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.

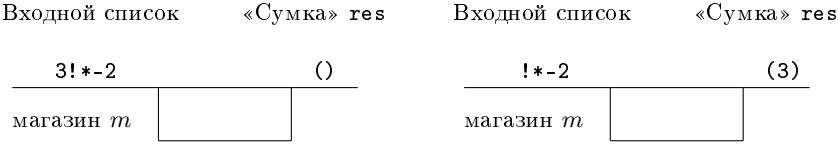
Если очередной терм – знак операции, то определяется класс этой операции. Из описания знака операции извлекается соответствующий аппликативный терм. Операция определена. Остается расставить скобки. Эта операция конкурирует с операцией из магазина. Если побеждает операция из магазина, то на нее (вместе 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

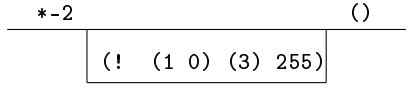
```
3!*-2
```

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:

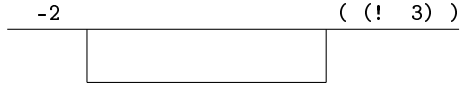


Следующий терм `!`. Это знак операции класса III. Магазин пуст. Операция `!` помещается в магазин.



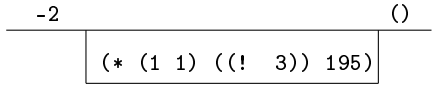
Заметим, что «сумка» `(3)` переместилась в магазин вместе с операцией `!`. Конечный автомат переходит в состояние `At`.

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.



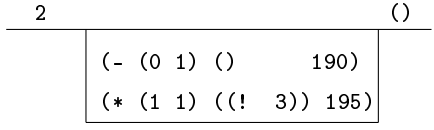
[5] В данном случае разделение элементов списка пробелами, согласно синтаксису FLACa, не обязательно.

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

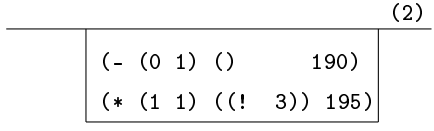


Конечный автомат переходит в состояние Bg.

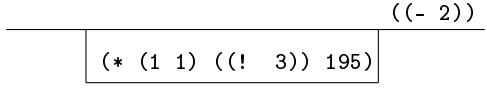
Очередной терм -. Это унарная операция. (см. таблицу определения класса). Ее левый приоритет 210. Правый приоритет операции * из магазина 195. Побеждает минус. Следовательно, минус помещается в магазин.



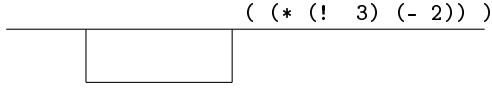
Очередной терм 2. Это не знак операции.



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)).

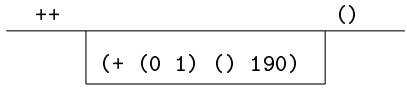
Пример 2. Пусть входной список

+++

В этом списке три терма[6].

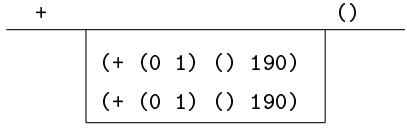
Первый терм +. Это знак операции. Это операция класса I. Помещаем ее в магазин.

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

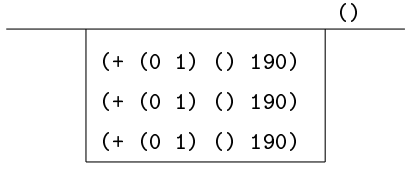


Конечный автомат остается в состоянии `Bg`.

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

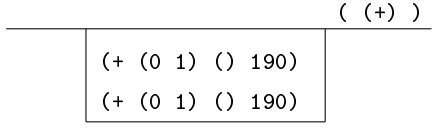


Очередной терм `+`. Аналогично предыдущему имеем:

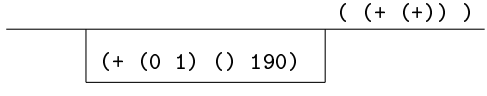


Теперь освобождается магазин. Унарная операция `+` имеет один правый аргумент. Правые аргументы должны находиться в «сумке» `res`.

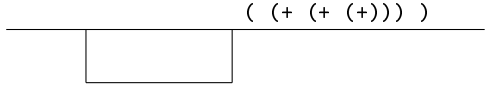
Но «сумка» пуста. Следовательно, имеем



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