостепенны.
Да ты ещё маячишь неизменно,
О белизна бумажного листа!

5

О белизна бумажного листа!
Ни завитка, ни чёрточки, ни знака.
Ни мысли и ни кляксы. Немота.
И слепота. Нейтральная бумага.

Пока она безбрежна и чиста,
Нужны или наивность, иль отвага
Для первого пятнающего шага –
Оставишь след и не сотрёшь следа.

Поддавшись страшной власти новизны,
Не оскверняй великой белизны
Поспешным жестом, пошлостью пространной.

Та белизна – дорога и судьба,

Та белизна – царица и раба,
Она источник жажды окаянной.

6

Она источник жажды окаянной.
Вся жизнь, что нам назначено прожить.
И соль и мёд, и горечь браги пьяной
Чем больше пьёшь, тем больше хочешь пить.

Сладкó вино за стенкою стаканной,
Мы пьём и льём, беспечна наша прыть,
До той поры, когда уж нечем крыть
И жалок мусор мелочи карманной.

За ледоход! За дождь! За листопад!
За синий свод – награду из наград,
За жаворонка в полдень осиянный,

За все цветы, за все шипы земли,
За постоянно брезжущий вдали
Манящий образ женщины желанной!

7

Манящий образ женщины желанной ...
Да – помыслы, да – книги, да – борьба.
Но всё равно одной улыбкой странной
Она творит героя и раба.

Ты важный, нужный, яркий, многогранный,
Поэт, главарь, – завидная судьба!
Уйдёт с другим, и ты сойдёшь с ума,
И будешь бредить пулею наганной.

Немного надо – встретиться любя.
Но если нет, то всюду ждут тебя
В пустых ночах пустые города,

Да всё-таки – надежды слабый луч,
Да всё-таки – сверкнувшая из туч
В ночи осенней яркая звезда.

8

В ночи осенней яркая звезда,
Перед тобой стою среди дороги.
О чём горишь, зовёшь меня куда,
Какие ждут невзгоды и тревоги?

Проходит лет, событий череда,
То свет в окне, то слёзы на пороге,
Глаза людей то ласковы, то строги,
Всё копится для страшного суда.

Для каждого наступит судный день:
Кем был, кем стал, где умысел, где лень?

Ты сам себе и жертва и палач.

Ну что ж, ложись на плаху головою,
Но оставайся всё-таки собою,
Себя другим в угоду не иначь.

9

Себя другим в угоду не иначь.
Они умней тебя и совершенней,
Но для твоих вопросов и задач
Им не найти ответов и решений.

Ты никуда не денешься, хоть плачь,
От прямиков, окольностей, кружений,
От дерзновенных взлётов и крушений,
От всех своих побед и неудач.

Привалов нет, каникул не бывает.
В пути не каждый сразу понимает,
Что жизнь не тульский пряник, не калач.

Рюкзак годов всё крепче режет плечи,
Но если вышел времени навстречу,
Души от ветра времени не прячь!

10

Души от ветра времени не прячь ...
Стоять среди железного мороза
Умеет наша светлая берёза,
В огне пустынь не гибнет карагач.

Но точит волю вечная угроза.
Но подлецом не должен быть скрипач.
Но губят песню сытость, ложь и проза,
Спасти её – задача из задач.

Берёшь, глядишь: такие же слова,
Похожа на живую, а мертва.
Но если в ней сознанье угадало

Хоть уголёк горячий и живой,
Ты подними её над головой,
Чтобы её как факел раздувало.

11

Чтобы её как факел раздувало,
Ту истину, которая в тебе,
Не опускай тяжёлого забрала,
Летя навстречу буре и борьбе.

Тлен не растлил, и сила не сломала.
И медлит та, с косою на горбе.
Хвала, осанна, ода, гимн судьбе –
Ты жив и зряч, не много и не мало!

С тобой деревья, небо над тобой.
Когда же сердце переполнит боль,
Оно взорвётся ярко, как фугас.

Возможность эту помни и держи,
Для этого от сытости и лжи
Хранится в сердце мужества запас.

12

Хранится в сердце мужества запас,
Как раньше порох в крепости хранили,
Как провиант от сырости и гнили,
Как на морском судёнышке компас.

Пускай в деревьях соки отбродили,
Пусть летний полдень засуху припас,
Пусть осень дышит холодом на нас
И журавли над нами оттрубили,

Пусть на дворе по-зимнему темно,
Согреет кровь старинное вино,
Уздечкой звякнет старенький пегас.

Придут друзья – обрадуемся встрече,
На стол поставим пушкинские свечи,
Чтоб свет во тьме, как прежде, не погас!

13

И свет во тьме, как прежде, не погас.
Да разве свет когда-нибудь погаснет?!
Костром горит, окном манит в ненастье,
В словах сквозит и светится из глаз.

Пустые толки; домыслы и басни,
Что можно, глыбой мрака навалясь,
Идущий день отсрочить хоть на час,
Нет ничего смешнее и напрасней!

И мрак ползёт. То атомный распад.
То душ распад. То свист, а то поп-арт.
Приоритет не духа, а металла.

Но под пустой и жалкой суетой
Он жив, огонь поэзии святой,
И тьма его, как прежде, не объяла.

14

И тьма его, как прежде, не объяла,
Мой незаметный, робкий огонёк.
Несу его то бодро, то устало,
То обогрет людьми, то одинок.

Уже не мало сердце отстучало,
Исписан и исчеркан весь листок,

Ошибок – воз, но этот путь жесток,
И ничего нельзя начать сначала.

Не изорвать в сердцах черновика,
Не исправима каждая строка,
Не истребима каждая черта.

С рассветом в путь, в привычную дорогу.
Ну, а пока дописан, слава Богу,
Венок сонетов – давняя мечта.

15

Венок сонетов – давняя мечта,
Вершина формы строгой и чеканной,
Когда невыносима суета,
К тебе я обращаюсь в день туманный.

О белизна бумажного листа!
Она источник жажды окаянной,
Манящий образ женщины желанной,
В ночи осенней яркая звезда!

Себя другим в угоду не иначь.
Души от ветра времени не прячь,
Чтобы её, как факел, раздувало.

Хранится в сердце мужества запас.
И свет во тьме, как прежде, не погас,
И тьма его, как прежде, не объяла!

§ 6. Алгоритмическая полнота. Алгорифмы Маркова. (Лекция №6)

Язык рефАЛ *алгоритмический*. Естественно встаёт вопрос об *алгоритмической полноте* Рефала. Оказывается, даже базисное подмножество Рефала, которое мы до сих пор только и рассматривали, является *алгоритмически полным языком*. То есть любой алгоритм может быть запрограммирован на базисном Рефале.

6.1. Алгоритмическая полнота.

Задача 1. Воспользовавшись тезисом Чёрча, докажите что базисный Рефал является алгоритмически полным языком.

Бóльшая часть языков программирования обладает свойством алгоритмической полноты. Следовательно, любую программу, написанную на одном из таких языков программирования, можно перевести на другой такой язык.

Разнообразие языков программирования объясняется их ориентацией на конкретные классы задач. Вопрос не в том можно или нельзя решить какую-то задачу на данном языке программировании L, а в том *удобно или нет* решать эту задачу на языке L. Язык есть инструмент для решения задачи: хотя и

можно гвозди забивать сковородкой (и даже лбом), но очень неудобно, – предпочитают использовать молоток.

Рефал ориентирован на *преобразование текстов* (символьной информации).

6.2. Алгорифмы Маркова. *Операционная семантика* языка Рефал основана на теоретической модели вычислений, называемой нормальными алгорифмами А. А. Маркова.

Пусть дан некоторый алфавит $\mathcal{A}$. Рассмотрим свободную алфавитную полугруппу G над $\mathcal{A}$. В качестве элементарной операции, на базе которой строятся алгорифмы Маркова, используется подстановка одного элемента G вместо другого. Если $x, y \in G$, то выражения $x \;\to\; y$ и $x \;\to\; \bullet\; y$ будем называть *формулами подстановки*. При этом предполагается, что стрелка $\to$ и точка $\bullet$ не являются буквами алфавита $\mathcal{A}$.

Формула подстановки $x \;\to\; y$ называется *простой*. Формула подстановки $x \;\to\; \bullet\; y$ называется *заключительной*. Пусть $P \;\to\; Q$ обозначает любую из формул подстановки (простую или заключительную). Конечная последовательность формул подстановки

$$\begin{cases} P_1 \;\to\; Q_1 \\ P_2 \;\to\; Q_2 \\ \cdots\cdots\cdots \\ P_r \;\to\; Q_r \end{cases}$$

называется схемой алгорифма.

Скажем, что последовательность (как элемент) $x \in G$ входит в последовательность $y \in G$, если существуют такие (возможно, пустые) $u, v \in G$, что $y = u\ x\ v$. Длину последовательности $x \in G$ обозначим $|x|$.

Задача 2. Пусть дано $y \in G$. Скажем, что существует два перекрывающихся вхождения $x \in G$ в y, если $\exists v \in G$ такое, что v входит в y, $|x| < |v| < 2|x|$ и $\exists p \in G\ \exists q \in G.\ \ v = x\ p = q\ x$. Написать на Рефале программу, проверяющую, что для данной последовательности y не существует последовательности x, которая имеет перекрывающееся вхождение в y.

Работа алгоритма, порождённого схемой алгорифма, может быть описана следующим образом. Пусть дано $p \in G$.

- Находим первую в схеме алгорифма формулу подстановки $P_m \;\to\; Q_m$ такую, что P_m входит в p.
- Подставляем Q_m вместо самого левого вхождения P_m в p. Пусть r_1 – результат такой подстановки.
- Если $P_m \;\to\; Q_m$ – заключительная формула подстановки, то работа алгоритма заканчивается и его значением на p является r_1.
- Если $P_m \;\to\; Q_m$ – простая, то применим к r_1 тот же поиск, который только что применяли к p, и так далее.
- Если мы на i-ом шаге получим такое r_i, что ни одна из $P_1, \ldots, P_r$ не входит в r_i, то работа алгоритма заканчивается и r_i будет его значением на p.

- При этом возможно, что описанный процесс никогда не закончится. В таком случае мы скажем, что алгоритм не применим к p.

Алгоритм, определённый таким образом, называется *нормальным алгорифмом Маркова в алфавите* $\mathcal{A}$.

Задача 3. Объясните в чём сходство и различие описанной выше машины Маркова и Рефал-машины. Что в машине Маркова является аналогом:

а). начальных данных?
б). вызова функции?
в). поля зрения?
г). активного вызова функции?
д). аварийной остановки «отождествление невозможно»?

Что в Рефал-машине является аналогом заключительной формулы подстановки?

Пример: Пусть $\mathcal{A} = \{b, c\}$. Пусть G – свободная алфавитная полугруппа над $\mathcal{A}$. Рассмотрим схему:

$$\begin{cases} b \to \bullet \\ c \to c \end{cases}$$

Определяемый этой схемой нормальный алгорифм перерабатывает всякое $p_0 \in G$, содержащее хотя бы одно вхождение буквы b, в элемент G, который получается вычёркиванием в p_0 самого левого вхождения буквы b. Пустая последовательность перерабатывается в саму себя. Алгорифм неприменим к непустым последовательностям, не содержащим буквы b.

Задача 4. Приведите конструктивное доказательство следующего утверждения. Если некоторый алгоритм A может быть описан схемой Маркова, то существует программа U, написанная на базисном Рефале такая, что для всех x из множества определения функции F, соответствующей алгоритму A, значение $F(x)$ может быть вычислено посредством программы U, то есть $F(x) = U(x)$. Под конструктивностью мы здесь понимаем явное предъявление программы U.

Задача 5. Реализуйте машину Маркова посредством Рефал-машины. Как связана данная задача с предыдущей? Проверьте работу построенной вами программы на всех алгорифмах Маркова, схемы которых рассматриваются в задачах или схемы которых требуется построить в задачах (данной лекции и домашнего задания).

6.3. Домашнее задание.

Задача 6. Пусть $\mathcal{A}$ – произвольный алфавит. Пусть G – свободная алфавитная полугруппа над $\mathcal{A}$. Рассмотрим (сокращённо записанную) схему алгорифма в алфавите $\mathcal{B} = \mathcal{A} \cup \{\alpha, \beta\}$, где $\alpha, \beta \notin \mathcal{A}$.

$$\begin{cases} \alpha\,\alpha \;\to\; \beta \\ \beta\,\xi \;\to\; \xi\,\beta & (\xi \in A) \\ \beta\,\alpha \;\to\; \beta \\ \beta \;\to\; \bullet \\ \alpha\,\eta\,\xi \;\to\; \xi\,\alpha\,\eta & (\eta, \xi \in A) \\ \quad \to\; \alpha \end{cases}$$

Эта схема определяет некоторый нормальный алгорифм Rev в алфавите $\mathcal{B}$. Докажите, что $\forall p \in G.\ \ Rev(p) = \overline{p}$. Где последовательность $\overline{p}$ получена из p обращением: если $p = a_1\ \ldots\ a_n$, то $\overline{p} = a_n\ \ldots\ a_1$.

Задача 7. Пусть алфавит $\mathcal{A}$ не содержит букву α, и пусть $\mathcal{B} = \mathcal{A} \cup \{\alpha\}$. Пусть G – свободная алфавитная полугруппа над $\mathcal{A}$. Построить нормальный алгорифм Маркова в $\mathcal{B}$, стирающий первую букву во всякой непустой последовательности $p \in G$.

Задача 8. Пусть алфавит $\mathcal{A}$ не содержит букв α, β, γ, и пусть $\mathcal{C} = \mathcal{A} \cup \{\alpha, \beta, \gamma\}$. Пусть G – свободная алфавитная полугруппу над $\mathcal{A}$. Построить нормальный алгорифм Маркова F в $\mathcal{C}$ такой, что $\forall p \in G$ было бы выполнено равенство $F(p) = p\,p$.

Задача 9. Даны две последовательности (два данных Рефала) $x_1,\ \ldots, x_n$ и $y_1,\ \ldots, y_k$. Требуется построить последовательность термов $z_1,\ \ldots, z_m$, состоящую из совпадающих и имеющих равные порядковые номера термов двух исходных последовательностей. Порядок термов z_i друг относительно друга не изменять.

Задача 10. Длина $Ln(\texttt{d})$ объектного Рефал-выражения `d` определяется индуктивно:

а). длина пустого выражения равна нулю;
б). $Ln(\texttt{t.x e.d}) = 1 + Ln(\texttt{e.d})$.

Определить на Рефале функцию, значение которой есть длина данного объектного выражения.

Задача 11. Размер $Size(\texttt{d})$ объектного Рефал-выражения `d` определяется индуктивно:

а). размер пустого выражения равен нулю;
б). $Size(\texttt{s.x e.d}) = 1 + Size(\texttt{e.d})$;
в). $Size(\texttt{(e.x) e.d}) = 2 + Size(\texttt{e.x}) + Size(\texttt{e.d})$.

Определить на Рефале функцию, значение которой есть размер данного объектного выражения.

§ 7. Вызов функции по определению (Лекция №7)

Кроме вызова функции по имени `<F e.args>`, то есть через указание имени данного определения функции, а не самого определения, синтаксис Рефала позволяет определить функцию непосредственно в точке её вызова, если обращение к такой функции в программе происходит лишь один раз. Начиная с этого места нашего изложения, мы выходим за рамки базисного Рефала.

7.1. Блоки. *Вызов функции по определению* иногда бывает удобен с точки зрения прозрачности исходного текста программы. Напомним, что в *базисном* Рефале определение функции имеет вид последовательности предложений, заключённой в фигурные скобки; а имя функции ставится перед её определением:

```
FunctionName {
   sentence1;
   .........
   sentencen;
}
```

где левая и правая части предложений разделены знаком равенства. Таким образом, определение само по себе (без имени) есть:

```
{
   sentence1;
   .........
   sentencen;
}
```

Если мы хотим определить функцию *непосредственно в точке её вызова*, то вместо имени функции (его у нас теперь нет) мы пишем сам текст определения, а аргументы этого вызова пишем перед фигурной скобкой, открывающей данное определение, отделив их от скобки знаком двоеточия «:». Непосредственно за *вызовом функции по определению* всегда ставится точка с запятой «;». Например, вызов

```
'abcd' (e.x) : {
                e.y (e.y) = True;
                e.y (e.z) = False;
               };
```

определяет, совпадает ли значение переменной `e.x` со строкой `'abcd'`, а вызов

```
<F e.w> (e.x) : {
                 e.y (e.y) = True;
                 e.y (e.z) = False;
                };
```

определяет, совпадает ли значение результата вызова (по имени) функции `F` со значением переменной `e.x`.

Теперь мы расширим понятие Рефал-предложения: кроме предложений, допускаемых базисным Рефалом, введём предложения, в которых левая часть отделяется от правой части запятой «,» (а не знаком равенства, как в базисном Рефале). Мы потребуем, чтобы в этом случае в правой части предложения (после запятой) обязательно стоял один вызов функции по определению, и только он. Естественно, мы допускаем вызовы по определению внутри любого определения – поименованного или непоименованного.

Пример №1: Следующая функция `S-Type` определяет, какого типа символ подан ей в качестве аргумента, и сообщает об ошибке, если аргумент не является Рефал-символом. Мы предварительно, используя встроенную функцию `Lower`, преобразуем все прописные буквы имени данного символа в строчные.

```
S-Type {
 s.x, <BelongsTo <Lower s.x> (<Letters>)>: {
                                            True   = Letter;
                                            False  = Other;
                                           };
 e.y = "Wrong argument: " e.y;
}

BelongsTo {
 s.x (e.y s.x e.z) = True;
 s.x (e.y) = False;
}

Letters {
 = 'abcdefghijklmnopqrstuvwxyz' 'абвгдеёжзийклмнопрстуфхцчшщъыьэюя';
}
```

Заметим, что `<S-Type 'f'>` $\neq$ `<S-Type f>`. Причина сего в том, что `'f'` есть символ-буква, а `f` есть символ-слово, имя которого состоит из одной буквы.

Рассмотрим общий вид Рефал-предложения с вызовом функции по определению:

```
pattern, argument :  {
                        sentence1;
                        .........
                        sentencen;
                       };
```

Множество переменных, входящих в образец `pattern`, может пересекаться с множеством переменных входящих в некоторое предложение $\texttt{sentence}_i$. Например,

```
t.x e.y, <F e.y>: {
                   t.x e.z = e.z;
                   e.y e.y e.z = e.z e.z;
                   e.z = e.y t.x;
                  };
```

Здесь первые два предложения (функции, вызванной по определению) содержат в левых частях переменные из образца `t.x e.y` основного предложения (вызывающего функцию по определению), а третье предложение содержит в правой части переменные из того же образца. Как понимать такую ситуацию? Какой смысл приписать подобному синтаксису? Семантика подобного синтаксиса в Рефале следующая: значения всех переменных, которые определились до запятой «`,`», Рефал-машина подставляет и в аргумент, и в само определение вызова функции по определению; и только после этого вычисляет указанный вызов (естественно, предварительно вычислив его аргументы). Следовательно, в третьем предложении

```
e.z = e.y t.x;
```

нет синтаксической ошибки, ибо в момент вызова оно превратится в

$$\texttt{e.z = e.y}_0\ \texttt{t.x}_0\texttt{;}$$

где $\texttt{e.y}_0$ и $\texttt{t.x}_0$ конкретные данные (константы) – значения переменных `e.y` и `t.x`.

Другими словами можно сказать, что определение функции, вызываемой по определению, может быть параметризовано. В рассмотренном выше примере определение функции зависит от параметров (неизвестных данных) `e.y`, `t.x`. Сущности `e.y`, `t.x` являются синтаксическими понятиями – переменными с точки зрения вызывающей функции, но семантическими понятиями – параметрами, с точки зрения вызываемой функции.

Пример №2: Следующая функция

```
a-z {
 e.1 'a' e.2, e.2: {
                    e.3 'z' e.4 = (e.1) 'a' e.3 'z' (e.4);
                    e.3 = <Prout 'No substring a-z found.'>;
                   };
 e.1 = <Prout  'No \'a\' found.'>;
}
```

выделяет первое вхождение в данном тексте строки, начинающейся на `'a'` и кончающейся на `'z'`. В последнем предложении мы «экранировали» две одинарные кавычки символом «\», так как они заключены в другую пару одинарных кавычек.

Задача 1. Рассмотрим другое определение функции из примера №2:

```
a-z-1 {
 e.1 'a' e.2 'z' e.3 = (e.1) 'a' e.2 'z' (e.3);
 e.3 = <Prout 'No substring a-z found.'>;
}
```

Объясните, почему определение `a-z` более эффективно, чем определение `a-z-1`. То есть найдется Рефал данное $\texttt{d}_0$ такое, что время вычисления вызова $\texttt{<a-z d}_0\texttt{>}$ будет *значительно* меньше времени вычисления вызова $\texttt{<a-z-1 d}_0\texttt{>}$, но не существует объектного выражения `d` такого, что время вычисления вызова `<a-z d>` будет *значительно* больше времени вычисления вызова `<a-z-1 d>`?

Вызов функции по определению в Рефале принято называть *блоком*. Присутствие блока в определении функции `F` приводит к *неопределённости понятия*

шага Рефал-машины при исполнении F: шаг функции F ещё не завершился, а уже внутри него начинают вычисляться вызовы функции, готовящие аргументы – входные данные для блока, да и сам блок вычисляется до окончания рассматриваемого шага функции F.

Задача 2. Провести пошаговый просмотр выполнения программы из примера №2 посредством отладчика. Понять и объяснить, какое действие Рефал-машины отладчик воспринимает как «шаг» Рефал-5 машины. Изучить поведение отладчика на последовательности команд `p act; com res;`.

7.2. Домашнее задание.

Задача 3. Глубину пустого выражения положим равной нулю. Глубина Depth Рефал-символа также, по определению, равна нулю.

```
Depth((e.d)) = 1+Depth(e.d)
```

Глубиной объектного выражения называется максимум глубин всех термов, входящих в это выражение. Определить на Рефале функцию, значение которой есть глубина данного объектного выражения.

Задача 4. В данном объектном Рефал-выражении (со скобками) уничтожить первое вхождение символа `'a'`, если таковое существует.

Задача 5. Даны две конечные неубывающие последовательности макроцифр. Построить неубывающую последовательность, элементами которой являются все элементы (включая их кратные вхождения) двух данных последовательностей.

Задача 6. Нарисовать ёлочку на экране компьютера средствами псевдографики (посредством символов клавиатуры). Высота ёлочки должна быть равна высоте экрана.

Задача 7. Дана конечная последовательность макроцифр $x_1 = 1$, $x_{n+1} = 1 + x_n$. Построить все различные перестановки данной последовательности.

Задача 8. Дана симметрическая группа перестановок S_n. Придумать представление элементов этой группы во множестве данных Рефала и реализовать групповые операции умножения и построения элемента, обратного к данному элементу.

Задача 9. Рассмотрим полугруппу по умножению квадратных матриц размера $n \times n$, состоящих из целых чисел. Придумать представление элементов этой полугруппы как данных Рефала и реализовать операцию умножения двух матриц.

§ 8. Рефал как метаязык программирования (Лекция №8)

Язык Рефал *ориентирован на преобразование текстов*. Рефал изначально разрабатывался В. Ф. Турчиным как *метаязык*, – то есть инструмент для анализа и преобразования текстов, написанных на других языках, в частности, на языках программирования. Программы (тексты на языках программирования) обычно анализируются либо с целью их перевода на другие языки

программирования, либо с целью их исполнения. В обоих случаях семантика соответствующего языка программирования должна быть определена в терминах Рефала.

Объектами преобразований Рефал-программ могут быть программы, написанные также на языке Рефал. В частности, из алгоритмической полноты Рефала следует, что функция семантики Рефала может быть описана в терминах самого Рефала. Рассмотрим некоторые проблемы, связанные с понятием *метаязыка*.

Перед прочим заметим, что примером метаязыка является русский язык, а примером анализа текстов на русском языке посредством самого языка является предложение, *которое вы сейчас читаете*.

При *самоанализе* возникает проблема с синтаксисом: если мы пишем текст о синтаксисе русского языка, то, например, знаки препинания мы вынуждены обозначать не самими этими знаками как таковыми, а *словами, их обозначающими* – иначе мы нарушим правила самого синтаксиса, о котором рассуждаем. И вместо предложения: *«В конце предложения нужно ставить точку.»* у нас получится предложение: *«В конце предложения нужно ставить ..»*, которое заканчивается знаком горизонтального двоеточия, отсутствующим в русском языке.

В Рефале обсуждаемая проблема проявляется, например, в том, что если требуется, при анализе одной Рефал-программы посредством другой Рефал-программы, определить имя некоторой переменной `s.x`, то переменную `s.x` (а не её значение) невозможно передать *напрямую* (как значение аргумента) для анализа:

```
$ENTRY Go {
  = <Analyze s.x>;
}
```

Снова возникает синтаксическая ошибка: переменная в правой части предложения, которой нет в левой части этого предложения. Как и в русском языке, в подобных случаях мы вынуждены представлять одни понятия языка посредством других. Например, так:

```
$ENTRY Go {
  = <Analyze (Variable 's' x)>;
}
```

Представление одних понятий языка посредством других называется кодировкой.

Кодировка должна позволять восстанавливать смысл закодированного понятия, как и позволять узнавать, является ли то или иное понятие кодировкой (или частью кодировки) другого понятия, или оно представляет само себя. В противном случае неизбежно возникнет путаница. В предложении *«Рассмотрим точку A.»* слово *точка* представляет *само себя*, а не знак, которым заканчивается повествовательное предложение. То же самое можно сказать и о слове *точка* как в предыдущем предложении (второе вхождение), так и в данном предложении. В нашем примере мы смогли определить смысл слова *«точка»* из контекста.

Язык программирования $L = \{D, P, S\}$, где D – множество данных, P – множество программ, S – функция семантики, назовём *метаязыком программирования*, если в нём фиксирована инъективная функция кодировки $\nu : P \to D$.

Инъективность здесь нужна для однозначного восстановления смысла понятий. Далее, для лучшей читаемости, мы будем обозначать отображение кодировки подчёркиванием. Например, $\nu(\texttt{s.x}) = \underline{\texttt{s.x}}$.

Задача 1. Дана матрица M размера $n \times n$, состоящая из нулей и единиц. Придумать представление M в множестве данных Рефала и написать программу, результатом которой является строка длины n, состоящая из нулей и единиц и не являющаяся строкой матрицы M.

8.1. Самоприменимые программы. Программу p назовём самоприменимой, если вызов <p p> не циклится (работает конечное время).

Пусть $\texttt{p} \in P$. Рассмотрим функцию $\phi : Im(\nu) \to \{\texttt{finite, infinite}\}$[2]

$$\varphi(p) = \begin{cases} \texttt{finite, если <p p> заканчивает работу в конечное время;} \\ \texttt{infinite, если вызов <p p> циклится.} \end{cases}$$

Везде ранее мы рассматривали только *вычислимые* (конструктивные) функции. Другими словами, функции которые можно определить на алгоритмически полном языке. Покажем, что функция φ *не является вычислимой.*

Теорема: Не существует программы, которая распознавала бы самоприменимые программы.

Доказательство будем вести методом от обратного. Предположим, что существует программа q, которая реализует функцию φ. Тогда рассмотрим программу f такую, что

<f p> = finite, если <q p> = infinite,
и <f p> циклится, если <q p> = finite.

Рассмотрим вызов <f f>. Предположим что <f f> = finite, тогда по определению f <q f> = infinite и, следовательно, теперь уже по определению q, программа f не является самоприменимой, что противоречит нашему предположению.

Следовательно, вызов <f f> циклится. Но тогда по определению f, <q f> = finite и, следовательно, по определению q, программа f является самоприменимой, что противоречит тому, что вызов <f f> циклится.

Таким образом, оба из логически возможных поведений исполнения вызова <f f> приводят к противоречию. И наше предположение о существовании программы q, реализующей функцию φ, неверно. Доказательство теоремы закончено.

Задача 2. Объясните, каким образом приведённое выше доказательство связано с задачей №1 про матрицу.

Задача 3. Докажите, что нельзя написать программу, которая для любой программы p определяла бы, что существует $d_0 \in D$ такое, что <p d_0> циклится.

[2] $Im(\nu)$ – множество значений (образ) отображения ν.

Множество M называется *счётным*, если его можно пересчитать. Другими словами: существует биекция $\mathbb{N} \to M$.

Задача 4. Докажите, что множество всех действительных чисел из интервала $(0, 1)$ несчётно. Объясните, каким образом утверждение данной задачи связано утверждением о несуществовании программы, которая распознавала бы самоприменимые программы.

8.2. Функция применения. Приведём пример встроенной *мета-функции* Рефала-5. Вызовы

```
<Mu s.fname e.args>
```

и

```
<Mu (e.fname) e.args>
```

вычисляют значение функции с именем, переданным через переменную, на аргументе `e.args`. В первом случае имя вычисляемой функции есть символ Рефала, во втором случае имя функции представлено последовательностью букв.

Задача 5. Даны имя функции и последовательность объектных термов. Требуется определить функцию, значение которой есть последовательность значений данной функции на каждом из термов данной последовательности. Порядок значений должен совпадать с порядком аргументов.

Задача 6. Даны имя функции и последовательность корневых деревьев (объектных Рефал-выражений). Требуется определить программу, заменяющую каждый лист данных деревьев значением данной функции на этом листе.

8.3. Домашнее задание.

Задача 7. Докажите, что ниже перечисленные множества счётны, определите соответствующие биекции на Рефале:

а). множество целых чисел $\mathbb{Z}$;
б). множество $\mathbb{Z}^2$;
в). множество $\mathbb{Z}^3$;
г). множество $\mathbb{Z}^{k_0}$, где k_0 фиксированное натуральное число.

Задача 8. Является ли счётным множество данных Рефала? Если ответ положителен, то определите соответствующую биекцию на Рефале.

Задача 9. Докажите, что нельзя написать программу, которая для любой программы `p` определяла бы, что для всех $d \in D$. `<p d>` $= d$.

Задача 10. Дан объектный терм T_0 и последовательность имён функций. Требуется определить функцию, значение которой есть значение композиции данных функций, первая из которых вычисляется на T_0.

§ 9. Рекурсивные условия (Лекция №9)

Ранее мы ввели в синтаксис Рефал-программ новое понятие – запятую, тем самым выйдя за рамки базисного Рефала. Кроме отделения левой части от

правой части предложения, при вызове функции по определению знак запятой используется для организации *рекурсивного условия выбора предложения* внутри одной функции. В обоих случаях знак запятой можно понимать как логическую связку – конъюнкцию (то есть «и»).

9.1. Другая интерпретация знака запятой. Вернёмся к нашему примеру выделения в данной строке подстроки, начинающейся на `'a'` и кончающейся на `'z'`. Ранее мы запрограммировали его, используя блок (вызов функции по определению).

```
a-z {
 e.1 'a' e.2, e.2: {
                    e.3 'z' e.4 = (e.1) 'a' e.3 'z' (e.4);
                           e.3 = 'No substring a-z found.';
                   };
 e.1 = 'No \'a\' found.';
}
```

Смысл пары запятая-двоеточие в первом предложении

```
e.1 'a' e.2, e.2:
```

можно понимать и так: если аргумент вызова функции `a-z` является Рефал-выражением вида `e.1 'a' e.2` и значение переменной `e.2` такое, что ... (далее см. определение блока). Или ещё так: если аргумент вызова функции `a-z` является объектным Рефал-выражением вида `e.1 'a' e.2`, где значение переменной `e.2` такое, что ... (далее см. определение блока). Именно такой смысл пары запятая-двоеточие вкладывается в «`, expression :`» при введении в *расширенный* Рефал-рекурсивных условий.

9.2. Рекурсивные условия выбора предложения. В *базисном* Рефале Рефал-машина использует структуру образца и алгоритм сопоставления данных-аргументов функции `F` с образцом для выбора предложения из последовательности предложений, определяющей тело функции `F`. Механизм отождествления не является «алгоритмически полным». (Например, он не может иметь бесконечных циклов.) Используем отмеченный выше смысл синтаксической конструкции «`, expression :`» для введения алгоритмически полного инструмента выбора Рефал-предложения.

В *расширенном* Рефале Рефал-предложение может иметь один из следующих видов:

1. `pattern = expression;`
2. `pattern , arguments :  { block };`
3. `pattern conditions right_part;`

где `right_part` есть либо «`= expression`», либо «`, arguments : { block }`», а

`conditions ::= ,` *`expression :  pattern`* `conditions |` *`empty`*

Причём `expression` имеет тот же вид что и правая часть предложения в базисном Рефале, и удовлетворяет тем же синтаксическим ограничениям: в `expression` допускаются только переменные, которые уже встречались в рассматриваемом предложении левее данного `expression`.

Мы начнём с простейшего примера, показывающего, что даже при отсутствии рекурсии в выборе предложения, новая конструкция *условия* позволяет писать программы более компактно.

Пример №1: Пусть функция `Look-For-I`(x_1, x_2, x_3), где x_i – строки символов, есть предикат, отвечающий на вопрос «Принадлежит ли символ `'I'` хотя бы одной строке x_i?». Тогда в базисном Рефале определение `Look-For-I`, если не использовать вспомогательных функций, выглядит так:

```
Look-For-I {
 (e.x 'I' e.x1) (e.x2)          (e.x3) = True;
 (e.x1)         (e.y 'I' e.x2) (e.x3) = True;
 (e.x1)         (e.x2)          (e.z 'I' e.x3) = True;
 (e.x1)         (e.x2)          (e.x3)         = False;
}
```

Используя блок, его можно упростить следующим образом:

```
Look-For-I {
(e.x1) (e.x2) (e.x3), e.x1 e.x2 e.x3 : {
                                        e.x 'I' e.y = True;
                                        e.z = False;
                                       };
}
```

Использование условия позволяет написать эту программу ещё короче:

```
Look-For-I {
 (e.x1) (e.x2) (e.x3), e.x1 e.x2 e.x3 :  e.x 'I' e.y = True;
  e.z = False;
}
```

9.2.1. *Откаты при выборе предложения.*

Пример №2: Дана строка `e.string` и лес деревьев `e.forest`. Требуется найти самое левое дерево, среди листов которого содержится самый левый символ строки `e.string`, из тех которые являются листьями данного дерева. Определим соответствующую программу, используя рекурсию в выборе предложений.

```
Tree {
 e.x s.leaf e.string (e.y t.tree e.forest)
     , <Get-leaves t.tree>: e.z s.leaf e.leaves = t.tree;
 e.x                 (e.forest)  = Fail;
}

Get-leaves {
 e.x (e.y) e.z = e.x <Get-leaves e.y e.z>;
 e.x = e.x;
}
```

Здесь рекурсивно происходят откаты при выборе Рефал-машиной предложения функции `Tree`. Два образца

`e.x s.leaf e.string (e.y t.tree e.forest)` и `s.z s.leaf e.leaves` имеют общую переменную `s.leaf` и взаимодействуют при работе алгоритмов отождествления. Согласно общему правилу Рефала при решении уравнения

`e.x s.leaf e.string (e.y t.tree e.forest) = string`$_0$ `(forest`$_0$`)`

, где `string`$_0$ и `forest`$_0$ – данные Рефала, из всего множества решений M_i

1. выбираются те решения, в которых `e.x` приняла значение с наименьшей возможной длиной (количеством деревьев в `e.x`$_0$),
2. если такой выбор не приводит к однозначности решения, тогда из выбранного подмножества решений выбираются те решения, в которых `e.y` приняла значение с наименьшей возможной длиной (количеством деревьев в `e.y`$_0$). Последний выбор всегда приведёт к однозначности, если множество решений не пусто.

Таким образом, в результате решения указанного выше уравнения Рефал-машина имеет значения переменных `t.tree = t.tree`$_0$`, s.leaf = s.leaf`$_0$ и может вычислить вызов `<Get-leaves t.tree`$_0$`>`. Пусть результат этого вычисления равен `leaves`$_0$, тогда Рефал-машина решает уравнение, определённое условием в первом предложении функции `Tree`:

`e.z s.leaf e.leaves = leaves`$_0$

если это уравнение имеет решение, тогда Рефал-машина нашла дерево `t.tree`$_0$ с требуемым свойством. Иначе Рефал-машина возвращается к уравнению

`e.x s.leaf e.string (e.y t.tree e.forest) = string`$_0$ `(forest`$_0$`)`

и считает, что выбранное ранее его решение

`e.x`$_0$`, s.leaf`$_0$`, e.string`$_0$`, e.y`$_0$`, t.tree`$_0$`, e.forest`$_0$

не удовлетворяет условию выбора данного предложения. Пусть $ln($`e.y`$_0)$ есть длина (количество термов-деревьев в) `e.y`$_0$, тогда множество решений данного уравнения сужается условием $ln($`e.y`$) > ln($`e.y`$_0)$.

- Если получившееся подмножество не пусто, то обозначим его $K_{i+1} \subset M_i$ ($K_{i+1} \neq M_i$), переименуем его M_i `:=` K_{i+1} и переходим к пункту 1).
- Иначе Рефал-машина снова возвращается к решению уравнения
  `e.x s.leaf e.string (e.y t.tree e.forest) = string`$_0$ `(forest`$_0$`)`
  Множество решений M_i данного уравнения сужается условием $ln($`e.x`$) > ln($`e.x`$_0)$. Обозначим получившееся подмножество $Q_{i+1} \subset M_i$ ($Q_{i+1} \neq M_i$), переименуем его M_i `:=` Q_{i+1} и переходим к пункту 1).

Задача 1. Доказать, что определённая выше функция `Tree` заканчивает работу за конечное время при любых значениях её аргументов.

Задача 2. Каковы будут значения переменных `e.x` и `e.y` при первом вызове функции `Get-leaves`? Зависят ли эти значения от аргументов вызова функции `Tree`?

Задача 3. Чему равно максимальное число откатов, которое сделает Рефал-машина, при поиске нужного решения уравнения
`e.x s.leaf e.string (e.y t.tree e.forest) = string`$_0$ `(forest`$_0$`)` ?

Вставим в нашу программу отладочную печать, чтобы проследить работу Рефал-машины в деталях.

```
$ENTRY Go {
 = <Prout '\nThe result: ' <Tree A B C ((E) (D (B)) ((B) (N C)))>>;
}

Tree {
 e.x s.leaf e.string (e.y t.tree e.forest),
   <Prout 'Try the leaf: ' s.leaf ' in the tree: ' t.tree
          '    \n\t , where the length of e.y=' e.y ' is ' <Ln e.y>>
   <Prout '    \t   and the length of e.x=' e.x ' is ' <Ln e.x>>
   <Get-leaves t.tree>: e.z s.leaf  e.leaves = t.tree;
 e.x  (e.forest)  = Fail;
}

Get-leaves {
 e.x (e.y) e.z = e.x <Get-leaves e.y e.z>;
 e.x = e.x;
}

Ln {
            = 0;
 t.term e.x = <+ (<Ln e.x>) 1>;
}
```

В программе мы использовали символ перевода строки '\n' и символ горизонтальной табуляции '\t', также мы воспользовались тем, что значение вызова <Prout e.args> всегда равно пустому выражению. В результате исполнения этой программы на экране компьютера имеем:

```
Try the leaf: A  in the tree: (E )
         , where the length of e.y= is 0
           and the length of e.x= is 0
Try the leaf: A  in the tree: (D (B ))
         , where the length of e.y=(E ) is 1
           and the length of e.x= is 0
Try the leaf: A  in the tree: ((B )(N C ))
         , where the length of e.y=(E )(D (B )) is 2
           and the length of e.x= is 0
Try the leaf: B  in the tree: (E )
         , where the length of e.y= is 0
           and the length of e.x=A  is 1
Try the leaf: B  in the tree: (D (B ))
         , where the length of e.y=(E ) is 1
           and the length of e.x=A  is 1

The result: (D (B ))
```

Задача 4. Выполните данную выше программу с отладочной печатью на вашем компьютере. Вставьте дополнительную печать, показывающую как из-

меняется при выборе предложения длина значения переменной `e.z`. Объясните полученный вами результат.

9.2.2. *Локальное присваивание.* Синтаксическую конструкцию условия можно также использовать для устранения повторных вычислений. В этом случае само условие тривиально – всегда истинно, и мы используем его конструкцию как конструкцию локального присваивания.

Пример №3: Пусть нам нужно определить функцию на множестве целых чисел $g(n) = (n-1) * (n-1)$. Определим `g` в базисном Рефале:

```
g {
  e.n = <* (<- (e.n) 1>) <- (e.n) 1>>;
}
```

Использование синтаксической конструкции условия позволяет устранить повторное вычитание 1.

```
g {
  e.n, <- (e.n) 1>: e.m = <* (e.m) e.m>;
}
```

Задача 5. Переопределить данную в примере №3 функцию `g` в базисном Рефале, но без повторного вычитания 1.

Присутствие условия в определении функции `F` может привести к *неопределённости понятия шага Рефал-машины* при исполнении `F` (как и присутствие блока в `F`): шаг функции `F` ещё не завершился, а уже внутри него начинают вычисляться вызовы функции, готовящие аргумент для этого условия.

Задача 6. Провести пошаговый просмотр выполнения программы из примера №2 посредством отладчика. Понять и объяснить какое действие Рефал-машины отладчик воспринимает как «шаг» Рефал-5 машины. Изучить поведение отладчика на последовательность команд `p act; com res`.

9.3. Домашнее задание.

Задача 7. Переопределить данную в примере №2 функцию `Tree`, ограничившись синтаксисом базисного Рефала.

Задача 8. Дано объектное Рефал-выражение d_0, в которое могут входить структурные скобки. Определить функцию f поиска первого вхождения знака аддитивной операции (`'+'` или `'-'`) на верхнем уровне скобочной структуры данного d_0. Значением f должно быть исходное d_0, в котором часть предшествующая найденному знаку аддитивной операции заключена в структурные скобки. Если требуемого знака не существует, то функция f должна выдавать не изменённое d_0.

Задача 9. Дан лес деревьев (объектное Рефал-выражение со скобками). Требуется определить на Рефале функцию, переставляющую листья деревьев в обратном порядке, если листья находятся на нечётной глубине, и не меняющую порядок листьев, находящихся на чётной глубине. Структуру деревьев не изменять. Задачу решить в двух вариантах:

а). Локально, то есть переставляются в каждом дереве листья, находящиеся на ветках одного и того же ближайшего к ним узла-ветвления. Например,

`(a b c) d t (f g 1 (3 4) 5)` $\Rightarrow$ `(c b a) d t (5 1 g (3 4) f)`;

б). Глобально, то есть последовательности листьев, находящихся на глубине $2 * n + 1$, приписываются друг к другу согласно следованию – слева направо. После этого элементы полученной последовательности переставляются в обратном порядке и затем развешиваются на соответствующие ветви деревьев.

Например, для леса `(a b c) d t (f g 1 (3 4) 5)` имеем последовательность листьев, соответствующую глубине 1: `a b c f g 1 5`. После перестановки элементов этой последовательности она преобразуется в `5 1 g f c b a`. Теперь развешиваем листья на деревья и получаем: `(5 1 g) d t (f c b (3 4) a)`.

§ 10. Сложность Колмогорова (Лекция №10)

Рефал ориентирован на *преобразование текстов*. Какие тексты являются простыми, а какие сложными? Классическое рифмованное стихотворение выучить наизусть много проще, чем белый стих. Кроме того, понятие сложности текста `T` зависит от нашего понимания `T`. Запоминание доказательства теоремы при подготовке к экзамену является делом безнадёжным, когда студент не понимает этого доказательства; в то же время, если доказательство понято, то оно легко восстанавливается, исходя из логики взаимозависимости его шагов. Великий русский математик Андрей Николаевич Колмогоров предложил *аппроксимировать понятие взаимозависимости частей данного текста посредством рекурсии*.

10.1. Сложность текста. Иногда возможно длинный текст описать посредством короткого. Например, текст

`миллион слов 'рефал'`,

состоящий из 20 символов, является описанием текста длиной пять миллионов символов. Ранее мы уже пробовали описывать русскую народную сказку «Репка» посредством как можно более короткой Рефал-программы.

Задача 1. Дано стихотворение Владимира Солоухина, рассмотренное нами на лекции №5 – задача №9. Обозначим его T_1. Создать текст T_2, состоящий из одних нулей, длина которого равна длине текста T_1. Из текста данной лекции №5 вырезать кусок T_3; длина текста T_3 должна равняться длине текста T_1. Сжать тексты T_1, T_2, T_3 всеми известными вам упаковщиками («архиваторами»), составить таблицу (одна строчка для каждого упаковщика) длин получившихся текстов и сравнить эти длины.

Пусть `p` – какая-нибудь программа. *Сложностью* текста T при способе описания `p` называется наименьшая длина текста Z такого, что `<p Z>` = T. При этом текст Z называется *описанием* текста T.

Иными словами, мы рассматриваем все сжатые версии текста T, из которых данный распаковщик `p` может восстановить T, и среди них выбираем самое короткое описание. Например,

```
миллион слов 'рефал'
```

и

```
пятьсот тысяч слов 'рефалрефал'
```

– два описания одного текста, но первое описание короче.

Некоторые упаковщики создают самораспаковывающиеся архивы. В таких архивах содержится и распаковщик, и заархивированный текст. Текст распаковщика должен быть каким-то образом отделён от заархивированного текста.

10.2. Структурные скобки и кодирование разделения аргументов. С подобной проблемой мы уже встречались: в Рефале все функции одноместны (имеют только один аргумент). Мы использовали конструктор структурных скобок, чтобы имитировать функции нескольких переменных. То есть, мы кодировали *понятие разделения аргументов* посредством структурных скобок. Другими словами, мы структурировали данные Рефала. И, как мы знаем, структурные скобки позволяют удобно структурировать Рефал-данные в виде леса корневых деревьев; в этом основная роль структурных скобок. За такую возможность мы заплатили ограничением алфавита: скобка `(` не может представлять сама себя в тексте Рефал-программы, – синтаксис требует закодировать её обрамлением в одинарные или двойные кавычки (так `'('` или так `"("`).

Оказывается, можно закодировать понятие разделения аргументов, не вводя никаких дополнительных ограничений на алфавит, если алфавит содержит по крайней мере две буквы. Разделить распаковщик и заархивированный текст можно, например, таким образом:

```
рраассппааккууюющщаааяя  ппррооггрраамммммаааабзапакованный текст
```

где мы каждую букву исходного текста распаковщика продублировали (после `яя` стоят два пробела), а сразу после закодированного текста распаковщика мы написали буквы `аб` ; сам запакованный текст никак не изменили.

Задача 2. Докажите, что указанное кодирующее отображение инъективно.

Задача 3. Решите следующую задачу, не используя структурные скобки. Даны текст и строка. Определить на Рефале функцию, значение которой есть отрезок данного текста, непосредственно следующий за *первым вхождением* данной строки, если такое вхождение существует, и сама данная строка, если она не входит в текст.

10.3. Теорема Колмогорова. Пусть дан алгоритмически полный язык программирования L (например, Рефал без структурных скобок), на котором мы намерены реализовывать упаковывающие программы. Пусть D есть множество данных языка L, а P – множество программ языка L.

Теорема 1. Существует способ, который даёт почти самые короткие описания всех текстов. То есть, существует программа `p` такая, что для любого текста T сложность описания T посредством `p` меньше, чем сложность описания T посредством любой другой программы `q` плюс некоторая константа

`Const`, зависящая только от `q`. То есть, `Const` не зависит от T. Обозначим через $\varphi_{\mathtt{p}}(T)$ сложность описания T посредством `p`, тогда

$$\exists p \in P\ \forall q \in P\ \exists \mathtt{Const} \in \mathbb{R}\ \forall T \in D.\ \varphi_{\mathtt{p}}(T) < \varphi_{\mathtt{q}}(T) + \mathtt{Const}$$

Теорема говорит о том, что можно написать универсальный почти оптимальный упаковщик `p`. Любой другой упаковщик `q` будет сжимать исходные тексты хуже, чем `p` – с точностью до некоторой константы.

Докажем теорему. Данный язык L, по условию, алгоритмически полон. Следовательно, мы можем реализовать на языке L *универсальную программу* `Int` $\in P$, которая умеет вычислять значение любой другой данной программы `q` $\in P$ (написанной на языке L) на любом конкретном $\mathtt{d}_0 \in D$. Программа `q` есть отображение из D в $D_\perp$ (смысл обозначения см. в лекции №2). Следовательно,

$$\mathtt{Int} : P \times D \to D_\perp$$

Условие универсальности программы `Int` формально выглядит так:

$$\mathtt{<Int\ \underline{qd}>} = \mathtt{<q\ d>}$$

где мы намеренно не разделили программу `q` и её данные `d`, – разделение делает кодировка, обозначенная нами подчёркиванием. Универсальные программы программисты называют *интерпретаторами* (interpreter – исполнитель).

Пусть у нас имеется произвольный способ описания $\mathtt{q} \in P$, тогда длину $\underline{\mathtt{q}Z}$ можно ограничить числом $2 * k + n + 2$, где k – длина `q`, n – длина текста Z, являющего описанием текста T при способе `q`. Далее,

$$\forall Z\ \mathtt{<Int\ \underline{q}}Z\mathtt{>} = \mathtt{<q}\ Z\mathtt{>} = T$$

то есть пара (`Int`, $\underline{\mathtt{q}Z}$) распаковывает текст T при способе `Int`. Следовательно, по определению описания,

$$\varphi_{\mathtt{Int}}(T) < 2 * k + n + 3$$

По определению, Z есть описание T при способе `q`. То есть $n = \varphi_{\mathtt{q}}(T)$. Следовательно,

$$\forall q\ \varphi_{\mathtt{Int}}(T) < \varphi_{\mathtt{q}}(T) + 2 * k + 3$$

где k – длина самого способа `q` (длина исходного текста программы `q`), которая зависит только от `q` и не зависит от T. Что и требовалось доказать.

10.4. Домашнее задание.

Задача 4. Выше мы описали кодировку понятия разделения аргументов.

а). Реализуйте описанную кодировку на Рефале.

б). Реализуйте описанную кодировку на Рефале, не используя структурные скобки.

Задача 5. Ограничим алфавит Рефал-символов тремя символами `'a'`, `'b'`, `'_'`. Получившийся язык обозначим Рефал$_{\mathtt{ab_}}$. Натуральные числа в языке Рефал$_{\mathtt{ab_}}$ будем представлять в троичной системе счисления, цифрами в которой являются символы `'_'`, `'a'`, `'b'` (`'_'` есть 0, `'a'` есть 1, `'b'` есть 2).

Рассмотрим следующую кодировку понятия разделения аргументов. Пусть даны два текста-аргумента `X` и `Y`; вычислим длину k первого аргумента `X`. Пусть $N(k)$ – представление числа k в троичной системе счисления с цифрами `'_'`, `'a'`, `'b'`. Рассмотрим пару $N(k)$ и `XY`, где мы текст `Y` приписали к тексту `X` без какого-либо разделителя между ними. Закодируем пару $N(k)$ и `XY` посредством удвоения букв в тексте $N(k)$ и отделением первого элемента пары от второго двумя неравными буквами (как мы описали выше в нашей лекции).

Докажите, что построенное отображение, кодирующее два текста–аргумента `X` и `Y`, инъективно.

а). Реализуйте описанную кодировку на Рефале.

б). Реализуйте описанную кодировку на Рефале, не используя структурные скобки.

в). Реализуйте описанную кодировку на языке Рефал$_{\texttt{ab_}}$.

г). Реализуйте описанную кодировку на языке Рефал$_{\texttt{ab_}}$, не используя структурные скобки.

Список литературы

[1] 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.

[2] V. F. Turchin, D. V. Turchin, A. P. Konyshev, A. P. Nemytykh, *Refal-5: Sources, Executable Modules*, ([online]: http://www.botik.ru/pub/local/scp/refal5/), 2000.

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

[4] А. В. Корлюков, *Введение в программирование на языке РЕФАЛ с приложениями в алгебре*, http://www.refal.net/~korlukov/refbook/index.htm, 2001.

[5] Н. Н. Воробьев, *Числа Фибоначчи*, Наука, Москва, 1978.

[6] Э. Мендельсон, *Введение в математическую логику*, Наука, Москва, 1971.

[7] А. Л. Семёнов, *Математика текстов*, МЦНМО, Москва, 2002.

[8] А. Шень, *Программирование: теоремы и задачи*, (доступна как zipped PDF-файл: ftp://ftp.mccme.ru/users/shen/progbook2/progbookpdf.zip), МЦНМО, Москва, 2004.

[9] И. А. Николаев, *Словарь по литературоведению*, ([online]: http://nature.web.ru/litera/11.2.html), Научная Сеть, 2004.

[10] В. Солоухин, *Стихотворения*, (цитируемое стихотворение доступно в файле: http://botik.ru/pub/local/scp/ugp/venok.html), Советская Россия, Москва, 1990.

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

А. П. Немытых

Заметка о переносе реализации Рефала-5 на операционную систему Windows Mobile 5.0

Мы сообщаем о первом опыте переноса реализации функционального языка программирования Рефал на операционную систему, используемую на сотовых теле/smart-фонах. Библ. 8 наим.

Работа выполнена в ИПС РАН в 2008 году.

§ 1. Введение

Реализация Рефала-5 [3] (автор диалекта В. Ф. Турчин [1]) представляет собой полуинтерпретатор: Рефал-программа компилируется на некоторый промежуточный язык и далее результат компиляции интерпретируется.

К достоинствам этой реализации следует отнести компактность, простоту установки, наличие пошагового отладчика программ. Недостатком является отсутствие поддержки внутреннего оконного интерфейса: программы исполняются из командной строки операционной системы, и пользователь должен редактировать программы посредством редакторов общего назначения, не ориентированных на синтаксис Рефала-5. Другими словами, Рефал-5 является консольным приложением.

Документацией к данной реализации является книга В. Ф. Турчина на английском языке [1]. На странице [3] можно читать эту книгу, находясь в сети, либо скопировать архив на ваш компьютер. В online версии книги вы можете исполнять (и изменять) Рефал-программы прямо в интернете – по ходу чтения книги. Например, при первом знакомстве с языком – ещё до установки реализации на вашем компьютере. На странице [2] имеется русский перевод устаревшей версии книги В. Ф. Турчина. Она также может оказаться очень полезной для первого знакомства с Рефалом-5. В этой версии книги используется устаревший синтаксис Рефала-5.

На электронной странице Рефала-5 [3] свободно распространяются исполняемые модули компилятора и интерпретатора, которые могут исполняться под современными версиями операционной системы `Windows`, предназначенными для «настольных» (не «карманных») вычислительных машин. Исходные тексты реализации также можно найти на этой странице. Вместе с исходными текстами свободно распространяются и `make`-файлы, с помощью которых можно собрать исполняемые модули под различными вариантами операционной системы `Linux`, под операционной системой `FreeBSD`, а также под некоторыми другими системами. Кроме того, со страницы [3] можно скопировать ещё один компилятор Рефала-5 [8], который выдает более подробные сообщения об ошибках, что немаловажно для начинающего пользователя Рефала.

В данной заметке мы сообщаем о первом опыте переноса реализации функционального языка программирования Рефал на операционную систему, используемую на «карманных» вычислительных машинах. Автор перенес реализацию Рефала-5 на операционную систему `Windows Mobile 5.0`. Перенесены и компилятор Рефала-5 в промежуточный язык (`RSL`), и интерпретатор этого языка. На странице [6] мы выложили только исполняемый модуль интерпретатора, предполагая, что крайне неудобно разрабатывать и отлаживать возможные Рефал-приложения для сотовых телефонов непосредственно на самих этих телефонах. `RSL`-модуль, построенный компилятором Рефала-5 на «настольной» вычислительной машине, может быть использован на «карманной». Особенности указанного интерпретатора рассматриваются в нижеследующем разделе.

§ 2. Об особенностях реализации Рефала-5 на операционной системе Windows Mobile 5.0

2.1. Об исходных текстах.

Стартовая версия реализации Рефала-5 была разработана Дмитрием В. Турчиным (г. Нью-Йорк, вторая половина 1980-х гг.) под руководством Валентина Федоровича. Языком реализации являлся `C` (язык `C++` еще не существовал). Все последующие (до рассматриваемой в данной статье) версии реализации транслировались компиляторами классического языка `C`. В 1990-х годах В. Ф. Турчин передал исходные тексты Рефала-5 в институт программных систем РАН (г. Переславль-Залесский) – для поддержки и дальнейшего развития. В 2000 году автор Рефала-5 принял решение о внесении изменений в синтаксис языка [1]. Соответствующие новому синтаксису изменения были реализованы Александром Конышевым. Версия Рефала-5 с измененным синтаксисом называется `Refal-5 Version-PZ` [3].

Адаптация исходных текстов Рефала-5 для системы `Windows Mobile 5.0` производилась в среде `Microsoft Visual Studio 2005 Version 8.0.50727.42` (`RTM.050727-4200`) с использованием компилятора `Microsoft Visual C++ 2005`. Таким образом, исходные тексты Рефала-5 впервые транслировались компилятором `C++`, который считает, например, некоторые конструкции классического языка `C` устаревшими. Кроме того, исправлялись некоторые функции, ответственные за общение с операционной системой, например, запрос и обработка времени. Для этих целей была использована библиотека `wce` (`wcelibcex.lib`).

2.2. Некоторые свойства операционной системы Windows Mobile 5.0 и инструменты тестирования.

Единственным физическим инструментом тестирования, которым обладал автор заметки на момент переноса, был "Smartphone htc P3300". При отладке активно использовался эмулятор «карманной» вычислительной машины `Pocket PC 2003 Second Edition (ARM4)`.

По идеологии построения, система `Windows Mobile 5.0` (direct touch technology) не предполагает (*a priori*) не только существования в её среде какой-либо консоли, но даже понятия командной строки. С другой стороны,

как уже отмечено выше, интерпретатор Рефала-5 является консольным приложением. Следовательно, для использования Рефала-5 необходимо установить консоль управления. При отладке и тестировании рассматриваемой реализации мы использовали `MSCONSOLE` (`Windows CE Console Copyright (C) 2004 Microsoft Corp.`).

Другим свойством системы `Windows Mobile 5.0`, которое пришлось встраивать в интерпретатор Рефала-5, является отсутствие поддержки переменных среды. Отсюда, например, следует, что понятие текущей директории не определено; встроенные функции Рефала-5 `GetCurrentDirectory` и `GetEnv` в обсуждаемой версии также фактически не определены – они всегда возвращают пустое выражение. По той же причине, при вызове из программы на Рефале-5 встроенных функций ввода/вывода необходимо явно указывать полные имена файлов, включающие путь от корневого каталога.

Существует две возможности размещения бинарных модулей Рефала-5 (как `*.exe`, так и `*.rsl` файлы) в файловой системе `Windows Mobile 5.0`:

а). В директории, откуда запускается Рефал-5 приложение пользователя.

- В этом случае, если в командной строке, запускающей интерпретатор на выполнение, указан ключ `-p`, тогда ко всем именам загружаемых `rsl`-модулей автоматически приписывается приставка – путь из корня \ к директории, из которой была отдана команда на исполнение интерпретатора Рефала-5.

б). В системных директориях операционной системы `Windows Mobile 5.0`, которыми являются \ (корень) и `\windows`.

- В этом случае в командной строке запуска интерпретатора необходимо явно указывать пути к тем загружаемым `*.rsl`-модулям, которые находятся в других (не системных каталогах), независимо от того, из какого каталога отдана команда на исполнение интерпретатора Рефала-5.

2.3. О разработке Рефал-5 приложений.

Мы предполагаем, что возможная разработка Рефал-5 приложений и их (частичная) отладка для «карманных» вычислительных машин (компиляция Рефал-программ в промежуточный язык – подготовка `rsl`-модулей) будут проводиться на стандартном «настольном» железе. Окончательную отладку можно производить и на мобильных устройствах, используя компилятор Рефала-5 `crefal` [8], написанный на Рефале-5.

Подготовленные `rsl`-модули нужно скопировать на сотовое устройство посредством программы `ActiveSync`, поддерживающей синхронизацию этих устройств, работающих под управлением `Windows Mobile`, с компьютером. При этом `ActiveSync` должна исполняться в режиме, позволяющем автоматически преобразовывать файлы в процессе синхронизации и копирования. Обычно такой режим исполнения действует «по умолчанию».

§ 3. Заключение

Рассмотренная выше версия реализации Рефала-5 [6] для `Windows Mobile 5.0` была протестирована посредством работы суперкомпилятора `SCP4` [5], [7] – самой большой из существующих на момент написания данной статьи Рефал-программ.

Для реальной разработки мобильных Рефал-приложений необходимо развитие/расширение библиотеки встроенных функций Рефала-5, которые позволяли бы принимать и обрабатывать информацию о внешних событиях, приходящих из сотового эфира.

Список литературы

[1] 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.

[2] В. Ф. Турчин, *РЕФАЛ-5. Руководство по программированию и справочник*, ([online]: http://www.refal.org/rf5_frm.htm русский перевод *устаревшей* версии книги [1].).

[3] 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.

[4] А. В. Корлюков, *Введение в программирование на языке РЕФАЛ с приложениями в алгебре*, ([online]: http://www.refal.net/~korlukov/refbook/index.htm), 2001.

[5] А. П. Немытых, *Суперкомпилятор SCP4: общая структура*, Издательство УРСС, Москва, 2007.

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

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

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

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

А. Н. Непейвода

Отношение Турчина и аппроксимация циклов при анализе программ

В статье рассматривается отношение Турчина на последовательностях, порожденных префиксными грамматиками, описываются его важнейшие свойства и показываются возможности применения данного отношения при автоматическом анализе программ. Библ. 15 наим.

Содержание

§ 1. Поиск зацикливаний при автоматическом анализе программ

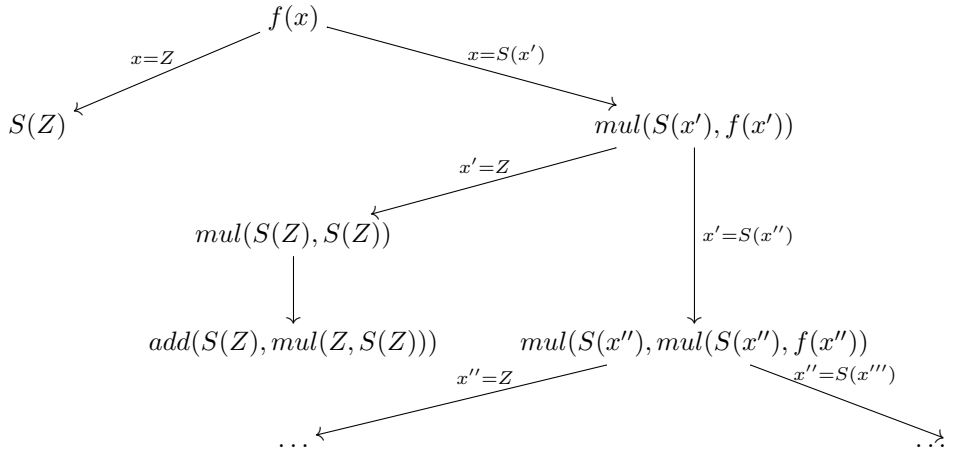
Существует два существенно различных подхода к анализу программ, называемые офлайн- и онлайн-анализом. Офлайн-анализ программы – это попытка определить ее свойства посредством анализа кода, без его интерпретации. Онлайн-анализ интерпретирует работу программы на некотором наборе входных данных и пытается определить ее свойства посредством анализа ее поведения за конечное число шагов. В суперкомпиляции, в частности, по ходу интерпретации программы делается попытка аппроксимации ее поведения на всех возможных путях развития – граф, представляющий последние, называется семантическим деревом программы.

Определение 1. ***Семантическое дерево программы*** – это дерево путей исполнения данной программы на всех данных, входящих в область ее определения.

Пример 1. Пусть дана программа вычисления факториала (в первой колонке таблицы приведена Рефал-версия программы, во второй – та же программа на простом функциональном языке типа Scala).

```
F {
   'Z' = 'S';                                   f(Z) = S(Z);
   'S' e.1 = <Mul ('S'e.1) <F e.1>>;            f(S(x)) = mul(S(x),f(x));
  }
Mul {
  ('Z') e.2 = 'Z';                              mul(Z,y) = Z;
  ('S'e.1) e.2 = <Add (<Mul (e.1)e.2>) e.2>;    mul(S(x),y)=add(mul(x,y),y);
    }
Add {
  ('Z') e.2 = e.2;                              add(Z,y) = y;
  ('S' e.1) e.2 = 'S'<Add (e.1)e.2>;            add(S(x),y) = S(add(x,y));
    }
```

Если вычисляется выражение $f(x)$, где величина x неизвестна, первые несколько шагов развертки семантического дерева при аппликативном порядке вычислений будут выглядеть следующим образом (ниже приведена развертка программы в версии Scala):



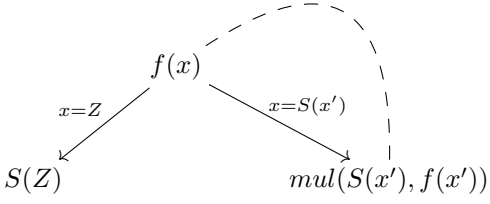
Задав область определения $x \in \{Z, S(Z), \ldots, S(S(S(S(S(Z)))))\}$, можно построить полную развертку семантического дерева этой программы. Это возможно и в любом другом случае, когда область определения конечна (поскольку исходная программа завершается на любых данных). Однако если не предполагать, что значение x ограничено, в дереве появляются ветви бесконечной длины, и за конечное время построить его явным образом не получается.

Заметим, что бесконечный путь в семантическом дереве программы всегда соответствует синтаксическому циклу (или, возможно, рекурсии). Однако наличие синтаксического цикла в коде программы не обязательно влечет наличие бесконечного пути в семантическом дереве. Например, следующий синтаксический цикл не порождает бесконечного пути (`x` — переменная типа целое число):

```
while (x mod 2 = 0) do
действия, не содержащие циклов и не меняющие значение x
x:=x+1;
od
```

Чтобы научиться отличать семантические циклы от подобных конечных конструкций, синтаксически выглядящих как цикл, в общем случае пришлось бы решить алгоритмически неразрешимую проблему останова произвольной машины Тьюринга. Поэтому приходится выбирать подходящее приближение разрешающей процедуры определения семантических циклов. В частности, простейший из них проверяет, превышает ли длина ветви развертки (пути в семантическом дереве программы) определенную фиксированную величину, и если да, то завершает развертку этой ветви. Однако в примере 1 выше подобный критерий может сработать, даже если область определения x конечна (и, следовательно, циклы отсутствуют), но содержит достаточно большое натуральное число. Другой подход к построению приближенного решения (например, если мы ограничиваемся анализом примитивно-рекурсивных программ) может обрывать лишь те ветви прогонки, длина которых превысила значение некоторой функции от длины входных данных, ограниченной снизу функцией Аккермана, но при этом анализ программы вычисления факториала на неопределенных входных данных потребует недопустимо большого количества шагов. Чтобы избежать подобных неприятностей, в онлайн-анализе применяются относительные критерии определения семантических циклов на ветви, называемые также критериями вложения (embeddings). Такой относительный критерий вложения сравнивает два терма, стоящих в узлах $Node_1$ и $Node_2$, (чаще всего таких, что $Node_1$ – предок $Node_2$), и оповещает об опасности зацикливания, если структура терма в $Node_2$ некоторым образом повторяет структуру терма в $Node_1$.

Вернемся к примеру 1 и посмотрим на начальный фрагмент его семантического дерева с точки зрения вложений.



$f(x)$ – подтерм терма $mul(S(x'), f(x'))$ с точностью до переименования переменных. На основании этого наблюдения можно сделать вывод, что вычисление $f(x')$ в $mul(S(x'), f(x'))$ будет повторять структуру вычисления $f(x)$, и сослаться на эту структуру при развертке узла, содержащего $mul(S(x'), f(x'))$, после чего разворачивать уже только $mul(S(x'), y)$, где y – результат вычисления $f(x')$. При этом бесконечное дерево превращается в конечный граф. Заметим, что если имеется дополнительная информация об исходных данных (например, исходный вызов выглядит не как $f(x)$, а как $f(S(S(x)))$), подобное точное вложение не появится при первых двух вызовах f, а если аргумент f

полностью определен, в семантическом дереве программы его вычисления не будет такого вложения ни на одной ветви. Так, пользуясь этим наблюдением, мы избежали и слишком поспешного завершения развертки, и чрезмерно большой вычислительной сложности алгоритма этой развертки. Однако, как заметил, в частности, В. Ф. Турчин [13], подобные точные вложения возникают на бесконечных путях в семантических деревьях далеко не всегда. Если мы хотим, чтобы алгоритм онлайн-анализа программ всегда завершался, потребуется ослабление точного вложения, такое, что на всякой бесконечной ветви существуют два узла, данные которых ему удовлетворяют. Подобные отношения называются почти полными.

Определение 2. Отношение R, $R \subset S \times S$, называется ***почти полным***, если всякая последовательность $\{A_n\}$ элементов S такая, что $\forall i, j (i < j \Rightarrow (A_i, A_j) \notin R)$, конечна. Если отношение R почти полно и транзитивно, то оно называется ***полным квазипорядком*** (wqo).

Последовательности $\{A_n\}$ такие, что $\forall i, j (i < j \Rightarrow (A_i, A_j) \notin R)$, называются ***плохими последовательностями*** относительно R.

Один из самых изученных полных квазипорядков, обобщающих отношение точного вложения и применяемых в качестве критерия завершения развертки ветви семантического дерева по вложению, – это отношение Хигмана–Крускала вложения с разрывами.

Определение 3. Пусть даны два слова в алфавите Υ: $A = a_1 a_2 \dots a_m$ и $B = b_1 b_2 \dots b_n$. A вкладывается в B ***по Хигману*** ($A \trianglelefteq B$), если A является подпоследовательностью B. Пары A, B такие, что $A \trianglelefteq B$, также будем называть ***хигмановыми парами***.

Отношение Крускала обобщает отношение Хигмана на деревья (скобочные структуры). Здесь мы не приводим точного определения этого обобщения, поскольку далее будем рассматривать лишь отношения на словах.

Вложение Хигмана–Крускала является полным квазипорядком ([4], на деревьях см. [6]) и поэтому оказалось привлекательно как возможный разумный критерий обрыва ветвей в семантическом дереве. Само по себе ([10], [3]), с дополнительной разметкой (по глубине) [5], или в композиции с другими полными квазипорядками [1] оно применялось в преобразовании программ как критерий аппроксимации циклов. Однако существенно важным условием, делающим отношение Хигмана пригодным для этого, является не транзитивность и даже не почти полнота на всем множестве слов из Υ^*, а почти полнота на множестве слов, являющихся возможными данными узлов семантического дерева программы. Мы рассмотрим особое множество буквенных последовательностей – а именно, последовательности, порожденные префиксными грамматиками.

§ 2. Префиксные грамматики и моделирование поведения стека функций

Определение 4. Тройка $\langle \Upsilon, \mathbf{R}, \Gamma_0 \rangle$, где Υ – алфавит, $\Gamma_0 \in \Upsilon^+$ – начальное слово, а $\mathbf{R} \subset \Upsilon^+ \times \Upsilon^*$ – конечный набор правил переписывания[1], называется ***префиксной грамматикой***, если правила $R : R_l \to R_r$ применимы лишь к словам вида $R_l\Phi$ (здесь R_l – префикс, а Φ – суффикс, возможно, нулевой длины) и порождают слова вида $R_r\Phi$.

Если длина левой части R_l всех правил $R : R_l \to R_r$ равна 1 (всякое правило изменяет лишь первую букву), префиксная грамматика называется ***алфавитной префиксной грамматикой***.

Цепочка, порожденная префиксной грамматикой $\mathbf{G} = \langle \Upsilon, \mathbf{R}, \Gamma_0 \rangle$, есть последовательность $\{\Phi_i\}$ (конечная или бесконечная) такая, что $\Phi_1 = \Gamma_0$ и для всех i, не превышающих длины этой последовательности, $\exists R(R : R_l \to R_r \ \& \ R \in \mathbf{R} \ \& \ \Phi_i = R_l\Theta \ \& \ \Phi_{i+1} = R_r\Theta)$ (Θ – суффикс, возможно, нулевой длины). Иными словами, элементы цепочки получаются из их предшественников непосредственным применением правил грамматики $\mathbf{G}$.

Пример 2. Рассмотрим префиксную грамматику с алфавитом $\Upsilon = \{a, b\}$ и следующими правилами:

$$R^{[1]} : a \to bb \qquad R^{[2]} : aa \to b \qquad R^{[3]} : b \to aa$$

Правило $R^{[3]}$ неприменимо к слову *aabaab*, поскольку оно не начинается с буквы b; к *aabaab* можно применить $R^{[1]}$ либо $R^{[2]}$, и единственно правильными результатами соответствующих применений будут слова *bbabaab* и *bbaab*.

Префиксными грамматиками удобно описывать развитие стека функций. В работе [2] описано соответствие префиксных грамматик автоматам с магазинной памятью – однако следует заметить, что речь в ней идет не о порождающих, а о распознающих префиксных грамматиках, правила переписывания в которых зависят не только от переписываемой строки (состояния стека), но и от входных данных (распознаваемого слова). При этом входные данные остаются неизменными, и прочитав букву, к ней, как правило, нельзя вернуться. О случае, когда имеется прямое произведение префиксных грамматик (меняются и стек, и данные), кратко будет сказано ниже. Пока же нас интересуют недетерминированные порождающие префиксные грамматики и их способность моделировать поведение стека функций при онлайн-анализе программы.

При решении задачи поиска суперкомпиляцией семантических циклов в программе В. Ф. Турчин рассматривал понятие преобразования стека функций, похожее на префиксную грамматику [13]. Поскольку функции в Рефале-5 формально одноместны, в первом приближении можно представить структуру хранимых функциональных вызовов не деревом, а словом; поскольку Рефал-программы исполняются аппликативно (а не лениво), это слово можно считать состоящим не из буквенных блоков, а из отдельных букв.

Пример 3. Построим префиксную грамматику, описывающую поведение стека функций при исполнении Рефал-программы примера 1. В первой колонке

[1] Вообще говоря, обычно также оговаривается конечность Υ, но это требование в данном случае можно опустить. Все наши построения существенно используют допущение о конечности только множества правил переписывания $\mathbf{R}$.

таблицы приведена исходная программа, во второй – правила соответствующей грамматики (Λ – пустое слово).

```
F { 'Z' = 'SZ';                                          f → Λ
    'S' e.1 = <Mul ('S' e.1) <F e.1>>; }                 f → fm

Mul { ('Z') e.2 = 'Z';                                   m → Λ
      ('S' e.1) e.2 = <Add (<Mul (e.1)e.2>) e.2>; }      m → ma

Add { ('Z') e.2 = e.2;                                   a → Λ
      ('S'e.1) e.2 = 'S' <Add (e.1) e.2>; }              a → a
```

Допустим, нам требуется вычислить `<Add <F e.x> e.x>`. В этом случае $\Gamma_0 = fa$, и начальный отрезок цепочки, соответствующей бесконечной ветви семантического дерева при неопределенном аргументе `e.x`, выглядит так:

fa
fma
$fmma$
$fmmma$
...

Из соображений компактности мы также будем записывать цепочки в строчку, используя знак $\to$:

$fa \to fma \to fmma \to fmmma \to \ldots$

В данном случае грамматика строится по программе однозначно. Если бы код начинался следующим образом:

```
F { 'Z' = 'SZ';
    'S' e.1 = <Mul (<Mul ('S' e.1) e.1>) <F e.1>>;
  }
```

возникла бы неоднозначность с переписыванием f из-за равноправного положения двух функций–аргументов внешнего умножения: соответствующее правило могло бы выглядеть как $f \to mfm$ или как $f \to fmm$. Обычно при вычислении функций в стеке навязывается порядок (при прочих равных) слева направо, поэтому поведению интерпретатора больше соответствует первый вариант.

Аппликативность Рефала сильно упрощает задачу моделирования грамматики, представляющей поведение стека функций. Чтобы получить искомую грамматику, требуется совершить следующие два действия:

1. Сопоставить каждому имени функции букву из Υ, не обращая внимание на местность соответствующих функций.
2. Сопоставить каждой строке программы правило переписывания: однобуквенное слово из Υ^+, представляющее имя описываемой функции, переписывается в слово, состоящее из букв, которые представляют имена вызываемых функций, в порядке от самой внутренней до самой внешней (функции, находящиеся на одном уровне, могут следовать друг за

другом в произвольном порядке, однако, как мы уже отмечали, обычно исполняются в порядке слева направо).

Поскольку в базисном Рефале нет функциональных вызовов в левых частях определений [12], таким образом получается алфавитная префиксная грамматика. Описанный выше алгоритм дает корректную, но неполную картину поведения стека функций, поскольку пренебрегает контекстами их вызовов, и при прямом его применении следующий фрагмент кода

```
F { 'Z' = <G 'SZ'>;
    'S' e.1 = <G e.1 e.1>;
  }
```

вырождается в единственное правило $f \to g$. Ниже мы обсудим, как частично избавиться от подобных неточностей при отображении кода в грамматику.

При попытках найти семантический цикл в программе отслеживать (не обращая поначалу внимания на данные) состояние стека функций может быть полезно, чтобы исключить ситуации, когда вершина стека проходит один и тот же путь развития неограниченное количество раз, в конце этого пути повторяя сама себя. Тогда в цепочке, порождаемой соответствующей префиксной грамматикой, встретятся слова `[Top][Context]`, `[Top][Middle][Context]`, ..., `[Top]`$[$`Middle`$]^n$`[Context]` (квадратными скобками здесь выделены фрагменты слова). В примере 3 наблюдалась именно такая картина: роль `[Top]` играло слово f, `[Middle]` – m, `[Context]` – a. При этом нигде на бесконечной ветви семантического дерева не возникает отношения подтерма (с точностью до переименования переменных), из-за внешнего вызова функции `Add`. Можно заметить, что термы, возникающие на этом пути, вкладываются друг в друга по Хигману, и вообще, `[Top]`$[$`Middle`$]^m$`[Context]` $\trianglelefteq$ `[Top]`$[$`Middle`$]^n$`[Context]`, если $m \leqslant n$, и мы обсудим вопрос использования отношения Хигмана для определения подобных случаев, однако оказывается удобно подойти к нему не напрямую, а посредством отношения, описанного В. Ф. Турчиным в [13] и затем более формально – А. П. Немытых в [8].

§ 3. Отношение Турчина

Заметим, что кусок стека `[Context]` в предыдущем примере бесконечного развития стека оставался неизменным. Чтобы формализовать это свойство, применим подход, описанный в [8], использующий понятие временных индексов. Длину слова Φ здесь и далее обозначаем $|\Phi|$.

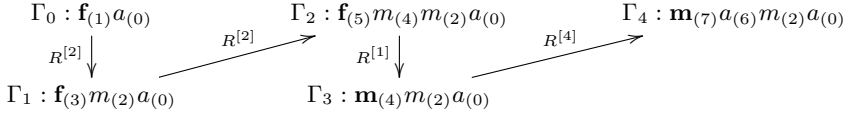
Определение 5. Рассмотрим цепочку $\{\Phi_i\}$, порожденную префиксной грамматикой $\mathbf{G} = \langle \Upsilon, \mathbf{R}, \Gamma_0 \rangle$. Припишем ко всем буквам слов Φ_i числа, соответствующие их ***временным индексам***, следующим образом. i-я буква слова Γ_0 помечается индексом $|\Gamma_0| - i$; если наибольший временной индекс в отрезке цепочки $\{\Phi_i\}_{i=1}^k$ есть M и Φ_{k+1} получается из Φ_k применением правила $R : R_l \to R_r$, то i-я буква Φ_{k+1} ($i \leqslant |R_r|$) помечается индексом $M + |R_r| - i + 1$. Временные индексы остальных букв Φ_{k+1} остаются такими же, как в Φ_k.

Это формально сложное определение имеет простой смысл. Будем просматривать нашу цепочку, выписывая каждое слово побуквенно справа налево, по-арабски, и помечать ее тем моментом времени, в который она впервые появилась в цепочке. Разметку, производимую таким образом, будем называть временным индексированием, а цепочку, размеченную временными индексами – ***вычислением***.

Пример 4. Обратимся к грамматике стека программы вычисления факториала $\mathbf{G_F}$ из примера 3 (Λ – пустое слово).

$$\begin{array}{lll} R^{[1]} : f \to \Lambda & R^{[3]} : m \to \Lambda & R^{[5]} : a \to \Lambda \\ R^{[2]} : f \to fm & R^{[4]} : m \to ma & R^{[6]} : a \to a \end{array}$$

$\Gamma_0 = fa$. Начальный отрезок вычисления, порожденного этой грамматикой, может выглядеть, например, так (жирным выделена буква, подлежащая очередному переписыванию):



(По поведению стека можно восстановить входные данные программы – это строка `'SSZ'`, то есть 2 в унарной системе счисления).

Будем обозначать прописными греческими буквами (Γ, Δ, Θ, Ψ, Φ) слова, буквы в которых размечены временными индексами. $\Delta[k]$ будет обозначать k-ю букву в слове Δ, считая от начала. Если Δ – слово, то Δ^- – это же слово без первой буквы; $\Delta[|\Delta|]$ – для удобочитаемости пишем как $\Delta[last]$ – последняя буква Δ.

Определим отношение эквивалентности слов с точностью до временных индексов их букв. $\Phi \approx \Psi$, если $|\Phi| = |\Psi|$ и $\forall i (i \geqslant 1 \;\&\; i \leqslant |\Phi| \Rightarrow (\Phi[i] = a_{(n)} \;\&\; \Psi[i] = b_{(m)} \Rightarrow a = b))$.

Теперь мы, наконец, готовы определить ***отношение Турчина*** $\Gamma \preceq \Delta$.

Определение 6. $\Gamma \preceq \Delta \Leftrightarrow \Gamma = \Phi\Theta_0 \;\&\; \Delta = \Phi'\Psi\Theta_0 \;\&\; \Phi' \approx \Phi$. Пары Γ, Δ такие, что $\Gamma \preceq \Delta$, будем называть ***турчинскими парами***[2].

Неформально это означает как раз то, что Γ и Δ можно разрезать таким образом, что $\Gamma = [\texttt{Top}][\texttt{Context}]$, а $\Delta = [\texttt{Top}][\texttt{Middle}][\texttt{Context}]$, причем так, что суффикс `[Context]` оставался неизменным на всем отрезке цепочки от Γ до Δ (включая и Δ).

Перечислим пары слов цепочки из примера 4, связанные отношением Турчина: $\Gamma_0 \preceq \Gamma_1$, $\Gamma_0 \preceq \Gamma_2$, $\Gamma_1 \preceq \Gamma_2$ и $\Gamma_3 \preceq \Gamma_4$.

Предложение 1. $\preceq$ – рефлексивное антисимметричное с точностью до $\approx$, но не транзитивное отношение.

[2] В [8] также оговаривается, что $|\Phi| > 0$. Если цепочки строятся в алфавитной префиксной грамматике, да и вообще, в любой грамматике, содержащей лишь правила с непустой левой частью, эту оговорку можно не делать, но если разрешить правила с пустой левой частью, без нее становится неверна теорема 1 этой статьи, и верхняя оценка на длину плохой последовательности становится неточной.

Доказательство.

1. Рефлексивность. Достаточно взять $\Psi = \Lambda$.
2. Антисимметричность с точностью до $\approx$. Пусть $\Gamma = \Phi_{(i)}\Theta = \Phi'_{(j)}\Psi'\Theta'$, а $\Delta = \Phi_{(k)}\Psi\Theta = \Phi'_{(l)}\Theta'$, где $\Phi_{(i)} \approx \Phi_{(k)}$ и $\Phi'_{(j)} \approx \Phi'_{(l)}$.
   $|\Phi_{(i)}| + |\Theta| = |\Phi'_{(j)}| + |\Psi'| + |\Theta'|$ и $|\Phi'_{(j)}| + |\Psi'| + |\Theta'| \geqslant |\Phi'_{(l)}| + |\Theta'|$, поэтому $|\Phi_{(i)}| + |\Theta| \geqslant |\Phi_{(k)}| + |\Psi| + |\Theta|$ и, аналогично, $|\Phi'_{(j)}| + |\Psi'| + |\Theta'| \leqslant |\Phi'_{(l)}| + |\Theta'|$, следовательно, $\Psi = \Psi' = \Lambda$
3. Отсутствие транзитивности. Рассмотрим грамматику

   $\mathbf{G_{ABC}}$:

   $R^{[1]} : a \to \Lambda$ $\quad$ $R^{[3]} : d \to \Lambda$

   $R^{[2]} : a \to ad$ $\quad$ $R^{[4]} : b \to fbe$

   $R^{[5]} : f \to ad$

   Развернем вычисление, начинающееся с abc.

$$\Gamma_0 : \mathbf{a}_{(2)}b_{(1)}c_{(0)}$$
$$\downarrow R^{[2]}$$
$$\Gamma_1 : \mathbf{a}_{(4)}d_{(3)}b_{(1)}c_{(0)}$$
$$\downarrow R^{[1]}$$
$$\Gamma_2 : \mathbf{d}_{(3)}b_{(1)}c_{(0)}$$
$$\downarrow R^{[3]}$$
$$\Gamma_3 : \mathbf{b}_{(1)}c_{(0)}$$
$$\downarrow R_{[4]}$$
$$\Gamma_4 : \mathbf{f}_{(7)}b_{(6)}e_{(5)}c_{(0)}$$
$$\downarrow R_{[5]}$$
$$\Gamma_5 : \mathbf{a}_{(9)}d_{(8)}b_{(6)}e_{(5)}c_{(0)}$$

   $\Gamma_0 \preceq \Gamma_1$ и $\Gamma_1 \preceq \Gamma_5$, но $\Gamma_0 \not\preceq \Gamma_5$.

□

Далее мы покажем почти полноту отношения Турчина на вычислениях, порожденных алфавитными префиксными грамматиками – это утверждение есть теорема Турчина [13], и мы докажем ее в полностью абстрактной форме. Из доказательства мы также извлечем верхнюю оценку на длину плохой последовательности, порожденной префиксной грамматикой, и покажем, в каких грамматиках она достигается, а также обсудим практический смысл перехода к таким грамматикам. Затем мы построим полный квазипорядок, являющийся подмножеством отношения Турчина, и обобщим теорему о почти полноте последнего на более широкий класс грамматик. Наконец, поговорим о связи отношения Хигмана и отношения Турчина, и какие выводы можно сделать о первом, исследуя свойства второго.

§ 4. Почти полнота отношения Турчина

Пусть Υ – алфавит, $\mathbf{G} = \langle \Upsilon, \mathbf{R}, \Gamma_0 \rangle$, $\mathbf{R} \subset \Upsilon \to \Upsilon^*$ – алфавитная префиксная грамматика с конечным набором правил.

Заметим, что если в вычислении, порожденном $\mathbf{G}$, Φ предшествует Ψ и притом существует i такое, что $\forall j(\Phi[i] \neq \Psi[j])$, то в отрезке цепочки между Φ и Ψ встречается слово длины не более чем $|\Phi| - i + 1$. Действительно, чтобы стереть $\Phi[i]$, нужно сперва до него добраться, а это возможно лишь получив слово $\Phi[i]\Phi[i+1]\ldots\Phi[last]$.

Правило $R : a \to R_r$ с непустой правой частью будем называть ***неукорачивающим***; длиной правила будем называть длину слова в его правой части $|R_r|$. Результат применения R к Δ записывается как $R(\Delta[1])\Delta^-$.

Как только в вычислении появляется пустое слово Λ или находятся такие Δ_1 и Δ_2, что $\Delta_1 \preceq \Delta_2$, будем считать, что вычисление завершается.

Предложение 2. Если R неукорачивающее, $R(\Theta_0[1])\Theta_0^-$ предшествует в цепочке $R(\Theta_1[1])\Theta_1^-$ и $\exists i(\Theta_1^-[i] = \Theta_0^-[1])$, то $R(\Theta_0[1])\Theta_0^- \preceq R(\Theta_1[1])\Theta_1^-$.

Доказательство. После применения R к Θ_0 все последующие слова имеют вид $\Psi[1]\Psi^-\Theta_0^-$ (поскольку $\Theta_0^-[1]$ остается неизменным). Применение R к любому из этих слов порождает слово $R(\Psi[1])\Psi^-\Theta_0^-$, и $R(\Theta_0[1])\Theta_0^- \preceq R(\Psi[1])\Psi^-\Theta_0^-$. □

Пусть R – неукорачивающее правило, примененное к $\Delta[1]$. Будем говорить, что в Δ_2 применение R ***было отменено***, если $\Delta_2[1] = \Delta_1^-[1]$ или в Δ_2 нет буквы $\Delta_1^-[1]$.

Лемма 1. Наибольшая длина слова в плохой последовательности (см. определение 2), порожденной алфавитной префиксной грамматикой с конечным числом правил переписывания[3], не превышает

$$|\Gamma_0| + \sum(|R_r^{[i]}| - 1)$$

где Γ_0 – начальное слово, и $\sum(|R_r^{[i]}| - 1)$ пробегает множество всех неукорачивающих правил.

Доказательство. Чтобы вновь применять неукорачивающее правило R после его применения к Θ_0 и при этом не получить турчинскую пару, необходимо укоротить слово, к которому оно будет применяться, до длины не более $|\Theta_0| - 1$. Таким образом, общая длина слова в плохой последовательности при переиспользовании правила уменьшается хотя бы на 1, а наибольшая длина не может превосходить длину слова, к которому были ровно по одному разу применены все неукорачивающие правила (и ни разу не применены стирающие). □

Теперь можно дать очень простое доказательство теоремы Турчина.

[3] Не обязательно она может быть достигнута; например, в грамматике примера 4 наибольшая длина слова в плохой последовательности равна 3.

Предложение 3 (Теорема Турчина).
Каковы бы ни были алфавитная префиксная грамматика $\mathbf{G}$ с конечным числом правил, вычисление, порождаемое ими, не будет содержать бесконечных плохих последовательностей в смысле отношения $\preceq$.

Доказательство. Длина слова в плохой последовательности ограничена и множество букв, стоящих в правых частях правил переписывания из $\mathbf{G}$, конечно, поэтому существует конечное число классов эквивалентности по $\approx$ слов, допустимых в плохой последовательности. Как только ее длина станет больше этого числа, какие-то два слова попадут в один класс эквивалентности, либо найдется слово длины, большей, чем допустимая. $\square$

Это доказательство теоремы Турчина сразу же позволяет дать первую грубую оценку максимальной длины антицепи в грамматике. Она не больше, чем количество различных слов в алфавите Υ, имеющих длину, не большую, чем $|\Gamma_0|+\sum(|R_r^{[i]}|-1)$. Обозначим размер алфавита Υ как $card(\Upsilon)$. В случае, если $card(\Upsilon)>1$, оценка имеет вид

$$C_{Max} = card(\Upsilon)^{|\Gamma_0|+\sum(|R_r^{[i]}|-1)}$$

В большинстве случаев она явно завышена. Небольшие усилия позволяют получить другую оценку, которая может дать намного более точный результат. И чтобы его получить, нам следует избавиться от случайных свойств грамматик, провоцирующих неинформативное обрывание плохих последовательностей. В частности, обратимся к уже рассмотренному примеру фрагмента программы, переписываемого в единственное правило.

```
F { 'Z' = <G 'SZ'>;
    'S' e.1 = <G e.1 e.1>;
  }
```

Видно, что хотя вызывается одна и та же функция, контексты ее вызовов различны. Если бы мы записали два правила $f \to g^{[Z]}$ и $f \to g^{[S]}$, считая $g^{[Z]}$ и $g^{[S]}$ разными буквами алфавита, то могли бы получить более точное представление поведения программы грамматикой. Формальное определение подобных грамматик приведено ниже.

Определение 7. Алфавитная префиксная грамматика $\mathbf{G}=\langle\Upsilon,\mathbf{R},\Gamma_0\rangle$, $\mathbf{R}\subset\Upsilon\times\Upsilon^*$ называется ***обогащенной***, если

1. Для всяких двух правил $R: a\to R_r, R': b\to R'_r$, если $\exists i,j(R_r[i]\approx R'_r[j])$, то $R_r\approx R'_r$;
2. Если $R^{[a]}\in\mathbf{R}$, $R^{[a]}: a\to R_r$ и $b\in\Upsilon$, то $R^{[b]}: b\to R_r\in\mathbf{R}$.
3. $\forall i,j,k(R_r^{[k]}[i]\neq\Gamma_0[j])$, то есть все буквы начального слова уникальны.

Проще говоря, в обогащенной грамматике правые части правил не содержат общих букв либо совпадают, и любая буква может быть переписана в любую правую часть. Второе условие не обеспечивает более точного представления поведения программы грамматикой, но позволяет освободиться от навязываемой грамматикой детерминированности переписывания: так, например, в обогащенных грамматиках верхняя оценка на длину слова в плохой последовательности $|\Gamma_0|+\sum(|R_r^{[i]}|-1)$ всегда достижима.

Рассмотрим следующее преобразование алфавитной префиксной грамматики $\mathbf{G}$ в обогащенную $\mathbf{G}'$.

1. Пусть $a = R_r[i]$, $a \in \Upsilon$. Заменим a на пару $\langle a, 2^r * 3^{i-1} \rangle$. Начальное слово считаем правилом номер 0.
2. Будем считать правилом $R' : x \to \Phi$ грамматики $\mathbf{G}'$ класс эквивалентности относительно совпадения правых частей правил $\langle a_i, n_i \rangle \to \Phi$, где Φ – правая часть некоторого правила из $\mathbf{G}$ после преобразования, описанного в пункте 1, а $\langle a_i, n_i \rangle$ – произвольная пара.

Пусть исходная грамматика $\mathbf{G}$ порождает плохую последовательность посредством применения набора правил $\{R^{[i]}\}_{i=1}^{n}$. Преобразуем все $R^{[i]} : a \to R_r{}^{[i]}$ в соответствующие классы эквивалентности $\{R'^{[i]}\}$, а Γ_0 – в Γ'_0 по алгоритму, данному выше, и затем применим подходящие правила из $\{R'^{[i]}\}$ к Γ'_0 и далее. Каждые два слова в полученном вычислении будут несравнимы по Турчину, поскольку их проекции на первую координату a пары $\langle a, 2^r * 3^{i-1} \rangle$ несравнимы. Описанная операция преобразования грамматики в обогащенную является идемпотентной: примененная повторно, она породит ту же самую грамматику (с точностью до обозначений), что и в первый раз. Правилами обогащенной грамматики называются классы эквивалентности правил переписывания по правой части, поэтому они будут записываться в общем виде: $R : x \to R_r$.

Пример 5. Переведем алфавитную префиксную грамматику $\mathbf{G_F}$ из примера 4 в обогащенную.

$\mathbf{G'_F}$:

$\Gamma_0 = \langle f, 1 \rangle \langle a, 3 \rangle$ $\qquad$ $R^{[2]} : x \to \langle m, 4 \rangle \langle a, 12 \rangle$

$R^{[1]} : x \to \langle f, 2 \rangle \langle m, 6 \rangle$ $\qquad$ $R^{[3]} : x \to \langle a, 8 \rangle$

$R^{[4]} : x \to \Lambda$

Цепочка из примера 4 соответствует после преобразования грамматики следующей цепочке:

$$\Gamma_0 : \langle \mathbf{f}, \mathbf{1} \rangle_{(1)} \langle a, 3 \rangle_{(0)}$$
$$\downarrow R^{[1]}$$
$$\Gamma_1 : \langle \mathbf{f}, \mathbf{2} \rangle_{(3)} \langle m, 6 \rangle_{(2)} \langle a, 3 \rangle_{(0)}$$
$$\downarrow R^{[1]}$$
$$\Gamma_2 : \langle \mathbf{f}, \mathbf{2} \rangle_{(5)} \langle m, 6 \rangle_{(4)} \langle m, 6 \rangle_{(2)} \langle a, 3 \rangle_{(0)}$$
$$\downarrow R^{[4]}$$
$$\Gamma_3 : \langle \mathbf{m}, \mathbf{6} \rangle_{(4)} \langle m, 6 \rangle_{(2)} \langle a, 3 \rangle_{(0)}$$
$$\downarrow R^{[2]}$$
$$\Gamma_4 : \langle \mathbf{m}, \mathbf{4} \rangle_{(7)} \langle a, 12 \rangle_{(6)} \langle m, 6 \rangle_{(2)} \langle a, 3 \rangle_{(0)}$$

Теперь $\Gamma_0 \not\preceq \Gamma_1$. Это имеет смысл и с точки зрения онлайн-анализа программы: у вызовов `F` в первом и втором случае разная природа, и этот факт теперь отмечен в грамматике.

Замечательное свойство обогащенных грамматик заключается в том, что они способны порождать самые длинные плохие последовательности по Турчину. Отсутствие пересечений в правых частях правил исключает случайные вложения, не обоснованные развитием семантического дерева, а расширение грамматики позволяет всегда выбирать для очередного переписывания то правило, которое позволит плохой последовательности не обрываться как можно дольше. Сейчас мы построим точный алгоритм, позволяющий это сделать.

§ 5. Оценка длины максимальной плохой последовательности по Турчину

Теорема 1. Если наибольшая длина правил обогащенной грамматики не больше, чем k ($k > 1$), и в ней не больше, чем N неукорачивающих правил, то при начальном слове Γ_0 длина наибольшей плохой последовательности равна

$$C'_{Max} = |\Gamma_0| * \frac{k^N - 1}{k - 1}.$$

Теперь докажем теорему 1. Чтобы это сделать, опишем свойства турчинских пар, появляющихся в вычислениях, порожденных обогащенными грамматиками. Мы знаем, что повторное применение неукорачивающего правила без его отмены влечет появление вложения по Турчину (см. раздел 4). Оказывается, что в обогащенных грамматиках плохие последовательности могут обрываться лишь появлением турчинских пар именно такого вида (или появлением Λ).

Теорема 2. В обогащенной грамматике всякая плохая последовательность завершается либо словом Λ, либо появлением пары вида $R(a)\Theta_0$, $R(b)\Psi\Theta_0$, причем R – неукорачивающее правило.

Доказательство. Рассмотрим пару $\Phi_1\Theta_0$, $\Phi_2\Psi\Theta_0$ такую, что $\Phi_1\Theta_0 \preceq \Phi_2\Psi\Theta_0$ ($\Phi_1 \approx \Phi_2$), которой завершается плохая последовательность. $\Phi_1[1]$ и $\Phi_2[1]$ должны порождаться[4] различными применениями R, и если $\Phi_1[1] \approx R_r[i]$, то $\Phi_2[1] \approx R_r[i]$. Обозначим слово $R_r[1]R_r[2]\dots R_r[i-1]$ как $R^{(i-1)}$ и рассмотрим результаты применений R, породивших $\Phi_1[1]$ и $\Phi_2[1]$. Первый имеет вид $R^{(i-1)}_{(k_1)}\Phi_1\Theta_0$, второй – $R^{(i-1)}_{(k_2)}\Phi_2\Psi\Theta_0$. Они образуют турчинскую пару, следовательно, совпадают с $\Phi_1\Theta_0$ и $\Phi_2\Psi\Theta_0$ по предположению о том, что $\Phi_1\Theta_0$, $\Phi_2\Psi\Theta_0$ есть турчинская пара, обрывающая плохую последовательность, следовательно, до появления в цепочке слова $\Phi_2\Psi\Theta_0$ в ней не может быть вложений по $\preceq$.

Итак, $\Phi_1 = R(a)\Phi'_1$, $\Phi_2 = R(b)\Phi'_2$ ($\Phi'_1 \approx \Phi'_2$). Если $\Phi'_1 = \Lambda$, утверждение теоремы доказано. Пусть $\Phi'_1 \neq \Lambda$. Тогда $\exists R', j(\Phi'_1[1] \approx R'_r[j]\ \&\ \Phi'_2[1] \approx R'_r[j])$ и $\Phi'_1[1] \neq \Phi'_2[1]$. Рассмотрим результаты применения R' к предшественникам $\Phi_1\Theta_0$ и $\Phi_2\Psi\Theta_0$. Они выглядят как как $R'^{(j-1)}_{(l_1)}\Phi'_1\Theta_0$ и $R'^{(j-1)}_{(l_2)}\Phi'_2\Psi\Theta_0$ и образуют турчинскую пару. По допущению, что $\Phi_1\Theta_0$, $\Phi_2\Psi\Theta_0$ есть первая турчинская пара в цепочке, получаем $\Phi'_1 = \Lambda$, значит, $\Phi_1\Theta_0 = R(a)\Theta_0$ и $\Phi_2\Psi\Theta_0 = R(b)\Psi\Theta_0$. □

[4] Ни одно из них не может принадлежать начальному слову, поскольку в силу свойств грамматики все буквы начального слова уникальны и не могут порождаться никакими правилами переписывания.

Таким образом, турчинские пары, обрывающие плохую последовательность, порожденную обогащенной грамматикой, могут появиться лишь после повторного появления неукорачивающего правила без его отмены. Теперь возможно описать стратегию построения самой длинной плохой последовательности и оценить длину этой последовательности более точно.

Скажем, что применение $R^{[i]}$ хуже, чем применение $R^{[j]}$, если $R^{[i]}$ позволяет построить более длинную плохую последовательность. Обратим внимание, что речь сейчас идет именно о применениях правил, а не о самих правилах.

Лемма 2. Если некоторое неукорачивающее правило было отменено (либо никогда не применялось), то его применение к Θ_0 хуже, чем укорачивание Θ_0 (применение правила вида $x \to \Lambda$).

Доказательство. Пусть R – неукорачивающее правило, которое было отменено или не применялось. В отрезке вычисления, начиная от $R(\Theta_0[1])\Theta_0^-$ и заканчивая Θ_0^- (отменой применения $R(\Theta_0[1])$), нет слов, таких, которые могли бы образовать вложение со словом, появившимся после Θ_0^-. Докажем это от противного. Допустим, на этом отрезке появился первый элемент турчинской пары $R'(a)\Theta_1$, и запишем второй элемент турчинской пары, в который вкладывается первый, как $R'(b)\Psi\Theta_1$. $\Theta_0^-[1]$ неизменна до отмены применения R и поэтому не может входить в $R'(a)$. Так что $\Theta_0^-[1]$ входит в Θ_1, но тогда $R'(b)\Psi\Theta_1$ не может его содержать, поскольку мы стерли $\Theta_0^-[1]$ сразу после появления в цепочке слова Θ_0^-. Противоречие. □

Из этого можно заключить, что худшая стратегия вычислений в обогащенной грамматике – это однократное применение всех неукорачивающих правил, затем укорачивание, пока одно из этих правил не отменится, повторное его применение и т.д. Так что всякий отрезок наибольшей плохой последовательности, сохраняющий суффикс Θ_0, должен завершаться словом Θ_0. Используя эти соображения, найдем длину наибольшей плохой последовательности по индукции.

Теорема 3. Если неукорачивающие правила применяются к начальному слову длины 1 с приоритетом $R^{[0]}, R^{[1]}, \ldots, R^{[N]}$ (из правил, применения которых не обрывают плохую последовательность, всегда применяется правило с наименьшим индексом), то наихудшая стратегия породит плохую последовательность длины

$$C'_{Max} = 1 + |R^{[0]}| * (1 + |R^{[1]}| * (\cdots * (1 + |R^{[N]}|) \ldots))$$

Доказательство. Найдем искомую величину по индукции.

Пусть в момент появления в цепочке слова Θ_0 осталось лишь одно неукорачивающее правило R, применение которого не породит вложения по Турчину. Тогда длина отрезка, начинающегося с Θ_0 и заканчивающегося Θ_0^-, будет $|R|+1$ (одно применение R и $|R|$ укорачиваний).

Пусть мы знаем, как искать длину самой большой плохой последовательности в случае N возможных неукорачивающих правил для применения. Пусть их осталось $N+1$. Результат применения самого приоритетного R к Θ_0 есть $R(\Theta_0[1])\Theta_0^-$. Построим, начиная с этого слова, самую длинную плохую последовательность, сохраняющую суффикс $R(\Theta_0[1])^-\Theta_0^-$ (по предположению, мы

знаем длину этой последовательности; обозначим ее $C(N)$). Она завершается словом $R(\Theta_0[1])^-\Theta_0^-$. Теперь построим, начиная с этого слова, отрезок плохой последовательности, сохраняющий $R(\Theta_0[1])^{--}\Theta_0^-$ и так далее, пока не достигнем слова Θ_0^-. Всякая буква в правой части R порождает отрезок плохой последовательности длины $C(N)$, плюс еще одно слово – применение самой R. Таким образом, общий прирост длины плохой последовательности составляет $1 + |R| * C(N)$. □

Если все правила имеют одинаковую длину, то получается в точности формула теоремы 1.

Заметим, что аргументы суммы $1 + |R^{[0]}| * (1 + |R^{[1]}| * (\cdots * (1 + |R^{[N]}|) \ldots))$ пробегают лишь множество правых частей правил, поэтому при переходе от префиксной грамматики общего вида к обогащенной не возникает экспоненциального роста длины плохих последовательностей. [5]

C'_{Max} достигает наибольшего значения, если правила в очереди на применение $R^{[0]}, R^{[1]}, \ldots, R^{[N]}$ рассортированы по неубыванию длины. Теперь можно описать детерминированный алгоритм построения самой длинной плохой последовательности в обогащенной грамматике.

1. Пока есть неукорачивающие правила, которые не применялись или были отменены, применять самое длинное из них.
2. Укорачивать до первой отмены. Чаще всего это будет отмена последнего примененного неукорачивающего правила, однако в некоторых случаях стиранием одной буквы будут отменяться сразу несколько правил.

Пример 6. Оценим длину самой длинной плохой последовательности в грамматике $\mathbf{G'_F}$ (пример 5). Длина начального слова есть 2, кроме того, имеются два неукорачивающих правила – $R^{[1]}$ и $R^{[2]}$ – также длины 2, и одно длины 1. Теперь легко оценить длину наибольшей плохой последовательности по формуле теоремы 3: $2 * (1 + 2 * (1 + 2 * (1 + 1))) = 22$.

Теперь построим эту плохую последовательность явно. Для удобства чтения мы переименовали различные пары в различные буквы (так что $\langle f, 1\rangle = a$, $\langle a, 3\rangle = b$, $\langle f, 2\rangle = c$ и т.д.).

$\mathbf{G'_F}$:

$\Gamma_0 = ab$	$R^{[2]} : x \to ef$
$R^{[1]} : x \to cd$	$R^{[3]} : x \to g$
$R^{[4]} : x \to \Lambda$	

[5] То, что полученная оценка сама по себе экспоненциальна, нас полностью устраивает: одновременно она достаточно плоха, чтобы соответствующие ветви вычислений в семантическом дереве не обрывались слишком рано, и достаточно мала, чтобы программа онлайн-анализа работала разумное время.

Γ_0 : $\mathbf{a}_{(1)}b_{(0)}$
Γ_1 : $\mathbf{c}_{(3)}d_{(2)}b_{(0)}$
Γ_2 : $\mathbf{e}_{(5)}f_{(4)}d_{(2)}b_{(0)}$
Γ_3 : $\mathbf{g}_{(6)}f_{(4)}d_{(2)}b_{(0)}$
Γ_4 : $\mathbf{f}_{(4)}d_{(2)}b_{(0)}$
Γ_5 : $\mathbf{g}_{(7)}d_{(2)}b_{(0)}$
Γ_6 : $\mathbf{d}_{(2)}b_{(0)}$
Γ_7 : $\mathbf{e}_{(9)}f_{(8)}b_{(0)}$
Γ_8 : $\mathbf{g}_{(10)}f_{(8)}b_{(0)}$
Γ_9 : $\mathbf{f}_{(8)}b_{(0)}$
Γ_{10} : $\mathbf{g}_{(11)}b_{(0)}$
Γ_{11} : $\mathbf{b}_{(0)}$
Γ_{12} : $\mathbf{c}_{(13)}d_{(12)}$
Γ_{13} : $\mathbf{e}_{15}f_{(14)}d_{(12)}$
Γ_{14} : $\mathbf{g}_{16}f_{(14)}d_{(12)}$
Γ_{15} : $\mathbf{f}_{(14)}d_{(12)}$
Γ_{16} : $\mathbf{g}_{(17)}d_{(12)}$
Γ_{17} : $\mathbf{d}_{(12)}$
Γ_{18} : $\mathbf{e}_{(19)}f_{(18)}$
Γ_{19} : $\mathbf{g}_{(20)}f_{(18)}$
Γ_{20} : $\mathbf{f}_{(18)}$
Γ_{21} : $\mathbf{g}_{(21)}$

Какое бы из правил $R^{[1]}$–$R^{[4]}$ мы ни применили к Γ_{21}, возникнет вложение по Турчину.

Если в алфавите Υ есть хотя бы две буквы, оценка теоремы 1 оказывается точнее, чем оценка из раздела 4 (извлеченная из доказательства теоремы Турчина). Действительно, пусть все неукорачивающие правила имеют длину k и их количество равно N. Оценка из теоремы Турчина C_{Max} может быть переписана как

$$C_{Max} = |\Upsilon|^{|\Gamma_0|+\sum(|R_r^{[i]}|-1)} = card(\Upsilon)^{|\Gamma_0|} * (card(\Upsilon)^{k-1})^N.$$

Если $card(\Upsilon) \geqslant 2$, то $|\Gamma_0| < card(\Upsilon)^{|\Gamma_0|}$ и $k \leqslant card(\Upsilon)^{k-1}$.

§ 6. Дальнейшие свойства отношения Турчина

Если при онлайн-анализе программ мы хотим отслеживать не только состояние стека, но и состояние статических данных (скажем, общую структуру вычисляемых термов), нетранзитивность отношения Турчина порождает некоторые вопросы. А именно, сохранится ли свойство почти полноты, если рассматривать прямое произведение отношения Турчина с некоторым полным квазипорядком? Для доказательства этого правдоподобного факта кажется разумным применить теорему Рамсея [14], однако вложение по Турчину не является почти полным всегда – мы знаем лишь, что оно почти полно на цепочках, порожденных префиксными грамматиками.

Пример 7. Пусть имеется произвольное вычисление. Раскрасим пары $\langle \Phi, \Psi \rangle$ такие, что Φ предшествует Ψ, в два цвета: те, в которых $\Psi \preceq \Phi$ – в зеленый, а все остальные – в красный. Существует бесконечная цепочка, являющаяся подпоследовательностью исходной, такая, что все пары, которые она содержит, окрашены в один и тот же цвет [14]. Если это красный, то, казалось бы, получается противоречие с теоремой Турчина, однако рассмотрим последовательность, построенную следующим образом. На i-м шаге будем стирать все буквы и записывать слово, состоящее из i букв a, за которыми следует единственная буква b. Никакие два слова в этой последовательности не образуют турчинской пары, однако если заменить полное стирание побуквенным

и одношаговую запись очередного слова разбить на элементарные действия, получится цепочка, порожденная префиксной грамматикой и содержащая бесконечную красную подпоследовательность (буква c здесь выступает маркером конца слова, она добавлена, чтобы ни на одном шаге вычисления не возникло слова Λ).

$\Gamma_0:$	$a_{(2)}b_{(1)}c_{(0)}$	$\Gamma_7:$	$b_{(5)}c_{(3)}$
$\Gamma_1:$	$b_{(1)}c_{(0)}$	$\Gamma_8:$	$c_{(3)}$
$\Gamma_2:$	$c_{(0)}$	$\Gamma_9:$	$b_{(10)}c_{(9)}$
$\Gamma_3:$	$b_{(4)}c_{(3)}$	$\Gamma_{10}:$	$a_{(12)}b_{(11)}c_{(9)}$
$\Gamma_4:$	$a_{(6)}b_{(5)}c_{(3)}$	$\Gamma_{11}:$	$a_{(14)}a_{(13)}b_{(11)}c_{(9)}$
$\Gamma_5:$	$a_{(8)}a_{(7)}b_{(5)}c_{(3)}$	$\Gamma_{12}:$	$a_{(16)}a_{(15)}a_{(13)}b_{(11)}c_{(9)}$
$\Gamma_6:$	$a_{(7)}b_{(5)}c_{(3)}$	$\ldots$	$\ldots$

Все пары бесконечной последовательности $\Gamma_0, \Gamma_5, \Gamma_{12}, \Gamma_{21}, \ldots$ окрашены только в красный. Хотя имеется и ряд бесконечных последовательностей, все пары из которых окрашены только в зеленый, их существование в общем случае не получается доказать без обращения к свойствам грамматики.

Теорема 4. Отношение $\preceq$ содержит квазипорядок, полный на всех цепочках, порожденных алфавитными префиксными грамматиками.

Доказательство. Пусть дана алфавитная префиксная грамматика $\langle \Upsilon, \mathbf{R}, \Gamma_0 \rangle$ Рассмотрим все бесконечные цепочки $\{\Phi_i\}_{i=1}^{\infty}$, порожденные этой грамматикой, такие, что $\exists N \forall i \exists j (i < j \ \& \ |\Phi_j| \leqslant N)$ – то есть сколь угодно далеко в цепочке встречаются слова, длина которых не превышает некоторого N.

Для каждой такой цепочки выберем наименьшее N, удовлетворяющее этому свойству. В силу конечности множества правил переписывания, слова, порожденные ими, будут состоять лишь из конечного множества букв, и поэтому какое-то слово длины N будет бесконечно повторяться (с точностью до временных индексов). Его первая буква всегда будет порождаться одним и тем же неукорачивающим правилом. Каждые два из результатов этих применений данного неукорачивающего правила будут образовывать турчинскую пару.

Все прочие бесконечные цепочки $\{\Phi_i\}_{i=1}^{\infty}$ обладают тем свойством, что длина слова в них постепенно растет: $\forall N \exists i_N \forall j (j > i_N \Rightarrow |\Phi_j| > N)$. Для каждого N в каждой цепочке такого рода выберем первое i_N такое, что длина слов в цепочке после элемента Φ_{i_N} никогда не уменьшается менее, чем до N: $\forall j (j < i_N \Rightarrow \exists k (k \geqslant j \ \& \ |\Phi_k| < N))$. Итак, $|\Phi_{i_N - 1}| < N$ и $|\Phi_{i_N}| \geqslant N$, и Φ_{i_N} порождено из предыдущего слова применением неукорачивающего правила R, $|R| \geqslant 2$: $\Phi_{i_N} = R(\Phi_{i_N - 1}[1])\Phi^{-}_{i_N - 1}$. $\Phi^{-}_{i_N - 1}$ неизменно, поскольку $|\Phi_{i_N - 1}| < N$. Все элементы последовательности $\{\Phi_{i_N}\}_{N=1}^{\infty}$ начинаются с правой части некоторого неукорачивающего правила, значит, существует $\{\Phi_{i_K}\}_{K=1}^{\infty}$ – бесконечная подпоследовательность $\{\Phi_{i_N}\}_{N=1}^{\infty}$, такая, что все ее элементы начинаются с правой части одного и того же правила. Поскольку все, кроме первой, буквы слов из $\{\Phi_{i_N}\}_{i=N}^{\infty}$, остаются неизменны, то каждые два элемента выделенной нами подпоследовательности $\{\Phi_{i_K}\}_{K=1}^{\infty}$, расположенные в порядке, соответствующем их порядку в исходной цепочке, образуют турчинскую пару.

Множество T всех пар, удовлетворяющих этим условиям, образует подмножество отношения Турчина, являющееся полным квазипорядком. □

Множество будем T называть далее ***усиленным отношением Турчина***. Казалось бы, оно представляет собой как раз то самое отношение «зацикливания», которое мы и искали, содержащее слова вида $[\texttt{Top}][\texttt{Middle}]^n[\texttt{Context}]$ (где $[\texttt{Middle}]^n - n$ раз подряд записанное слово $[\texttt{Middle}]$). Так и происходит, если префиксная грамматика детерминированна (для каждой буквы из Υ имеется лишь одно правило, содержащее эту букву в левой части). Тогда в любой бесконечной цепочке, начиная с некоторого номера, каждое слово представляется как $\Xi\Delta\Theta$, причем существуют слова $\Xi\Delta^n\Theta$ для всех $n > 1$.

Действительно, пусть $\Gamma\Theta_0$ принадлежит отрезку цепочки, заключенному между членами турчинской пары $\Phi\Theta_0$ и $\Phi\Psi\Theta_0$. Выделим в Γ часть, которая никогда больше не будет меняться. Обозначим ее Γ_0, а оставшийся префикс – Ξ. В силу детерминированности грамматики Φ всегда развивается в $\Xi\Gamma_0$, а Ξ – в $\Phi\Psi_0$, где $\Psi = \Psi_0\Gamma_0$. Значит, Ξ развивается в $\Xi\Gamma_0\Psi_0$, и можно взять $\Delta = \Gamma_0\Psi_0$, чтобы получить искомое представление $\Gamma\Theta_0 = \Xi\Delta\Theta$. Слово $\Gamma_0\Psi_0$ может быть и пустым, тогда $\Gamma\Theta_0$ будет бесконечно повторять само себя.

Однако в общем случае ситуация несколько сложнее.

Пример 8. Рассмотрим простейшую грамматику с алфавитом $\{a, b, c\}$ и набором из двух правил:

$R^{[1]} : c \to ca$

$R^{[2]} : c \to cb$

Пусть развитие (бесконечной) цепочки происходит так:

ca

cba

$caba$

$caaba$

$cbaaba$

...

$cbaaabaaba$

...

$cbaaaabaaabaaba$

...

При таком развитии не существует ни одного слова такого, что оно представляется в виде $\Xi\Delta\Theta$ так, что в цепочке встречаются слова $\Xi\Delta^n\Theta$ для всех $n > 1$.

Заметим, что если $\langle\Phi, \Psi\rangle \in T$, то Φ и Ψ оба порождены непосредственным применением некоторого неукорачивающего правила. Так что условие Турчина можно не проверять после укорачиваний – на оставшихся парах оно все равно останется почти полным. Кроме того, существование T влечет два важных практических следствия. Первое из них касается возможности комбинировать отношение Турчина с другими критериями определения семантических циклов, если только они обладают свойствами полного квазипорядка.

Лемма 3. Пусть R – квазипорядок, полный на произвольных последовательностях. $R \cap \preceq$ – почти полное отношение (также почти полно $R \cap T$).

Может возникнуть вопрос, зачем рассматривать пересечение отношения Турчина и, например, отношения Хигмана, если первое является явным сужением второго. Но следует напомнить, что отношение Турчина рассматривает стек, тогда как отношение Хигмана-Крускала – древесную структуру терма. Пусть f – функция, а A, B – произвольные статические данные. Рассмотрим термы $f(A(x))$ и $f(B(x))$. Соответствующие конфигурации стека есть $f_{(k_1)}$ и $f_{(k_2)}$, так что они образуют турчинскую пару, но, очевидно, $f(A(x)) \not\trianglelefteq f(B(x))$. В то же время даже отношение Хигмана в некоторых случаях недостаточно учитывает контекст вычисления и строит слишком поспешную аппроксимацию семантического цикла там, где отношение Турчина не выполняется; яркий пример подобного случая приведен в разделе 7.

Оговорка про полноту квазипорядка R на произвольных последовательностях в лемме 3 не может быть опущена: прямое произведение отношений Турчина на цепочках, порожденных префиксными грамматиками, может оказаться не почти полным.

Определение 8. Скажем, что слово Γ вкладывается в слово Δ не более, чем с одним разрывом, если существуют такие слова Φ, Ψ, Θ (возможно, пустые), что $\Gamma = \Phi\Theta$, $\Delta = \Phi\Psi\Theta$.

Скажем, что слово Γ вкладывается в слово Δ ***не более, чем с*** $n+1$ ***разрывом***, если существуют такие слова Φ, Ψ, Θ_1, Θ_2 (возможно, пустые), что $\Gamma = \Phi\Theta_1$, $\Delta = \Phi\Psi\Theta_2$ и слово Θ_1 вкладывается в слово Θ_2 не более, чем с n разрывами.

Например, слово *abac* вкладывается в *abrac* не более, чем с одним разрывом, но в слово *abracadabra* – не более, чем с двумя разрывами (*abac* разделяется на две части: *ab* и *ac*, однако конец слова *abracadabra* не следует непосредственно за *ac*, и этот разрыв между *ac* и концом слова тоже учитывается).

Лемма 4. Если отношение R вложения слов допускает ограниченное число разрывов, оно не является почти полным на цепочках, порожденных $G_1 \times G_2$, где G_1 и G_2 – префиксные грамматики, даже в детерминированном случае.

Доказательство. Рассмотрим класс грамматик $\mathbf{G}^{[n]}$ на парах слов с алфавитом $\{a_1, \ldots, a_n, A_1, \ldots, A_n, e, E\} \times \{a_1, \ldots a_n, A_1, \ldots, A_n, e, E\}$, начальным словом $\langle e, A_1 A_2 \ldots A_n\rangle$ и правилами переписывания:

$R^{[00]} : \langle e, a_1\rangle \to \langle E, a_1 a_1\rangle$
$R^{[01]} : \langle E, A_1\rangle \to \langle e, A_1 A_1\rangle$
$R^{[02]} : \langle e, A_1\rangle \to \langle A_1 e, \Lambda\rangle$
$R^{[03]} : \langle E, A_1\rangle \to \langle E, \Lambda\rangle$
...
$R^{[i0]} : \langle a_i, a_i\rangle \to \langle \Lambda, a_i a_i\rangle$
$R^{[i1]} : \langle A_i, A_i\rangle \to \langle \Lambda, A_i A_i\rangle$
$R^{[i2]} : \langle a_i, A_i\rangle \to \langle a_i a_i, \Lambda\rangle$
$R^{[i3]} : \langle A_i, a_i\rangle \to \langle A_i A_i, \Lambda\rangle$
$R^{[i4]} : \langle a_i, a_{i+1}\rangle \to \langle \Lambda, a_i a_{i+1} a_{i+1}\rangle$
$R^{[i5]} : \langle A_i, A_{i+1}\rangle \to \langle \Lambda, A_i A_{i+1} A_{i+1}\rangle$

$R^{[i6]} : \langle a_i, A_{i+1}\rangle \to \langle a_{i+1}a_i a_i, \Lambda\rangle$
$R^{[i7]} : \langle A_i, a_{i+1}\rangle \to \langle A_{i+1}A_i A_i, \Lambda\rangle$
...
$R^{[n0]} : \langle a_n, e\rangle \to \langle \Lambda, a_n e\rangle$
$R^{[n1]} : \langle A_n, E\rangle \to \langle \Lambda, A_n E\rangle$
$R^{[n2]} : \langle a_n, E\rangle \to \langle a_n a_n, e\rangle$
$R^{[n3]} : \langle A_n, e\rangle \to \langle A_n A_n, E\rangle$

Каково бы ни было N, начиная с некоторого n, найдется грамматика из класса $\mathbf{G}^{[n]}$ (с достаточно большим, но конечным количеством правил), которая будет порождать бесконечную цепочку, не содержащую ни одной пары $\langle\Phi_1, \Psi_1\rangle$, $\langle\Phi_2, \Psi_2\rangle$ такой, что Φ_1 вкладывается в Φ_2, а Ψ_1 вкладывается в Ψ_2 с, самое большее, N разрывами. □

Второе важное свойство отношения Турчина, следующее из почти полноты усиленного отношения T – это возможность его обобщения на более широкий класс грамматик.

Лемма 5. Отношение Турчина является почти полным на всех вычислениях, порожденных произвольными префиксными грамматиками с конечным числом правил (не обязательно переписывающими лишь первую букву слова).

Доказательство. Рассмотрим правило $a_1 a_2 \dots a_n \to b_1 b_2 \dots b_m$ неалфавитной префиксной грамматики. Его действие эквивалентно композиции действий:
$a_1 \to \Lambda$
...
$a_{n-1} \to \Lambda$
$a_n \to b_1 b_2 \dots b_m$.
Отношение T почти полно на последовательностях, порожденных алфавитными префиксными грамматиками. Значит, $\preceq$ (а также и T) почти полно на последовательностях, порожденных произвольными префиксными грамматиками с конечным набором правил переписывания. □

Свойство почти полноты усиленного отношения Турчина T оказалось, в частности, полезно в задаче верификации пинг-понг протоколов связи [9]. Оно не играет существенной роли при суперкомпиляции программ, написанных на базисном Рефале, поскольку в нем запрещены вызовы функций в левых частях определений.

§ 7. Отношение Турчина и отношение Хигмана

Теорема Турчина не только гарантирует существование вложения по Турчину на всяком бесконечном пути, но и утверждает, что эти вложения можно найти за экспоненциальное время. В случае хигмановых пар, точная верхняя граница их существования оценивается многократно рекурсивной функцией [11], даже если длина слова на i-м шаге вычисления ограничена величиной $|\Gamma_0| + i * k$ (k – константа) ([11], [15]). Этот сложностной разрыв возникает из-за

той разницы, что вложение по Турчину рассматривается лишь в ограниченном классе цепочек, а отношение Хигмана распространяется на произвольные последовательности, даже если они не порождаются никаким алгоритмом. Мы уже видели, что в этом случае о почти полноте отношения Турчина говорить нельзя (см. пример 7).

Тем не менее, длина плохой последовательности в смысле Хигмана в обогащенных префиксных грамматиках также оказывается в худшем случае экспоненциальной. С одной стороны, она не может превышать длины наибольшей плохой последовательности в смысле Турчина; с другой, алгоритм построения наихудшей плохой последовательности по Турчину в обогащенной грамматике порождает также и плохую последовательность по Хигману. Более того, не только в худшем случае, но и вообще всегда в обогащенных грамматиках длины плохих последовательностей относительно этих двух отношений совпадают.

Теорема 5. Хигманова пара, обрывающая плохую последовательность в обогащенной грамматике, является турчинской парой.

Доказательство. Предположим, что имеются такие Φ_1 и Φ_2, что $\Phi_1 \trianglelefteq \Phi_2$, причем Φ_1 входит в Φ_2 с $n+1$ разрывом (с точностью до временных индексов). Выделим в них общий суффикс Θ_0 (возможно, пустой). Тогда $\Phi_1 = A_1 A_2 \dots A_n \Theta_0$ и $\Phi_2 = B_1 A'_1 B_2 A'_2 \dots B_n A'_n B_{n+1} \Theta_0$, причем $A_i \approx A'_i$) для всех i от 1 до n. Рассмотрим результаты порождения букв $A_n[1]$ и $A'_n[1]$. По свойствам грамматики, они оба порождены одним и тем же правилом $R : x \to R_r$ и представляют собой слова $\Delta A_n \Theta_0$ и $\Delta' A'_n B_{n+1} \Theta_0$ причем $\Delta \approx \Delta'$, поскольку это префиксы правой части одного и того же правила до буквы $A_n[1]$. Значит, $\Delta A_n \Theta_0 \trianglelefteq \Delta' A'_n B_{n+1} \Theta_0$. Следовательно, $i = 1 = n$, поскольку Φ_1 и Φ_2 – первая хигманова пара в цепочке. Но если $i = 1 = n$, то $\Phi_1 = A_1 \Theta_0$ и $\Phi_2 = A'_1 B_{n+1} \Theta_0$, так что $\Phi_1 \preceq \Phi_2$. □

Эта простая теорема может иметь важное практическое следствие для тех систем онлайн-анализа программ, которые используют отношение Хигмана в качестве критерия обрыва ветви семантического дерева. Если добавить в грамматику (не обязательно даже префиксную), представляющую программу, простую разметку по контексту, это позволит получить преимущества отношения Турчина (такие, как учет особенностей развития ветви при поиске вложений) без потери независимости почти полноты отношения Хигмана от свойств грамматики, порождающей вычисление.

Покажем, как это совмещение идей может дать практический выигрыш, на простом примере.

Пример 9. Рассмотрим программу, вычисляющую наименьшую степень 2, большую, чем входное число (в первом столбце представлен вариант программы на Рефале, во втором – на Scala).

```
F { 'Z' = 'Z';                          f(Z)    = Z;
    'S' e.1 = 'S' <F <G 'S' e.1>>;      f(S(x)) = S(f(g(S(x))));
  }
G { 'Z' = 'Z';                          g(Z)    = Z;
    'S' e.1 = <H e.1>;                  g(S(x)) = h(x);
  }
```

```
H { 'Z' = 'Z';                          h(Z)    = Z;
    'S' e.1 = 'S' <G e.1>;              h(S(x)) = S(g(x));
  }
```

При использовании в онлайн-анализе этой программы чистого отношения Хигмана развертка $f(S(x))$ в $S(f(g(S(x))))$ сразу же влечет обрыв ветви, поскольку $f(S(x)) \trianglelefteq S(f(g(S(x))))$. Таким образом, даже $f(S(Z))$ не может быть досчитано до конца. Тогда как усиленное отношение Турчина T позволит развиться следующей цепочке:

$$f_{(0)} \to g_{(2)} f'_{(1)} \to h_{(3)} f'_{(1)} \to f'_{(1)} \to \Lambda$$

Можно заметить, что уже в случае начального вызова $f(S(S(Z)))$ слишком ранний обрыв ветви совершает уже и усиленное отношение Турчина, однако его прямое произведение с отношением Хигмана либо размеченное отношение Хигмана оказывается успешным.

Благодарности

Искренне благодарю редактора сборника Андрея Петровича Немытых за вдумчивые замечания к тексту, которые помогли улучшить его изложение.

Список литературы

[1] E. Albert, J. Gallagher, M. Gomez-Zamalla, G. Puebla, "Type-based Homeomorphic Embedding for Online Termination", *Journal of Information Processing Letters*, **109(15)**, 2009, 879–886.

[2] D. Caucal, "On the Regular Structure of Prefix Rewriting", *Theoretical Computer Science*, **106**, 1992, 61–86.

[3] M. C. Bolingbroke, S. L. Peyton Jones, D. Vytiniotis, "Termination combinators forever", *Proceedings of the 4th ACM SIGPLAN Symposium on Haskell*, 2011, 23–34.

[4] G. Higman, "Ordering by divisibility in abstract algebras", *Bulletin of London Mathematical Society*, **3(2)**, 1952, 326–336.

[5] I. Klyuchnikov, S. Romanenko, "Proving the equivalence of higher-order terms by means of supercompilation", *Perspectives of Systems Informatics*, LNCS, **5947**, 2010, 193–205.

[6] J. B. Kruskal, "Well-quasi ordering, the tree theorem, and Vazsonyi's conjecture", *Transactions of the American Mathematical Society*, **95**, 1960, 210–225.

[7] C. St. J. A. Nash-Williams, "On well-quasi-ordering infinite trees", *Proceedings of Cambridge Philosophical Society*, **61**, 1965, 697–720.

[8] А. П. Немытых, *Суперкомпилятор SCP4: общая структура*, ЛКИ, Москва, 2007.

[9] A. Nepeivoda, "Ping-Pong Protocols as Prefix Grammars and Turchin Relation", *VPT 2013. First International Workshop on Verification and Program Transformation*, EPiC Series, **16**, ред. A. Lisitsa and A. Nemytykh, EasyChair, 2013, 74–87.

[10] M. H. Sørensen, R. Glück, "An algorithm of generalization in positive supercompilation", *Logic Programming: Proceedings of the 1995 International Symposium*, 1995, 465–479.

[11] H. Touzet, "A Characterisation of Multiply Recursive Functions with Higman's Lemma", *Information and Computation*, **178**, 2002, 534–544.

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

[13] V. F. Turchin, "The algorithm of generalization in the supercompiler", *Partial Evaluation and Mixed Computation*, 1988, 341–353.

[14] W. Veldman, M. Bezem, "Ramsey's theorem and the pigeonhole principle in intuitionistic mathematics", *Journal of London Mathematical Society*, **47(2)**, 1993, 193–211.

[15] A. Weiermann, "Phase transition thresholds for some Friedman-style independence results", *Mathematical Logic Quarterly*, **53(1)**, 2007, 4–18.

А. Н. Непейвода (A. N. Nepeivoda)
Переславль-Залесский

Научное издание

Сборник трудов по программированию

Сборник трудов по функциональному языку программирования Рефал, том I, 2014 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. **8.83**.
Усл. печ. л. **15.64**. Подписано к печати **28.01.2014**.
Ответственная за выпуск: *Н. А. Федотова*.

СБОРНИК
Издание трудов научных конференций

Отпечатано в **ООО "Регион"**.
Печать **цифровая**. Бумага **мелованная**. Тираж **100 экз**. Заказ 1.
152025 Ярославская область г. Переславль-Залесский ул. Строителей, д. 41