

# Лекция 8

## **Стадия синтеза на примере компилятора Простого Рефала**

## **Введение** в диалект Простого Рефала.

*Простой Рефал* — это диалект Рефала, ориентированный на компиляцию в исходный текст на C++. Разрабатывался с целью изучить особенности компиляции Рефала в императивные языки. Особенности:

- Поддержка только подмножества Базисного Рефала (предложения имеют вид *образец = результат*), отсутствие более продвинутых возможностей (условия, откаты, действия).
- Поддержка вложенных функций.
- Простая схема кодогенерации, отсутствие каких-либо мощных оптимизаций.
- Является самоприменимым компилятором.
- В основе лежит классическая списковая реализация.

## Типы данных Простого Рефала

Основной (да и единственный) тип данных Рефала — объектное выражение — последовательность объектных термов.

Разновидности объектных термов:

- Атомы:
  - ASCII-символы. Примеры: 'a', 'c', 'ы'.
  - Целые числа в диапазоне  $0 \dots (2^{32} - 1)$ . Примеры: 42, 121.
  - Замыкания (сочетание указателя на функцию и значений некоторых локальных переменных) — создаются из глобальных функций или безымянных вложенных функций. Примеры: Fact, Go, { t.B = (t.A t.B); }.
  - Идентификаторы. Примеры: #True, #Success.
- Составные термы:
  - Структурные скобки.

# Синтаксис Простого Рефала

Т.к. одной из задач при проектировании языка было написание максимально простого генератора кода C++, синтаксис языка наследует некоторые черты целевого языка, в частности необходимость предобъявлений.

**Пример.** Программа, заменяющая 'a' на 'b':

```
// объявления библиотечных функций
$EXTERN ReadLine, WriteLine;
// объявление локальной функции
$FORWARD Fab;
// Точка входа в программу
$ENTRY Go {
    =
    <WriteLine
        <Fab <ReadLine>>
    >;
}

Fab {
    e.Begin 'a' e.End = e.Begin 'b' <Fab e.End>;

    e.Other = e.Other;
}
```

## Пример. Выполнение программы, заменяющей 'a' на 'b'.

```
// В начале выполнения программы поле зрения инициализируется
// вызовом функции <Go>
<Go>
<writeLine <Fab <ReadLine>>>
// Здесь происходит приостановка выполнения Рефал-машины,
// Ожидается ввод пользователя.
// Пользователь вводит 'abracadabra!!!'.
<writeLine <Fab 'abracadabra!!!'>>
<writeLine 'b' <Fab 'bracadabra!!!'>>
<writeLine 'bbrb' <Fab 'cadabra!!!'>>
<writeLine 'bbrbcb' <Fab 'dabra!!!'>>
<writeLine 'bbrbcbdb' <Fab 'bra!!!'>>
<writeLine 'bbrbcbdbbbrb' <Fab '!!!!'>>
<writeLine 'bbrbcbdbbbrb!!!!'>
// Здесь происходит вывод на экран 'abrbcbdbbbrb!!!!',
// поле зрения становится пустым, рефал-машина останавливается.
```

## Пример. Программа со вложенными функциями.

```
// функция Map преобразует каждый терм выражения согласно заданному правилу
// Вызов <Map s.Trans e.Elems> == e.Transformed
$ENTRY Map {
    s.Trans t.First e.Tail = <s.Trans t.First> <Map s.Trans e.Tail>;

    s.Trans = /* пусто */;
}

// функция CardProd вычисляет декартово произведение двух множеств
// Вызов <CardProd ('a' 'b' 'c') (1 2)>
// == ('a' 1) ('a' 2) ('b' 1) ('b' 2) ('c' 1) ('c' 2)
$ENTRY CardProd {
    (e.SetA) (e.SetB) =
        <Map
            {
                t.A =
                    <Map
                        { t.B = (t.A t.B); }
                        e.SetB
                    >;
            }
            e.SetA
        >;
}
```

# Абстрактная рефал-машина

**Определение.** *Рефал-машиной* называется абстрактное устройство, которое выполняет программы на Рефале.

**Определение.** *Определённым выражением* называется выражение, содержащее скобки конкретизации, но при этом не содержащее переменных.

**Определение.** Определённое выражение, обрабатываемое рефал-машиной, называется *полем зрения*.

Работа рефал-машины осуществляется в пошаговом режиме. За один шаг рефал-машина находит в поле зрения *первичное активное подвыражение* (самую левую пару скобок конкретизации, не содержащую внутри себя других скобок конкретизации), вызывает замыкание, следующее за открывающей скобкой с выражением между этим замыканием и закрывающей скобкой в качестве аргумента.

Затем ведущая пара скобок заменяется на определённое выражение, являющееся результатом выполнения замыкания, и рефал-машина переходит к следующему шагу.

Выполнение рефал-машины продолжается до тех пор, пока поле зрения будет содержать скобки конкретизации.

## § 40. Структуры данных Простого Рефала

- Поле зрения представляется в виде двусвязного списка.
- Узлы списка содержат тег типа (tag) и поле информации (info). Узел (в зависимости от типа) может представлять собой атом, одну из структурных скобок или одну из скобок конкретизации.
- Узлы-атомы (числа, идентификаторы, символы, замыкания без контекста) в поле info содержат само значение атома.
- Узел, представляющий структурную скобку, в поле info содержит ссылку на соответствующую ему парную скобку. Это обеспечивает эффективное (за постоянное время) распознавание скобок в образце.
- Открывающие угловые скобки содержат ссылки на соответствующие закрывающие скобки.
- Закрывающие угловые скобки указывают на открывающие угловые скобки, которые станут лидирующими после выполнения текущей пары скобок конкретизации. Таким образом, угловые скобки образуют стек вызовов функций.
- Для ускорения операций создания новых узлов, а также для предотвращения утечек памяти, используется список свободных узлов.



## Структура узла

Для наглядности некоторые типы узлов в DataTag и некоторые поля в объединении пропущены.

```
typedef struct Node *NodePtr;  
typedef struct Node *Iter;
```

```
typedef enum DataTag {  
    cDataIllegal = 0,  
    cDataChar,  
    cDataNumber,  
    cDataFunction,  
    cDataIdentifier,  
    cDataOpenBracket,  
    cDataCloseBracket,  
    cDataOpenCall,  
    cDataCloseCall,  
    cDataClosure,  
    cDataClosureHead  
} DataTag;
```

```
typedef  
    FnResult (*RefalFunctionPtr) (  
        Iter begin, Iter end  
    );
```

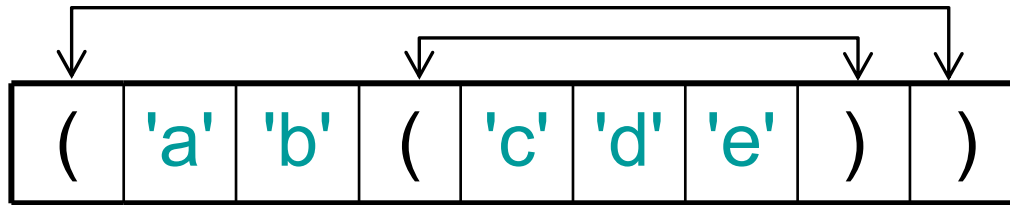
```
typedef struct RefalFunction {  
    RefalFunctionPtr ptr;  
    const char *name;  
} RefalFunction;
```

```
typedef unsigned long RefalNumber;
```

```
typedef const char  
    *(*RefalIdentifier) ();
```

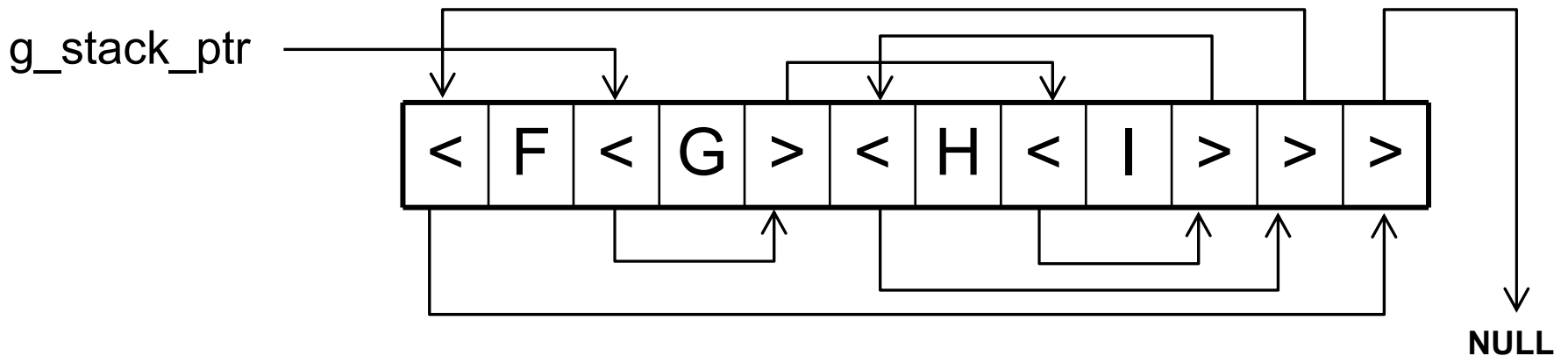
```
typedef struct Node {  
    NodePtr prev;  
    NodePtr next;  
    DataTag tag;  
    union {  
        char char_info;  
        RefalNumber number_info;  
        RefalFunction function_info;  
        RefalIdentifier ident_info;  
        NodePtr link_info;  
    };  
} Node;
```

## Представление структурных скобок



## Представление угловых скобок

Угловые скобки образуют односвязный список, на голову которого указывает глобальная переменная `g_stack_ptr`.



## Представление замыканий с контекстом

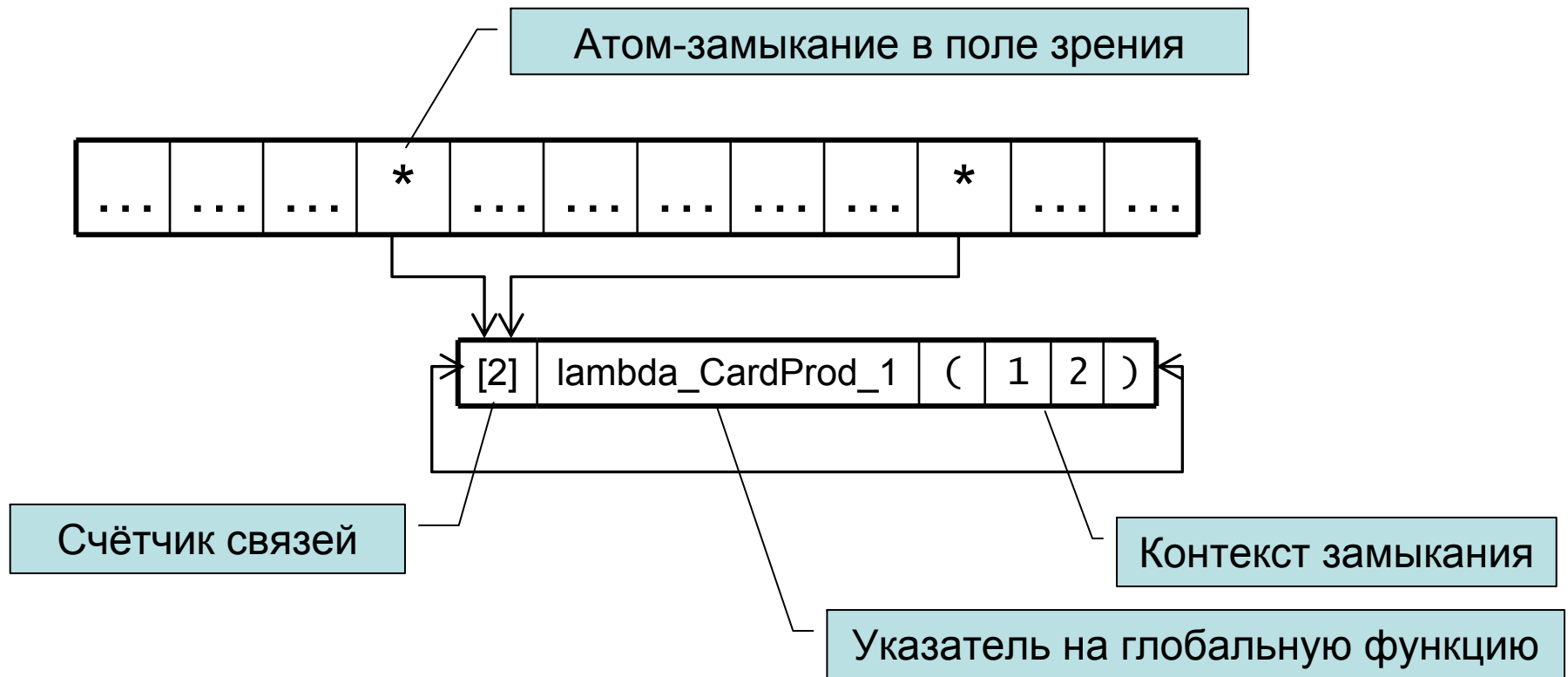
**Определение.** Контекстом замыкания вложенной функции называется множество переменных, связанных снаружи и используемых внутри функционального блока.

**Пример.** Контекстом внешней вложенной функции (#1) является переменная `e.SetB`.  
Контекстом внутренней вложенной функции (#2) является переменная `t.A`.

```
$ENTRY CartProd {  
  (e.SetA) (e.SetB) =  
    <Map  
      { // #1  
        t.A =  
          <Map  
            { t.B = (t.A t.B); } // #2  
            e.SetB  
          >;  
      }  
    e.SetA  
  >;  
}
```

Вложенные функции неявно преобразуются в глобальные функции и операции связывания с контекстом.

```
lambda_CartProd_0 {  
  t.A t.B = (t.A t.B);  
}  
  
lambda_CartProd_1 {  
  (e.SetB) t.A =  
    <Map  
      <refalrts::create_closure lambda_CartProd_0 t.A>  
      e.SetB  
    >;  
}  
  
$ENTRY CartProd {  
  (e.SetA) (e.SetB) =  
    <Map  
      <refalrts::create_closure lambda_CardProd_1 (e.SetB)>  
      e.SetA  
    >;  
}
```



Замыкания с контекстом реализованы как кольцевой список, содержащий счётчик связей, имя соответствующей глобальной функции и контекст.

Атомы-замыкания из поля зрения (или из контекстов других замыканий) указывают на счётчик связей.

Т.к. при копировании атома-замыкания сам контекст не копируется, то, чтобы отслеживать число указателей на замыкание (и в нужный момент его удалить), используется счётчик связей.

**§ 41.** Общая схема генерации кода в Простом Рефале  
Компиляция осуществляется независимыми друг от друга фрагментами, которые представляют собой объявления и отдельные предложения функций.

### Код на Рефале

```
// Объявления библиотечных функций
$EXTERN WriteLine, Dec, Mul;
// Объявление локальной функции
$FORWARD Fact;
// Точка входа в программу
$ENTRY Go {
    = <WriteLine '6! = ' <Fact 6>>;
}
```

```
Fact {
    0 = 1;
    s.Number =
        <Mul
            s.Number
            <Fact <Dec s.Number>>
        >;
}
```

### Код на C++

```
// Automatically generated file. Don't edit!
#include "refalrts.h"

extern refalrts::FnResult WriteLine(refalrts::Iter
    arg_begin, refalrts::Iter arg_end);

extern refalrts::FnResult Dec(refalrts::Iter arg_begin,
    refalrts::Iter arg_end);

extern refalrts::FnResult Mul(refalrts::Iter arg_begin,
    refalrts::Iter arg_end);

static refalrts::FnResult Fact(refalrts::Iter arg_begin,
    refalrts::Iter arg_end);

refalrts::FnResult Go(refalrts::Iter arg_begin,
    refalrts::Iter arg_end) {
    Код предложения
    return refalrts::cRecognitionImpossible;
}

static refalrts::FnResult Fact(refalrts::Iter arg_begin,
    refalrts::Iter arg_end) {
    Код первого предложения
    Код второго предложения
    return refalrts::cRecognitionImpossible;
}

//End of file
```

**Генерация идентификаторов** основана на том, что дублирующиеся instantiation шаблонов в C++ в разных единицах трансляции как правило устраняются компоновщиком.

## Код на Рефале

```
$LABEL Success;
```

```
$LABEL Fails;
```

```
F {
```

```
  #Success =
```

```
  #Fails;
```

```
}
```

## Код на C++

```
// Automatically generated file. Don't edit!  
#include "refalrts.h"
```

```
//$LABEL Success  
template <typename T>  
struct SuccessL_ {  
    static const char *name() {  
        return "Success";  
    }  
};
```

```
//$LABEL Fails  
template <typename T>  
struct FailsL_ {  
    static const char *name() {  
        return "Fails";  
    }  
};
```

```
static refalrts::FnResult F(refalrts::Iter arg_begin,  
    refalrts::Iter arg_end) {
```

```
    ...  
    ... & SuccessL_<int>::name ...  
    ...  
    ... & FailsL_<int>::name ...  
    ...  
}
```

```
//End of file
```

Используется следующая структура функции:

```
refalrts::FnResult  
FunctionName(refalrts::Iter arg_begin, refalrts::Iter arg_end) {  
    ...  
    do {  
        Код предложения N  
    } while(0);  
    ...  
    return refalrts::cRecognitionImpossible;  
}
```

Логика выполнения такая:

1. В случае успешного выполнения, выход из предложения осуществляется инструкцией ***return refalrts::cSuccess.***
2. При недостатке памяти функция завершается инструкцией ***return refalrts::cNoMemory.***
3. При неуспешном сопоставлении с образцом выполняется инструкция ***break.*** Для последнего предложения происходит переход к следующему предложению, в случае последнего осуществляется возврат ***refalrts::cRecognitionImpossible.***



## Три стадии выполнения предложения

Для удобства отладки функция разделена на три стадии:

1. *Сопоставление с образцом.* На этом этапе содержимое терма активации (угловые скобки, имя функции и сам аргумент) не изменяется, чтобы в случае неудачи сопоставления следующее предложение получило аргумент в том же виде, а если предложение последнее, то чтобы по дампу поля зрения можно было понять, в каком случае функция рухнула.
2. *Распределение памяти для новых узлов.* На этом этапе начало списка свободных блоков инициализируется новыми значениями (копии переменных, новые узлы-литералы: атомы, скобки). Содержимое терма активации здесь тоже не изменяется из соображений отладки.
3. *Построение результата.* Т.к. построение осуществляется только путём изменения указателей двусвязного списка, эта стадия не может завершиться неуспешно. Те части терма активации, которые не понадобились в результате, переносятся в список свободных блоков. Этот этап всегда завершается инструкцией ***return refalrts::cSuccess;***

## Псевдокод предложения

```
refalrts::FnResult FunctionName(refalrts::Iter arg_begin, refalrts::Iter
    arg_end) {
    // Первое предложение
    do {
        // 1 стадия – сопоставление с образцом
        if( сопоставление неуспешно )
            break;
        // 2 стадия – распределение памяти
        if( недостаточно памяти )
            return refalrts::cNoMemory;
        // 3 стадия – построение результата
        ...
        return refalrts::cSuccess;
    } while(0);

    // Второе предложение
    do {
        ...
    } while(0);

    // Возврат при неудаче распознавания
    return refalrts::cRecognitionImpossible;
}
```

## § 42. Генерация кода для сопоставления с образцом в Простом Рефале

Левая часть предложения преобразуется в последовательность элементарных команд распознавания по достаточно сложному алгоритму. Элементарными называются команды отщепления заданного атома, скобочного терма, нераспознанной переменной, повторной переменной от правого или левого конца объектного выражения.

### **Особенности сопоставления с образцом:**

- Объектные выражения представляются парой указателей на диапазон узлов поля зрения.
- В начале сопоставления мы имеем только пару указателей на диапазон аргумента.
- При успешном отщеплении жёсткого элемента — элемента с уже известной длиной: атома, скобочного терма, s- и t-переменной, повторной e-переменной — один из указателей диапазона смещается на длину распознанного отщеплённого элемента.

## Особенности сопоставления с образцом (продолжение):

- Успешное отщепление скобочного терма помимо изменения диапазона, приводит к инициализации новой пары указателей диапазона для подвыражения в скобках.
- Успешное отщепление переменной (s-, t- или повторной) помимо изменения диапазона приводит к инициализации указателя на эту переменную (для e-переменных — пары указателей).
- Сопоставление с закрытой e-переменной всегда выполняется успешно. В результате сопоставления пара указателей на e-переменную инициализируется парой указателей диапазона.
- Существует команда сопоставления диапазона с пустым выражением.
- Команда сопоставления с диапазоном, содержащим открытую e-переменную транслируется в цикл удлинения e-переменной.
- При невозможности сопоставления вне цикла удлинения открытой e-переменной выполняется инструкция ***break;***, внутри цикла — инструкция ***continue;***

## Особенности распознавания открытых е-переменных

- Перед циклом удлинения пара указателей на открытую е-переменную инициализируется как пустой диапазон.
- На каждой итерации цикла длина диапазона увеличивается на один терм.
- Цикл продолжается до тех пор, пока правый конец открытой е-переменной не выйдет за пределы допустимого диапазона.
- При неуспешном сопоставлении внутри цикла удлинения е-переменной выполняется инструкция ***continue***;, приводящая к следующей итерации цикла.

```
do {  
    // Сопоставление вне цикла  
    if( сопоставление неуспешно )  
        break;  
    // Цикл удлинения открытой е-переменной  
    for( инициализация; проверка на допустимость длины; удлинение ) {  
        // Сопоставление внутри цикла  
        if( сопоставление неуспешно )  
            continue;  
        ...  
        return refalrts::cSuccess;  
    }  
} while(0);
```

**Пример.** Генерация образца без открытых е-переменных.  
Для наглядности префикс *refalrts::* убран.

### Код на Рефале

```
$LABEL A;  
$LABEL B;  
  
F {  
  (e.X #A) e.Y #B = результат;  
}
```

### Псевдокод

- $V0 \leftarrow \text{аргумент функции}$
- $V0 \rightarrow V0 \#B$
- $V0 \rightarrow (V1) V0$
- $V1 \rightarrow V1 \#A$
- $V1 \rightarrow e.X$
- $V0 \rightarrow e.Y$
- *Построение результата*

### Код на C++

```
...  
do {  
  Iter bb_0 = arg_begin;  
  Iter be_0 = arg_end;  
  move_left( bb_0, be_0 );  
  move_left( bb_0, be_0 );  
  move_right( bb_0, be_0 );  
  static Iter eX_b_1, eX_e_1, eY_b_1, eY_e_1;  
  // (~1 e.X # A )~1 e.Y # B  
  if( ! ident_right( & BL_<int>::name, bb_0, be_0 ) )  
    break;  
  Iter bb_1 = 0, be_1 = 0;  
  if( ! brackets_left( bb_1, be_1, bb_0, be_0 ) )  
    break;  
  if( ! ident_right( & AL_<int>::name, bb_1, be_1 ) )  
    break;  
  eX_b_1 = bb_1;  
  eX_e_1 = be_1;  
  eY_b_1 = bb_0;  
  eY_e_1 = be_0;  
  
  Построение результата  
  
} while ( 0 );  
...
```

**Пример.** Генерация образца с открытыми е-переменными. В начале цикла происходит сохранение состояния вычислений. Вместо инструкции *break* используется инструкция *continue*. Для наглядности префикс *refalrts::* убран.

### Код на Рефале

```
$LABEL A;

F {
  e.X #A e.Y = результат;
}
```

### Псевдокод

- $V0 \leftarrow$  аргумент функции
- **cycle (e.X V0)**
- $V0 \rightarrow \#A V0$
- $V0 \rightarrow e.Y$
- *Построение результата*
- **End of cycle**

### Код на C++

```
...
do {
  Iter bb_0 = arg_begin;
  Iter be_0 = arg_end;
  move_left( bb_0, be_0 );
  move_left( bb_0, be_0 );
  move_right( bb_0, be_0 );
  static Iter ex_b_1, ex_e_1, ey_b_1, ey_e_1;
  // e.X # A e.Y
  Iter bb_0_stk = bb_0, be_0_stk = be_0;
  for(
    Iter
      ex_b_1 = bb_0_stk, ex_oe_1 = bb_0_stk,
      bb_0 = bb_0_stk, be_0 = be_0_stk;
    ! empty_seq( ex_oe_1, be_0 );
    bb_0 = bb_0_stk, be_0 = be_0_stk, next_term( ex_oe_1, be_0 )
  ) {
    bb_0 = ex_oe_1;
    ex_b_1 = bb_0_stk;
    ex_e_1 = ex_oe_1;
    move_right( ex_b_1, ex_e_1 );
    if( ! ident_left( & AL_<int>::name, bb_0, be_0 ) )
      continue;
    ey_b_1 = bb_0;
    ey_e_1 = be_0;

    Построение результата

  }
} while ( 0 );
...
```

## **§ 43. Основные принципы генерации кода для построения результата функции в Простом Рефале**

В отличии от первой стадии, правая часть предложения преобразуется в последовательность элементарных команд по более простому алгоритму. К элементарным командам распределения памяти относятся команды создания атомов, создания скобок и копирования переменных. К элементарным командам построения результата относятся команды сборки результата из фрагментов, связывания пар структурных скобок и помещения угловых скобок на стек вызовов.

### **Особенности стадий распределения памяти и построения результата**

- Все элементарные команды поддерживают инварианты двусвязных списков поля зрения и свободных узлов. Т.е. между вызовами элементарных операций не происходит ни «разрывов» списков, ни возникновения «висячих» фрагментов.
- Фрагменты, требуемые для построения результата вызова функции, находятся либо в поле зрения, либо в списке свободных блоков.



## Особенности стадий распределения памяти и построения результата (продолжение)

- Сборка результата осуществляется операциями переноса вида *splice(pos, begin, end)*, где *begin* и *end* — указатели на начало и конец переносимого фрагмента, *pos* — указатель на узел, *перед* которым будет помещён фрагмент.
- Фрагменты, из которых строится результат, помещаются в поле зрения перед открывающей скобкой вызова текущей функции. Туда переносятся как фрагменты из списка свободных узлов, так и переменные, присутствующие в образце. Таким образом, после сборки результата между скобками вызова функции будут находиться только ненужные фрагменты поля зрения — они переносятся в список свободных узлов.

## Пример. Генерация распределения памяти и сборки результата.

[\(Сгенерированный код1\)](#) [\(Сгенерированный код2\)](#)

### Код на Рефале

```
Fab {  
  e.X #A e.Y =  
    e.X #B <Fab e.Y>;  
  
  e.X = e.X;  
}
```

### Псевдокод

- *// первое предложение*
- **cycle** (e.X<sub>1</sub>, B0)
- B0 → #A B0
- B0 → e.Y<sub>1</sub>
- n0 ← allocate(#B)
- n1 ← allocate(<)
- n2 ← allocate(Fab)
- n3 ← allocate(>)
- Push(n3)
- Push(n0)
- Build(e.X<sub>1</sub>, n0, n1, n2, e.Y<sub>1</sub>, n3)
- Free(arg\_begin, arg\_end)
- **return** cSuccess
- **End of cycle**
  
- *// второе предложение*
- B0 → e.X<sub>1</sub>
- Build(e.X<sub>1</sub>)
- Free(arg\_begin, arg\_end)
- **return** cSuccess

## Пример. Генерация распределения памяти и сборки результата.

### Код на Рефале

```
Fact {  
  0 = 1;  
  s.Number =  
    <Mul  
      s.Number  
      <Fact <Dec s.Number>>  
    >;  
}
```

### Псевдокод (начало)

- // первое предложение
- $B0 \rightarrow 0 \ B0$
- $B0 \rightarrow \text{empty}$
- $n0 \leftarrow \text{allocate}(1)$
- $\text{Build}(n0)$
- $\text{Free}(\text{arg\_begin}, \text{arg\_end})$
- **return** cSuccess

### Псевдокод (продолжение)

- // второе предложение
- $B0 \rightarrow s.\text{Number}_1 \ B0$
- $B0 \rightarrow \text{empty}$
- $s.\text{Number}_2 \leftarrow \text{copy}(s.\text{Number}_1)$
- $n0 \leftarrow \text{allocate}(<)$
- $n1 \leftarrow \text{allocate}(\text{Mul})$
- $n2 \leftarrow \text{allocate}(<)$
- $n3 \leftarrow \text{allocate}(\text{Fact})$
- $n4 \leftarrow \text{allocate}(<)$
- $n5 \leftarrow \text{allocate}(\text{Dec})$
- $n6 \leftarrow \text{allocate}(>)$
- $n7 \leftarrow \text{allocate}(>)$
- $n8 \leftarrow \text{allocate}(>)$
- $\text{Push}(n8)$
- $\text{Push}(n0)$
- $\text{Push}(n7)$
- $\text{Push}(n2)$
- $\text{Push}(n6)$
- $\text{Push}(n4)$
- $\text{Build}(n0, n1, s.\text{Number}_1, n2, n3, n4, n5, s.\text{Number}_2, n6)$
- $\text{Free}(\text{arg\_begin}, \text{arg\_end})$
- **return** cSuccess

## §44. Два способа построения результата функции: прямая кодогенерация и интерпретация

В коде на C++ могут быть как явно прописаны элементарные операции последних двух стадий в виде вызовов соответствующих функций (режим *прямой кодогенерации*), так и последовательность интерпретируемых команд в виде константного статического массива, который затем передаётся специальной функции-интерпретатору (режим *интерпретации*).

- Программа, скомпилированная в режиме прямой кодогенерации, выполняется быстрее, чем в режиме интерпретации (заметно только на старых машинах либо на больших объёмах вычислений).
- Размер программы, скомпилированной в режиме интерпретации, примерно на треть меньше размера программы, скомпилированной в режиме прямой кодогенерации.
- В сгенерированном тексте присутствует код, сгенерированный обоими режимами. Выбор режима осуществляется директивами условной компиляции препроцессора C++.

**Пример.** Стадии 2 и 3 для [примера с функцией Fab](#) (первое предложение). Режим прямой кодогенерации.

```
#ifdef INTERPRET
    ...
#else
    refalrts::reset_allocator();
    refalrts::Iter res = arg_begin;
    refalrts::Iter n0 = 0;
    if( ! refalrts::alloc_ident( n0, & BL_<int>::name ) ) return refalrts::cNoMemory;
    refalrts::Iter n1 = 0;
    if( ! refalrts::alloc_open_call( n1 ) ) return refalrts::cNoMemory;
    refalrts::Iter n2 = 0;
    if( ! refalrts::alloc_name( n2, & Fab, "Fab" ) ) return refalrts::cNoMemory;
    refalrts::Iter n3 = 0;
    if( ! refalrts::alloc_close_call( n3 ) ) return refalrts::cNoMemory;
    refalrts::push_stack( n3 );
    refalrts::push_stack( n1 );
    res = refalrts::splice_elem( res, n3 );
    res = refalrts::splice_evar( res, eY_b_1, eY_e_1 );
    res = refalrts::splice_elem( res, n2 );
    res = refalrts::splice_elem( res, n1 );
    res = refalrts::splice_elem( res, n0 );
    res = refalrts::splice_evar( res, eX_b_1, eX_e_1 );
    refalrts::splice_to_freelist( arg_begin, arg_end );
    return refalrts::cSuccess;
#endif
```

## Структура элементарной интерпретируемой команды

```
typedef enum iCmd {  
    icChar,  
    icInt,  
    icFunc,  
    icIdent,  
    icString,  
    icBracket,  
    icSpliceSTVar,  
    icSpliceEVar,  
    icCopySTVar,  
    icCopyEVar,  
    icEnd  
} iCmd;
```

```
typedef enum BracketType {  
    ibopenBracket,  
    ibopenCall,  
    ibCloseBracket,  
    ibCloseCall  
} BracketType;
```

```
typedef struct ResultAction {  
    iCmd cmd;  
    void *ptr_value1;  
    void *ptr_value2;  
    int value;  
} ResultAction;
```

// интерпретатор

```
extern FnResult interpret_array(  
    const ResultAction raa[],  
    Iter allocs[],  
    Iter begin,  
    Iter end  
);
```

**Пример.** Стадии 2 и 3 для [примера с функцией Fab](#) (первое предложение). Режим интерпретации.

```
#ifdef INTERPRET
    const static refalrts::ResultAction raa[] = {
        {refalrts::icSpliceEVar, & eX_b_1, & eX_e_1},
        {refalrts::icIdent, (void*) & BL_<int>::name},
        {refalrts::icBracket, 0, 0, refalrts::ibOpenCall},
        {refalrts::icFunc, (void*) & Fab, (void*) "Fab"},
        {refalrts::icSpliceEVar, & eY_b_1, & eY_e_1},
        {refalrts::icBracket, 0, 0, refalrts::ibCloseCall},
        {refalrts::icEnd}
    };
    refalrts::Iter allocs[2*sizeof(raa)/sizeof(raa[0])];
    refalrts::FnResult res = refalrts::interpret_array( raa, allocs, arg_begin, arg_end );
    return res;
#else
    ...
#endif
```

## § 45. Поддержка времени выполнения

Библиотека поддержки времени выполнения содержит следующие компоненты:

- Поддержка самого языка (файлы *refalrts.h* и *refalrts.cpp*) обеспечивает имитацию абстрактной рефал-машины и включает в себя такие функции:
  - определение структур данных узлов поля зрения и интерпретируемых команд;
  - реализации элементарных операций сопоставления с образцом, распределения памяти и сборки результата выполнения функции;
  - средства распределения памяти для новых узлов, поддержка списка свободных блоков;
  - вывод дампа поля зрения в отладочных целях;
  - основной цикл программы;
  - интерпретатор (для режима интерпретации).
- Библиотека функций, написанных на С++ (файл *Library.cpp*) включает в себя те функции, которые невозможно написать на Рефале: средства ввода-вывода, арифметические операции, операции преобразования атомов и т.д.



## Основной цикл программы

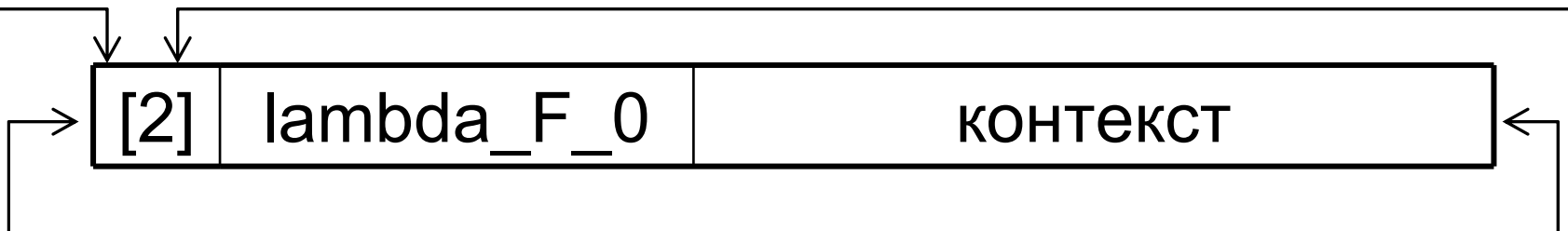
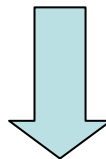
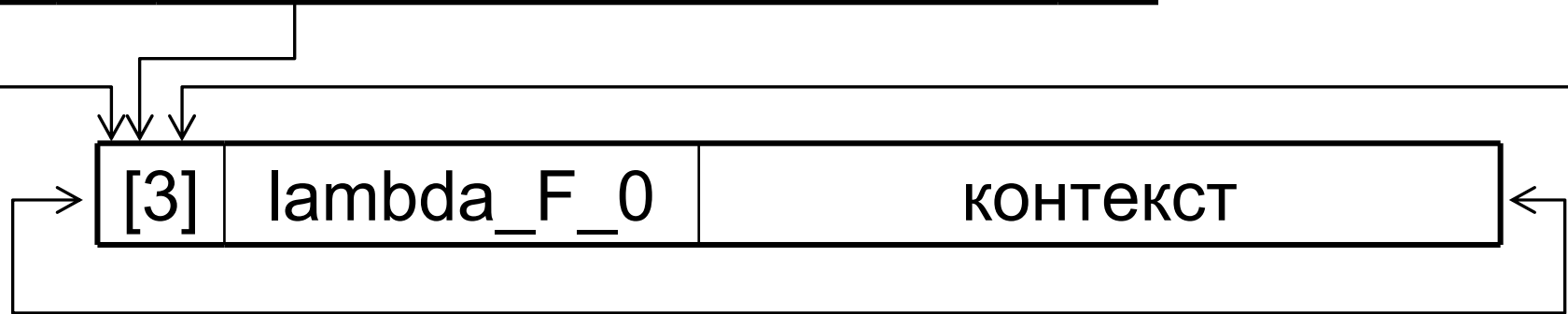
Библиотека времени выполнения моделирует работу рефал-машины в так называемом основном цикле программы. Каждая итерация основного цикла соответствует одному шагу рефал-машины и выполняется следующим образом:

- Находит в поле зрения следующее первичное активное подвыражение (как пару указателей на угловые скобки).
- Определяет тип атома, следующий за открывающей угловой скобкой. В зависимости от типа этого атома:
  - если слева от «<» находится атом, соответствующий глобальной функции, то данная функция вызывается, после чего анализируется результат выполнения функции: если он отличен от `refalrts::cSuccess`, происходит аварийный останов рефал-машины;
  - если слева от «<» находится атом-замыкание, в зависимости от значения счётчика ссылок копируется (больше 1) или переносится (равен 1) содержимое замыкания (указатель на глобальную функцию + контекст) в поле зрения на место атома-замыкания;
  - в остальных случаях происходит аварийный останов рефал-машины.

# Основной цикл программы (псевдокод)

```
refalrts::FnResult main_loop() {
    while( g_stack_ptr != NULL )
    {
        arg_begin = pop_stack();
        arg_end = pop_stack();
        if( arg_begin->tag == cDataFunction ) {
            result = (*arg_begin->function_info.ptr)(arg_begin, arg_end);
            if( result != cSuccess ) {
                return result;
            } else {
                continue;
            }
        } else if( arg_begin->tag == cDataClosure ) {
            if( счётчик связей > 1 ) {
                скопировать содержимое замыкания в поле зрения;
                --счётчик связей;
            } else {
                переместить содержимое замыкания в поле зрения;
            }
            push_stack(arg_end);
            push_stack(arg_begin);
            continue;
        } else {
            return cRecognitionImpossible;
        }
    }
    return cSuccess;
}
```

## Вставка содержимого замыкания в поле зрения



## Вставка содержимого замыкания в поле зрения (продолжение)

