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

Содержание 43
1. Введение 44
2. Выбранное представление данных эффективно использует память, позволяет быстро копировать, но ограничивает возможности перестановок 45
3. Алгоритм отождествления 48
4. FLAC-машина 51
5. Представление атомов 52
6. Язык FAL 53
   6.1. Операторы отождествления 53
   6.2. Операторы построения 60
   6.3. Оператор копирования 63
   6.4. Стек активных функций. Программа и данные 64
   6.5. Перемещение кусков из существующих графов в строящиеся требует дополнительных действий при отождествлении 66
   6.6. Оператор замены 70
   6.7. Другие операторы 71
7. Программы на языке FAL 73
8. Встроенные функции 74
9. Загрузка. Механизм модульности. 76
10. Интерпретатор 79
11. Оператор языка FAL – встроенная функция `QUOTE` 80
12. Арифметика 82
13. Отражение состояния FLAC-машины в памяти компьютера 84
   13.1. Представление вершины графа 84
   13.2. «Внутренности» атомов 85
   13.3. Встроенные функции и операторы языка FAL 85
   13.4. Выбор основания системы счисления 85
14. Счетчик ссылок позволяет избавиться от глобальной сборки «мусора». Сборка «мусора» в области атомов. 85
15. Реакция системы на ошибки. Встроенные функции `RUNEND`, `SYNTAX`. 86
16. Таблица кодов языка FAL. Статистические тесты. 87
Список литературы 91

§ 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.

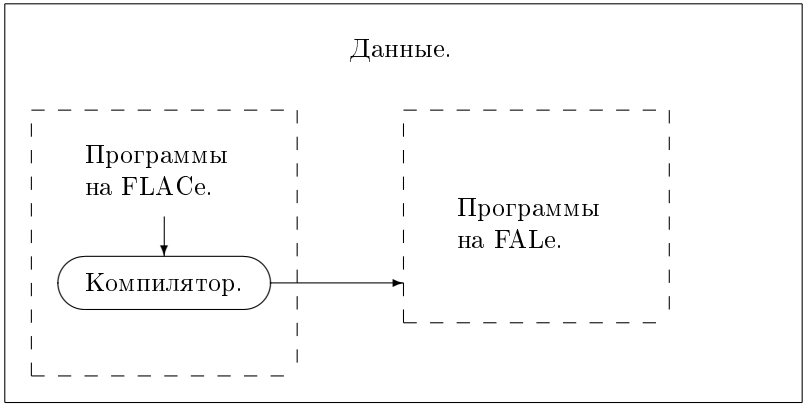


Рис. 1. Язык программирования FLAC.

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.).

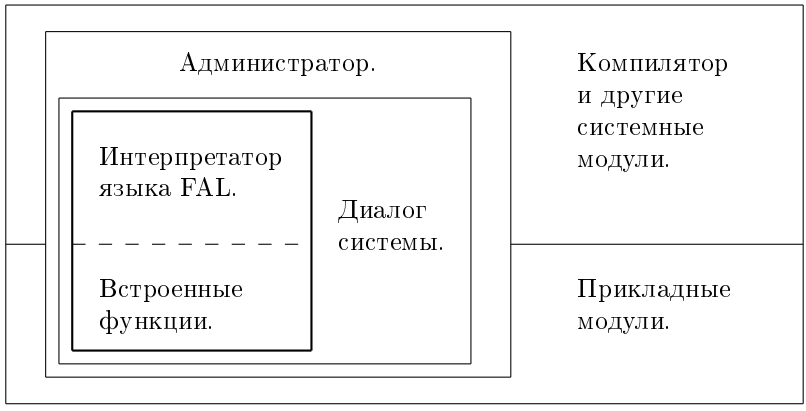


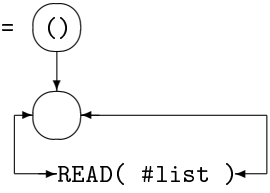
Рис. 2. Система компьютерной алгебры FLAC. (Обведённое жирной чертой реализовано на языке «C», оставшаяся часть – на FLACe.)

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)`

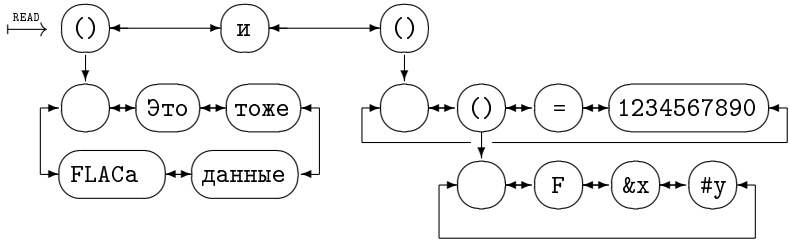


Рис. 3

Наиболее интересным является третий пример:

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

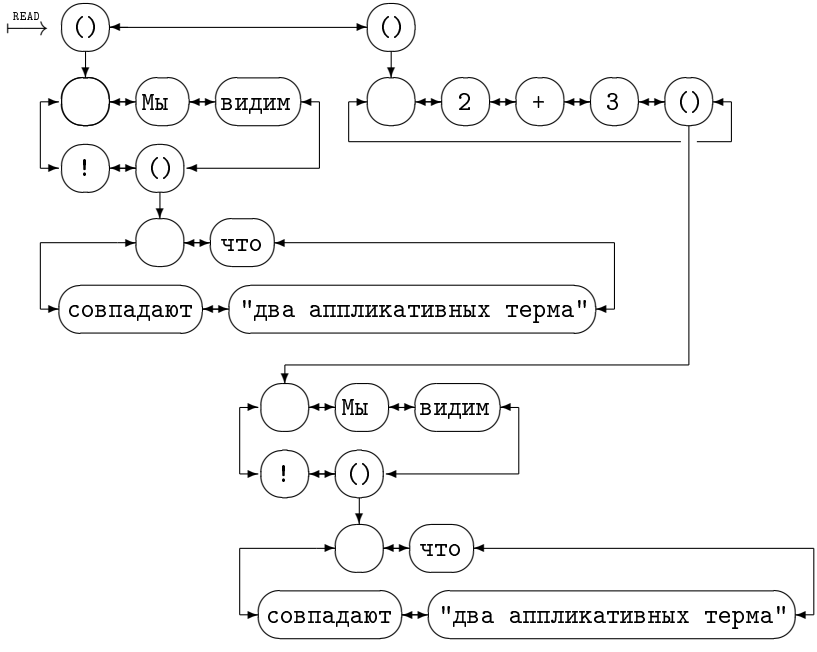


Рис. 4

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

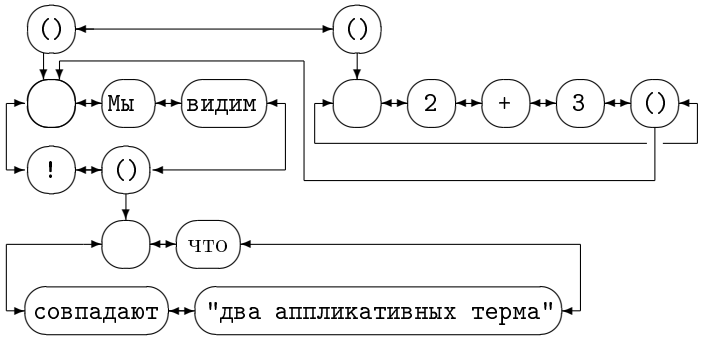


Рис. 5

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
```

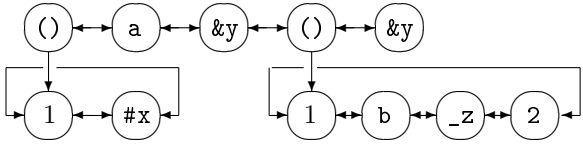


Рис. 6

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` :

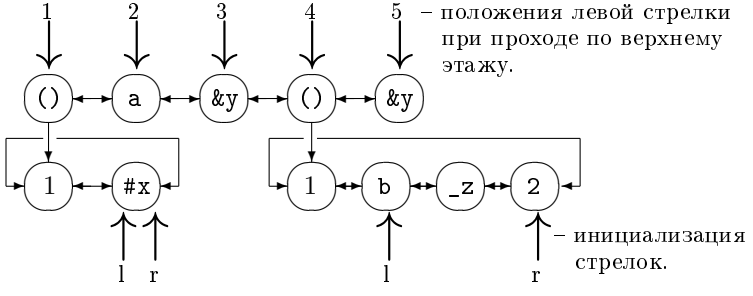


Рис. 7

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].

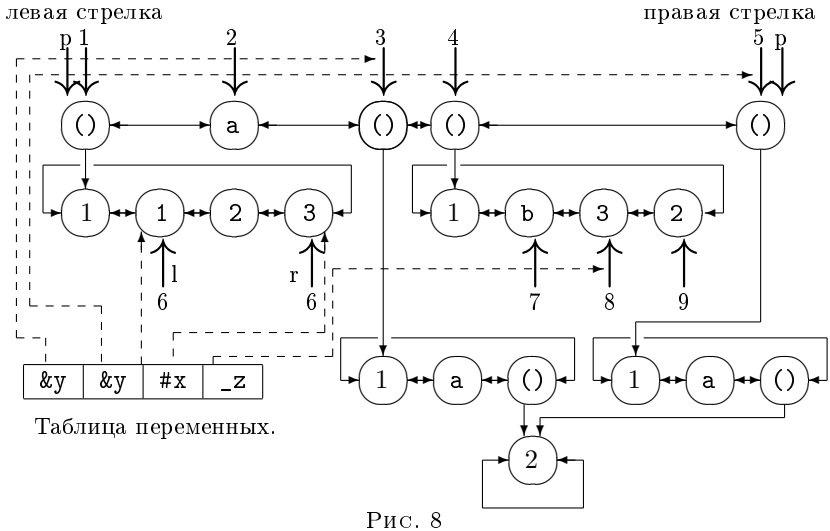

левая стрелка
правая стрелка
p 1
2
3
4
5 p
()
a
()
()
()
1
1
2
3
1
b
3
2
l
6
r
6
7
8
9
&y
&y
#x
_z
1
a
()
1
a
()
2
Таблица переменных.

Рис. 8

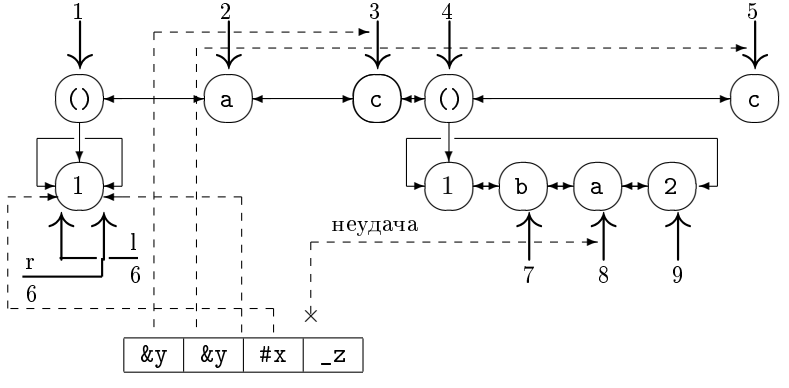

1
2
3
4
5
()
a
c
()
c
1
1
b
a
2
неудача
r
6
l
6
7
8
9
&y
&y
#x
_z
Таблица переменных.

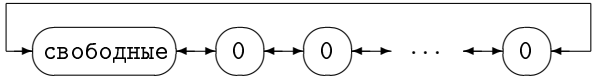
Рис. 9

§ 4. FLAC-машина

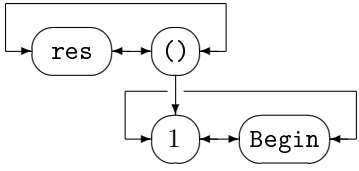
Опишем абстрактную FLAC-машину – машину переписывания рассматриваемых нами графов.

В выключенном состоянии она представляет собой несколько графов:

а) список свободных вершин –

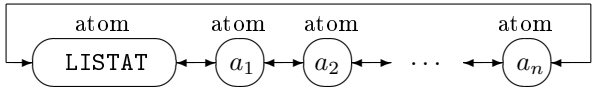


б) граф вызовов функций и хранения всех результатов выглядит так –



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.

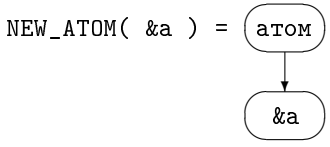
4) Корректировка стека активных функций; вызовы функций, стоящие в правой части предложения, вычисляются слева направо, и к моменту вычисления функции ее аргументы должны быть определены.

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

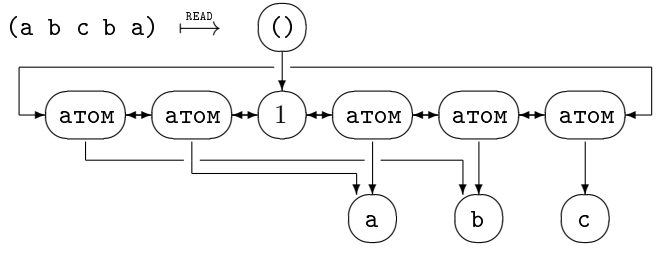
Для того чтобы рассказать о таблице атомов, активизируем функцию `NEW_ATOM` [см. п.2].



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?

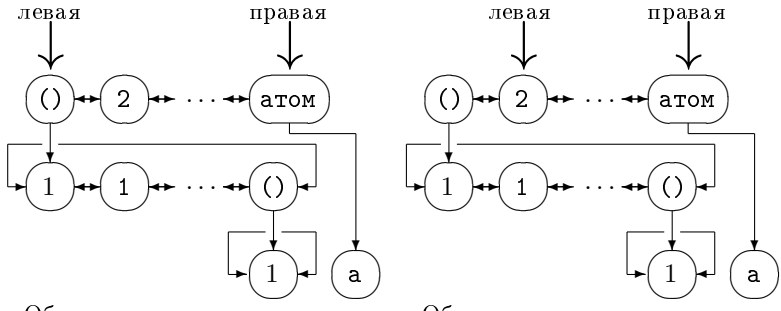
Равны ли графы, начинающиеся с вершины-аргумента и с вершины, на которую указывает левая стрелка?

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 ()) )`.

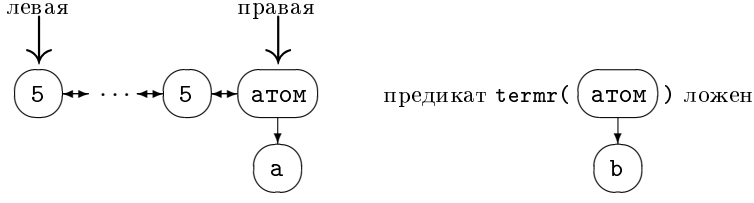


Рис. 10

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.

```
}
```

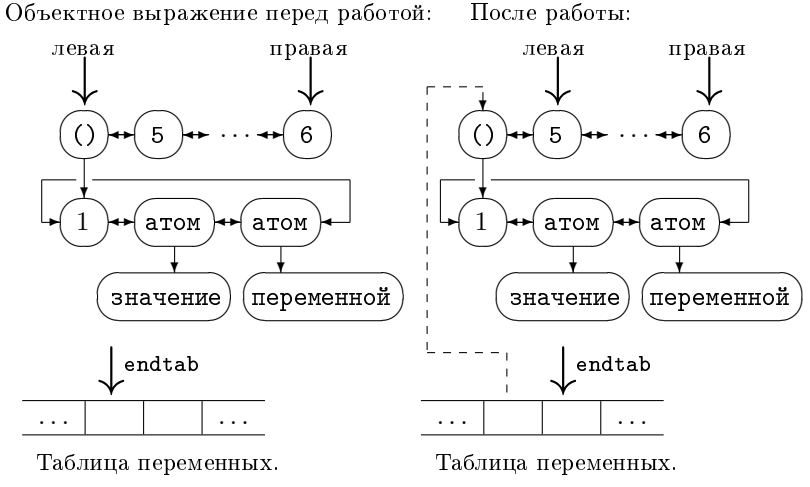


Рис. 11. Работа `vterml()`.

Предикат: число?

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

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

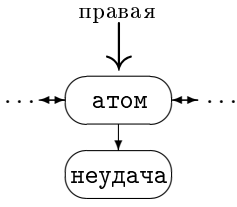
Левая стрелка стоит на числе?

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.

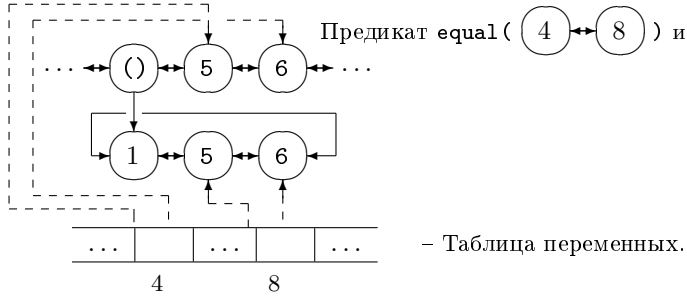


Рис. 13

Оператор `list()` фиксирует переменную типа список.

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

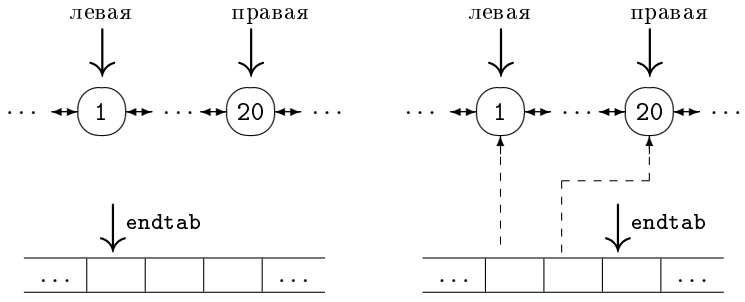
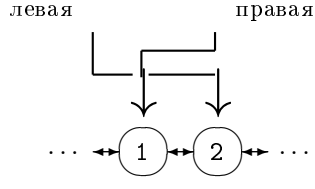


Рис. 14. Работа `list()`.

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.

```
}
```

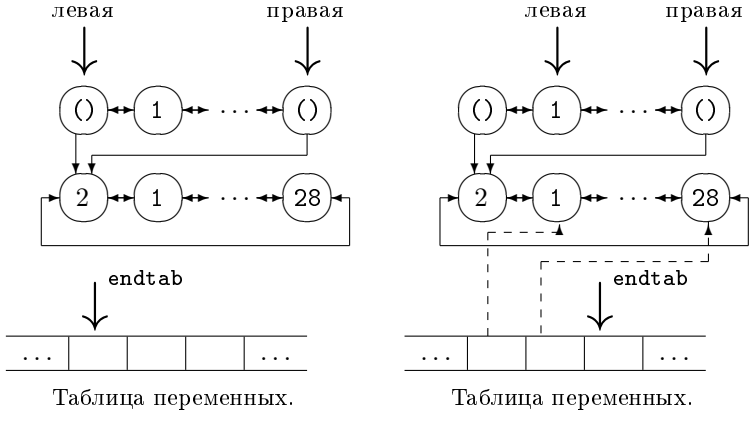


Рис. 16. Предикат `brackl()`.

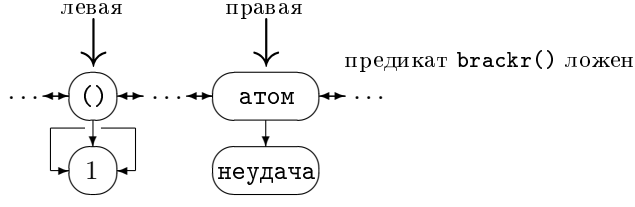


Рис. 17

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.

}

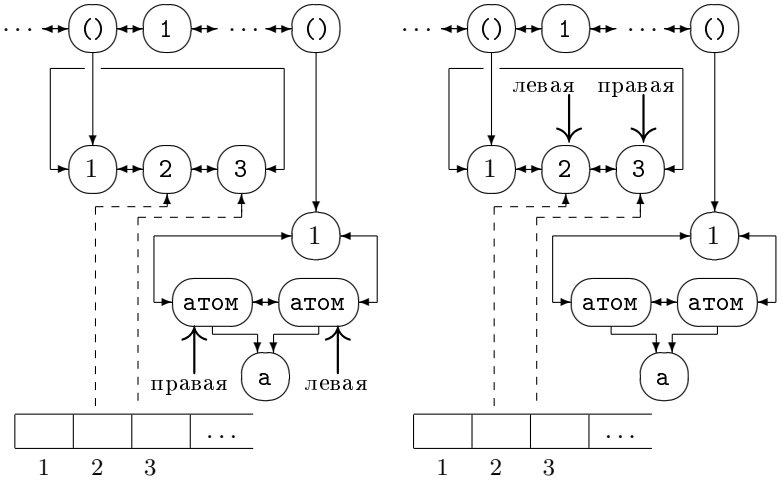


Рис. 18

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.)
}

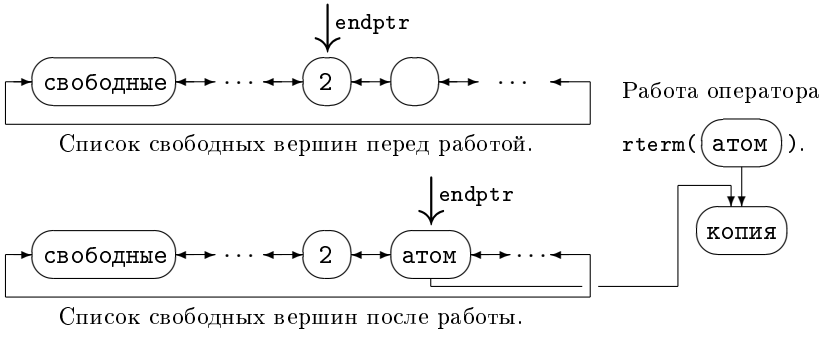


Рис. 19

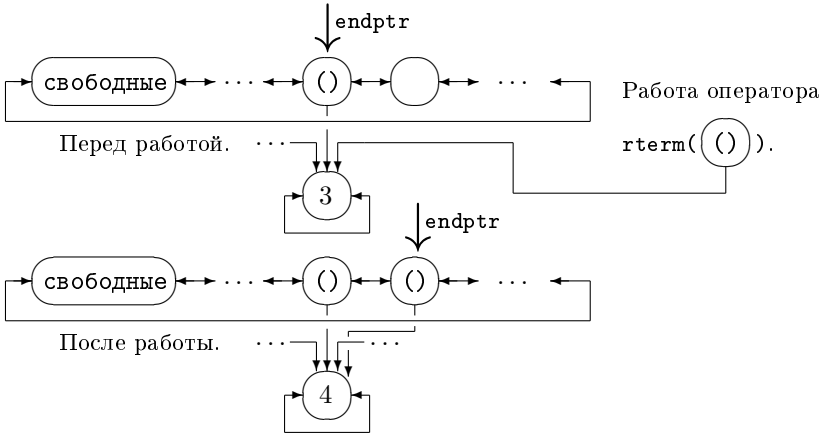


Рис. 20

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.
}

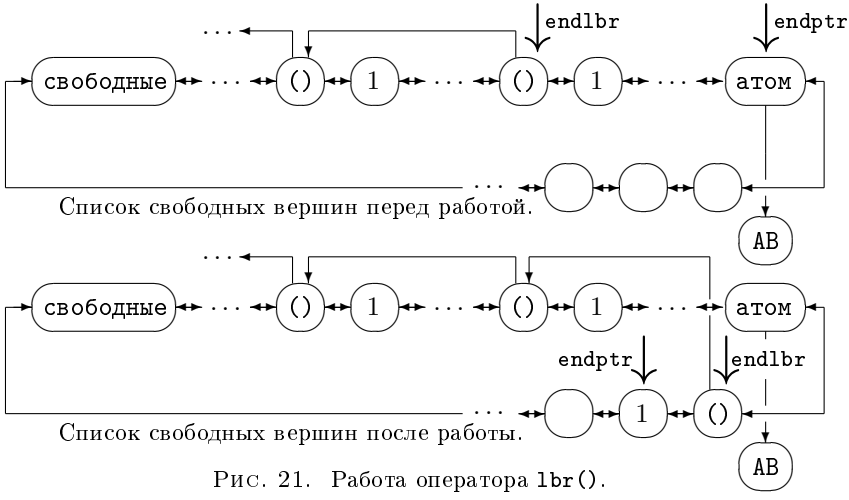


Рис. 21. Работа оператора `lbr()`.

`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.
}

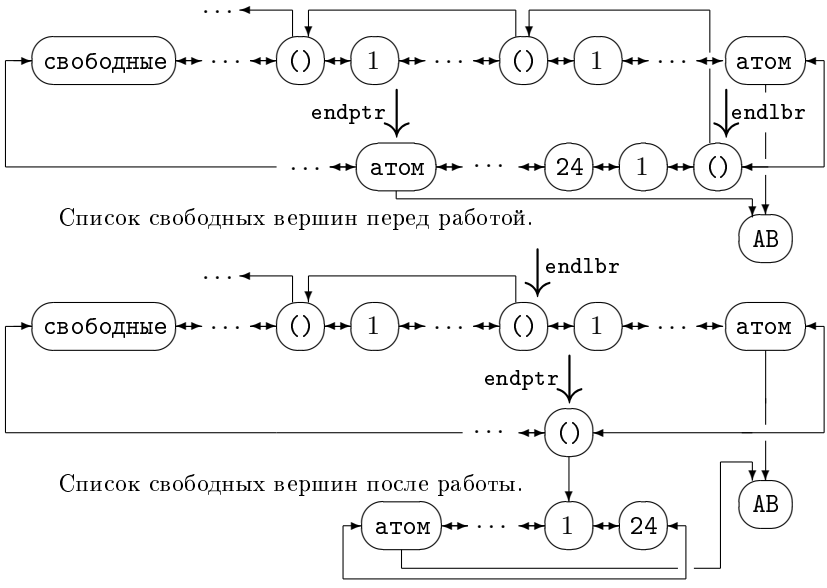


Рис. 22. Работа оператора `rbr()`.

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.

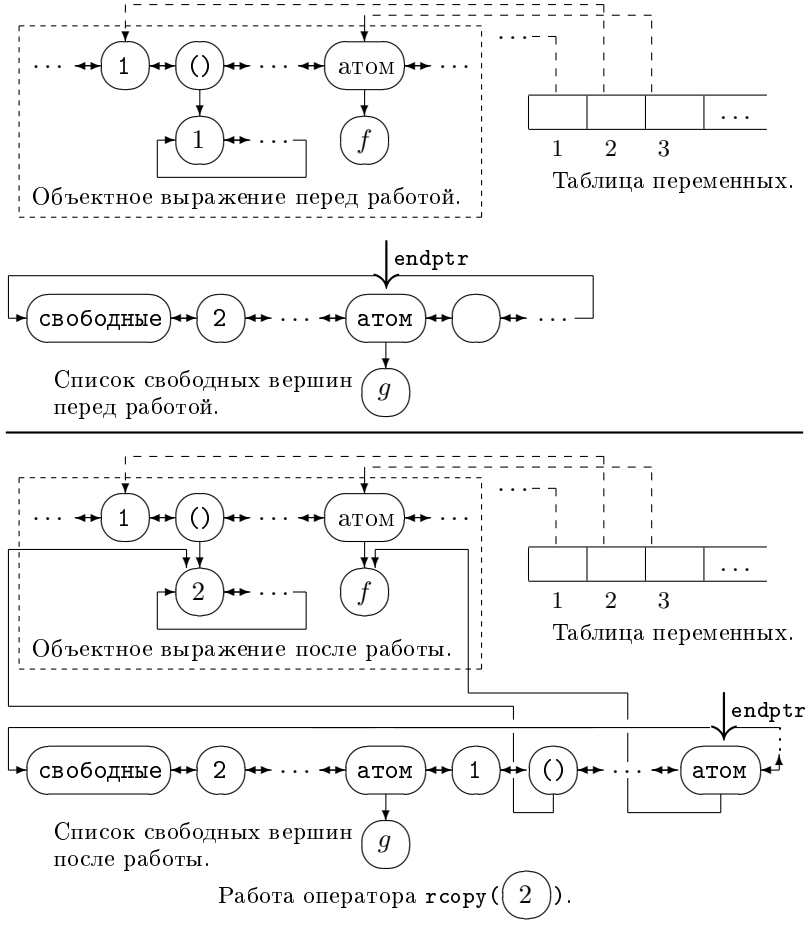


Рис. 23

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.

```
}
```

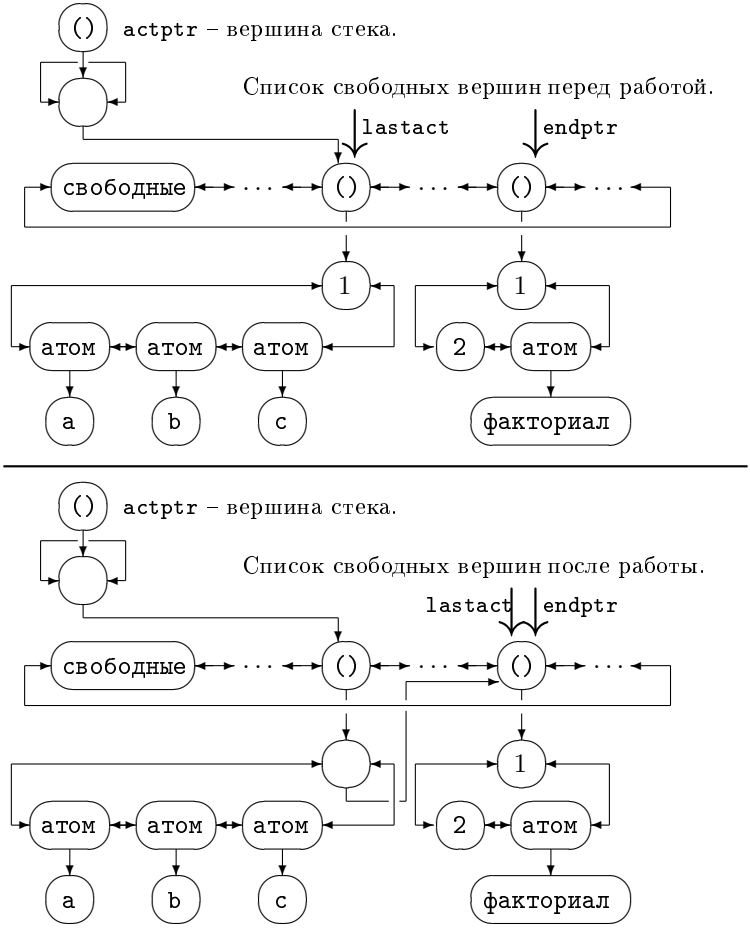


Рис. 24. Работа оператора `act()`.

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.

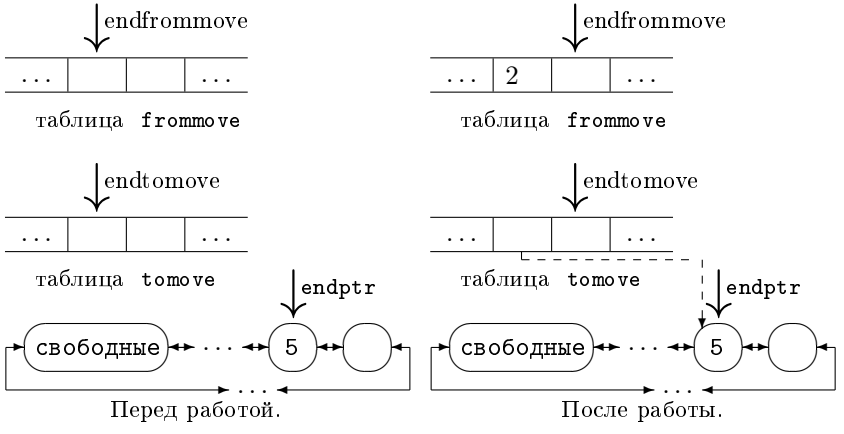
(Пример: `f( #x ) = #x a #x`. Стрелки указывают перестановки.)

`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 ())`.

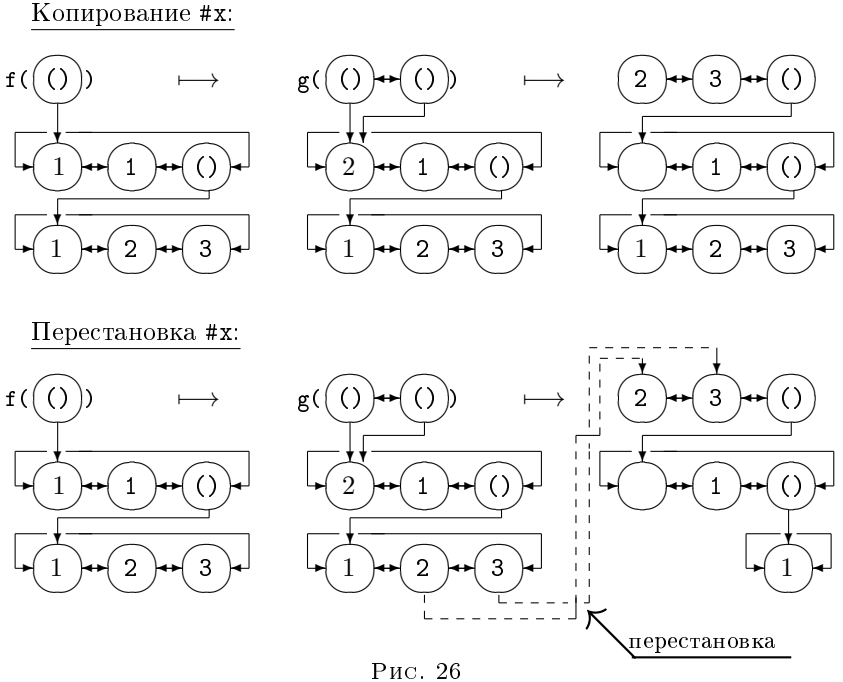


Рис. 26

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.
`}`

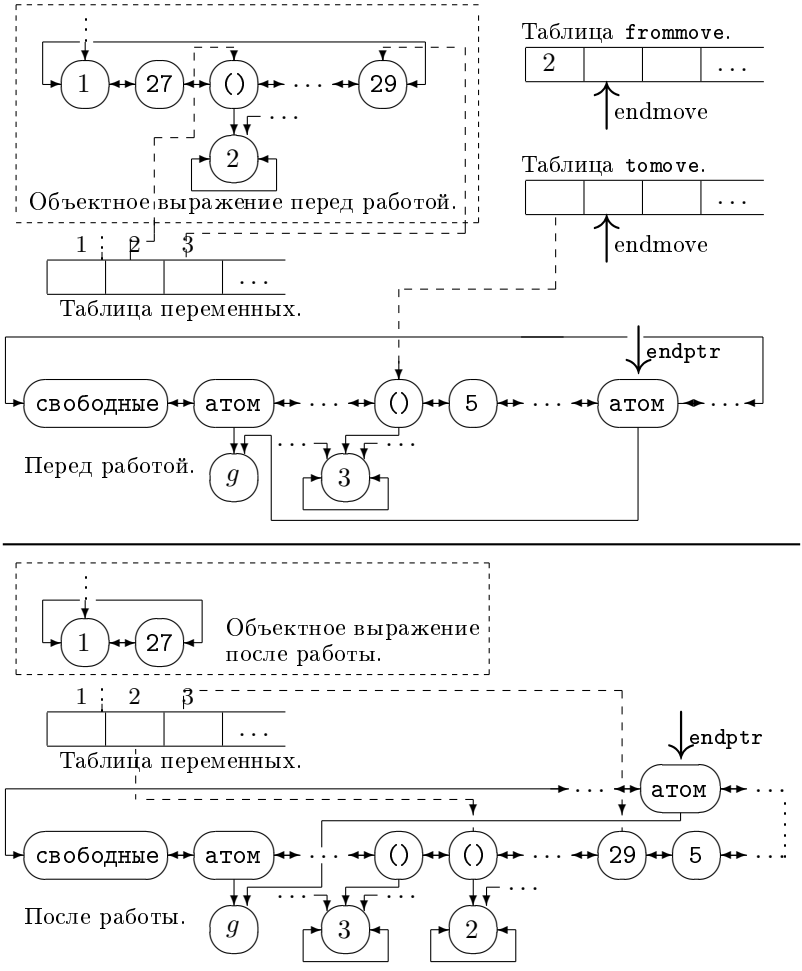


Рис. 27. Работа оператора `subst()`.

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 $\{e_n\}$, $\{r_m\}$.

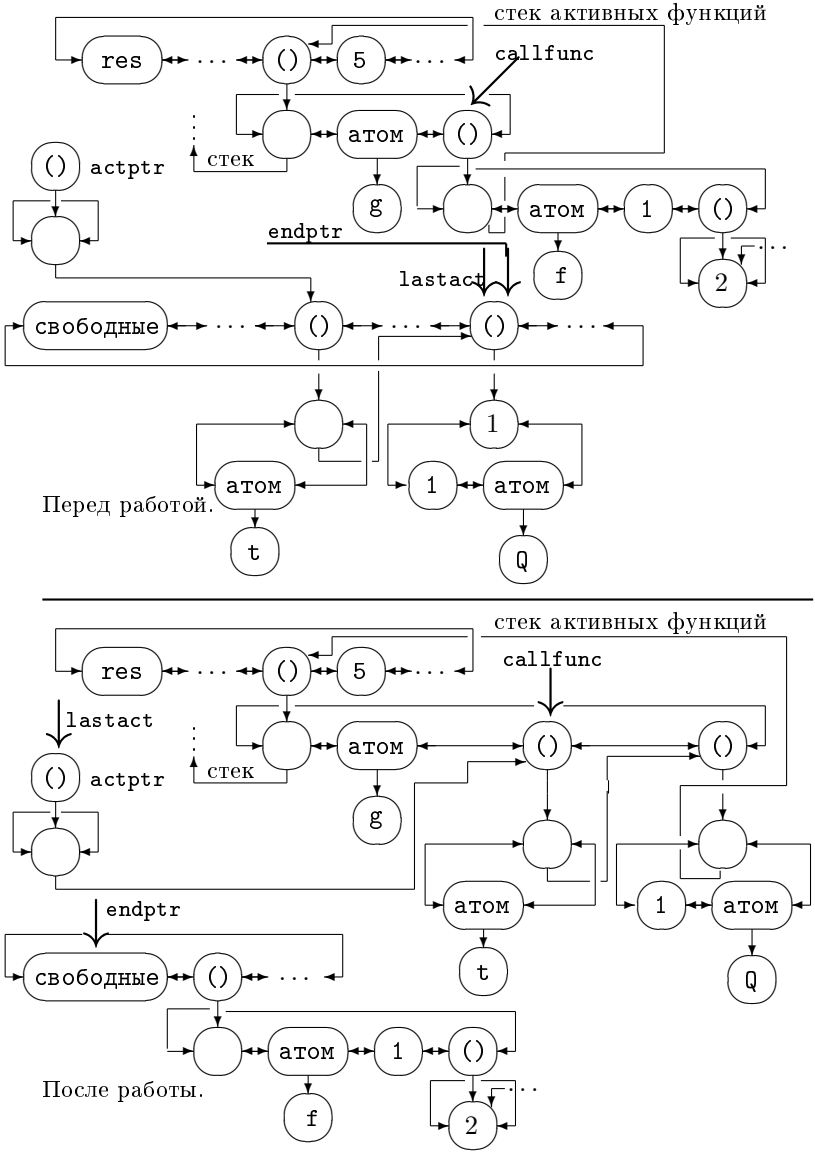


Рис. 28. Работа оператора `end()`.

§ 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).

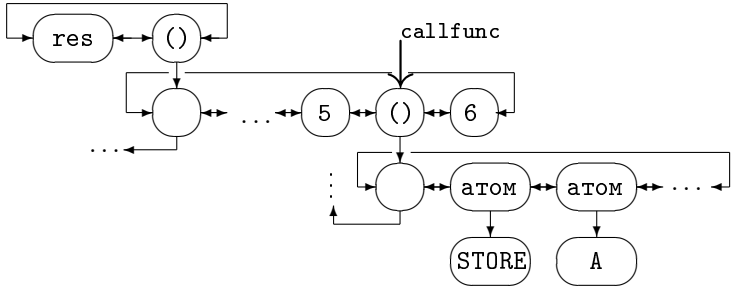
```
2. CFLAC( f( &x )  = g( &x &x ) )
            = f( (10 18) (1 g 4 0 4 0 5 6))
```

```
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.

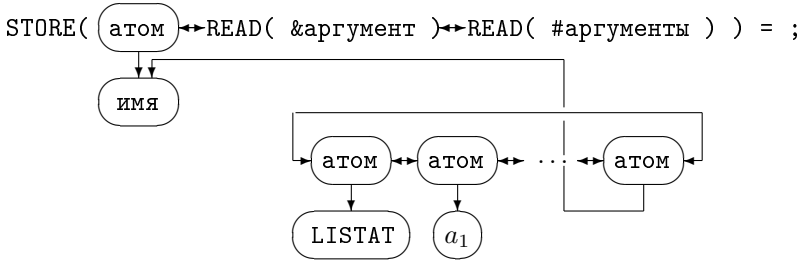


Таблица атомов перед вычислением STORE.

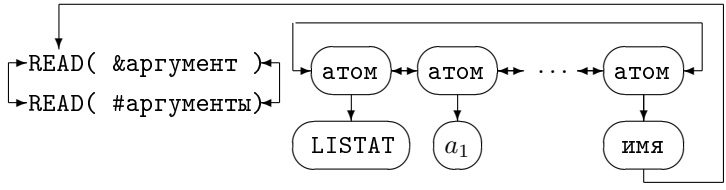
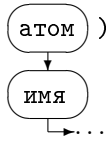


Таблица атомов после вычисления STORE.

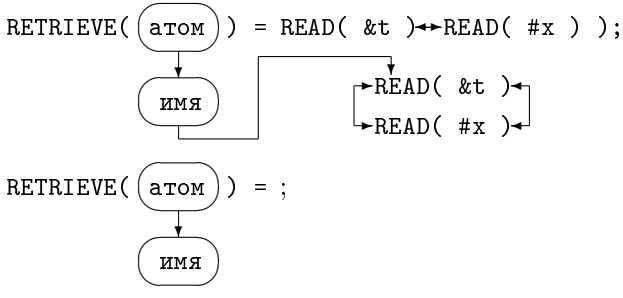
Вызов STORE(атом) уничтожает связь от внутренности аргумента.



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.

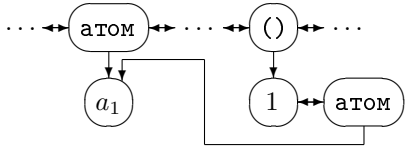
Содержимое FLAC-машины [см. описание п.4] дополняет программами функция `LOAD`. Ее аргумент – имя файла с FAL программой. Эта функция реализует механизм модульности; написана на языке FLAC.

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

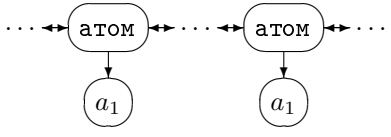
Определение 1. Два графа-атома равны тогда и только тогда, когда «внутренность» первого атома является «внутренностью» второго (*«внутренности» представлены одной вершиной графа*).

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

Примеры:

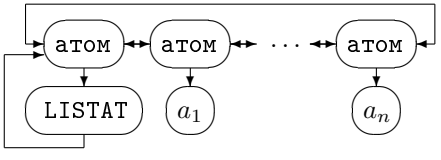


Атомы равны.



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

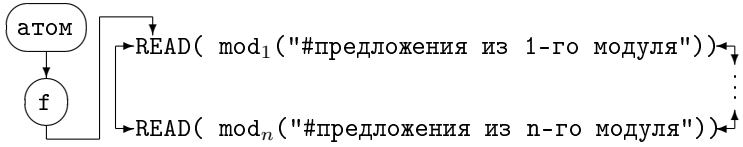
```
"имя модуля"(
  "#список имен всех функций в этом модуле"
            ) .
```

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 */
```

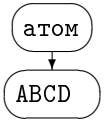
Пример:

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]].

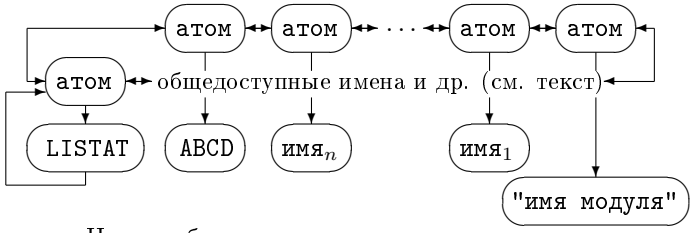
Следовательно, в новую таблицу, при ее инициализации, данный атом не попадет, хотя его «внутренность» имеется в системе.

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

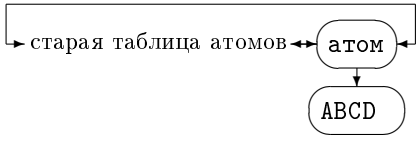


, строит его и добавляет в список. «Внутренность» построенного атома не совпадает с «внутренностью» атома с тем же именем, но существовавшем в системе до работы `LOAD` (так как одна «внутренность» уже существовала, другую построили в данный момент).

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.

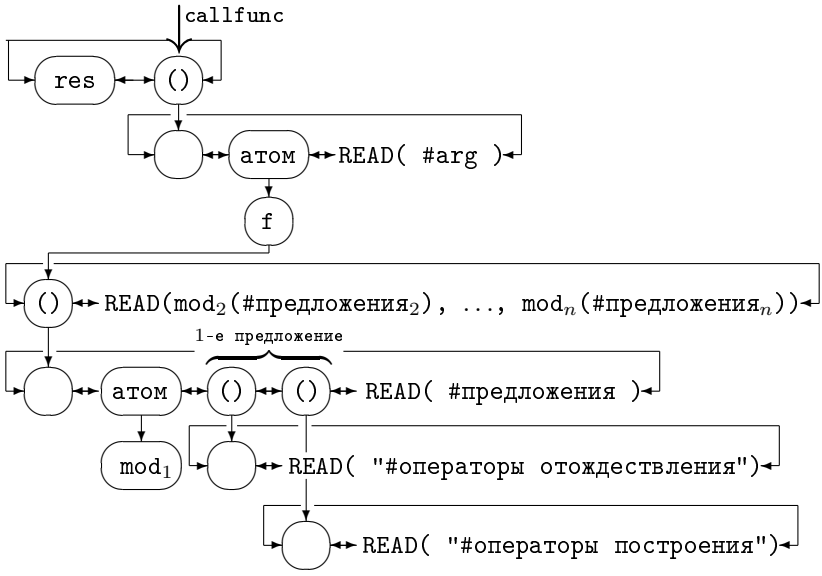
В момент включения FLAC-машины она содержит одну программу – диалог. Диалог представляет собой зацикленную программу, поддерживающую интерфейс системы с пользователем. Функция Begin (см. описание FLAC-машины) – это первая функция диалога.

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].

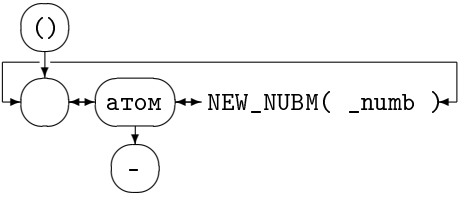
Пусть функция Calc переводит десятичное представление натурального числа в его представление в системе счисления по основанию β ($\beta \in \mathbb{N}_+$); упорядоченную последовательность десятичных цифр в упорядоченную последовательность β-цифр.

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

($d_i, D_j \in \mathbb{N}$, 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.

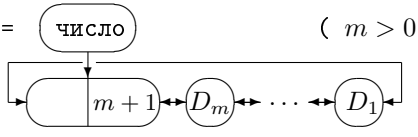
NEW_NUMB((- _numb)) =



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

NUMBER(D_0) = (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, *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