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 $0…(2^{32} - 1)$. Примеры: 42, 121.
  - Замыкания — создаются из глобальных функций или безымянных вложенных функций. Примеры: Fact, Go.
  - 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, вычисляющая факториал

```
// 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>>
    >;
}
```

Пример. Программа со вложенными функциями.

(реализация)

```
// Библиотечные функции
$EXTERN ReadLine, WriteLine, Map;

Replace {
  s.In e.Map-B (s.Symbol e.Out) e.Map-E = e.Out;
  s.Other e.Map = s.Other;
}

MakeReplace {
  e.Replaces =
    { s.In = <Replace s.In e.Replaces>; };
}

// Точка входа в программу
$ENTRY Go {
  =
    <WriteLine
      <Map
        <MakeReplace ('&' '&amp;') ('<' '&lt;') ('>' '&gt;')>
        <ReadLine>
      >
    >;
}
```

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.

Выполнение функций на Простом Рефале

Функция на Простом Рефале представляет собой набор предложений. Предложение — это пара вида *образец* = *результат*. Каждое предложение может отработать с любым из трёх результатов:

1. Происходит успешное сопоставление с образцом, в соответствии с правой частью предложения строится результат выполнения функции, распределяется память для новых узлов списка. При этом функция успешно завершается.
2. Сопоставление с образцом выполняется успешно, но для построения результата памяти оказывается недостаточно. Происходит аварийная остановка рефал-машины с выдачей сообщения о недостатке памяти.
3. Сопоставление с образцом происходит неуспешно. При этом:
    1. Если предложение не последнее, выполняется переход к следующему предложению.
    2. Если предложение последнее, рефал-машина аварийно останавливается с выдачей сообщения о невозможности сопоставления.

Пример. Выполнение программы, вычисляющей факториал.

```
<Go>
<WriteLine '6! = ' <Fact 6>>
<WriteLine '6! = ' <Mul 6 <Fact <Dec 6>>>>
<WriteLine '6! = ' <Mul 6 <Fact 5>>>
<WriteLine '6! = ' <Mul 6 <Mul 5 <Fact <Dec 5>>>>>
<WriteLine '6! = ' <Mul 6 <Mul 5 <Fact 4>>>>
<WriteLine '6! = ' <Mul 6 <Mul 5 <Mul 4 <Fact <Dec 4>>>>>>
<WriteLine '6! = ' <Mul 6 <Mul 5 <Mul 4 <Fact 3>>>>>
<WriteLine '6! = ' <Mul 6 <Mul 5 <Mul 4 <Mul 3 <Fact <Dec 3>>>>>>>
<WriteLine '6! = ' <Mul 6 <Mul 5 <Mul 4 <Mul 3 <Fact 2>>>>>>
<WriteLine '6! = ' <Mul 6 <Mul 5 <Mul 4 <Mul 3 <Mul 2 <Fact <Dec 2>>>>>>>>
<WriteLine '6! = ' <Mul 6 <Mul 5 <Mul 4 <Mul 3 <Mul 2 <Fact 1>>>>>>>
<WriteLine '6! = ' <Mul 6 <Mul 5 <Mul 4 <Mul 3 <Mul 2 <Mul 1 <Fact <Dec
1>>>>>>>>>
<WriteLine '6! = ' <Mul 6 <Mul 5 <Mul 4 <Mul 3 <Mul 2 <Mul 1 <Fact 0>>>>>>>>
<WriteLine '6! = ' <Mul 6 <Mul 5 <Mul 4 <Mul 3 <Mul 2 <Mul 1 1>>>>>>>
<WriteLine '6! = ' <Mul 6 <Mul 5 <Mul 4 <Mul 3 <Mul 2 1>>>>>>
<WriteLine '6! = ' <Mul 6 <Mul 5 <Mul 4 <Mul 3 2>>>>>
<WriteLine '6! = ' <Mul 6 <Mul 5 <Mul 4 6>>>>
<WriteLine '6! = ' <Mul 6 <Mul 5 24>>>
<WriteLine '6! = ' <Mul 6 120>>
<WriteLine '6! = ' 720>
// Здесь происходит вывод на экран 6! = 720 и поле зрения становится пустым.
// Рефал-машина останавливается.
```

§ 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.
- Как правило, узлы-атомы 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 с памятью, а также для предотвращения утечек памяти, используется список свободных узлов.

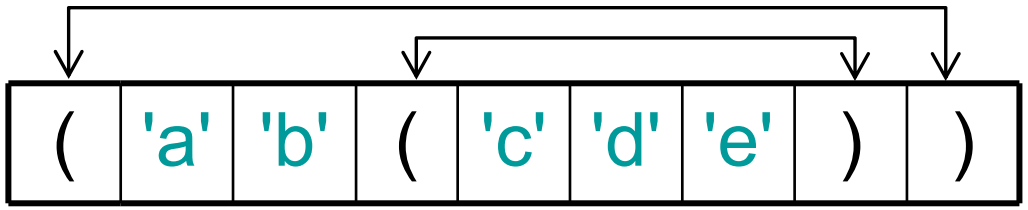
Структура узла

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

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

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

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

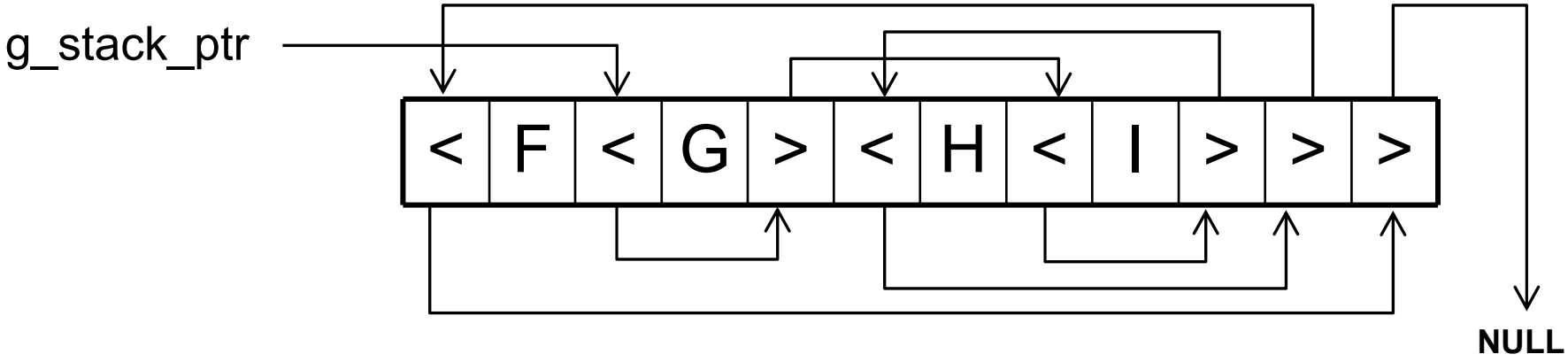
```
typedef struct RefalFunction {
  RefalFunctionPtr ptr;
  const char *name;
} RefalFunction;

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;
    void *file_info;
    RefalSwapHead swap_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

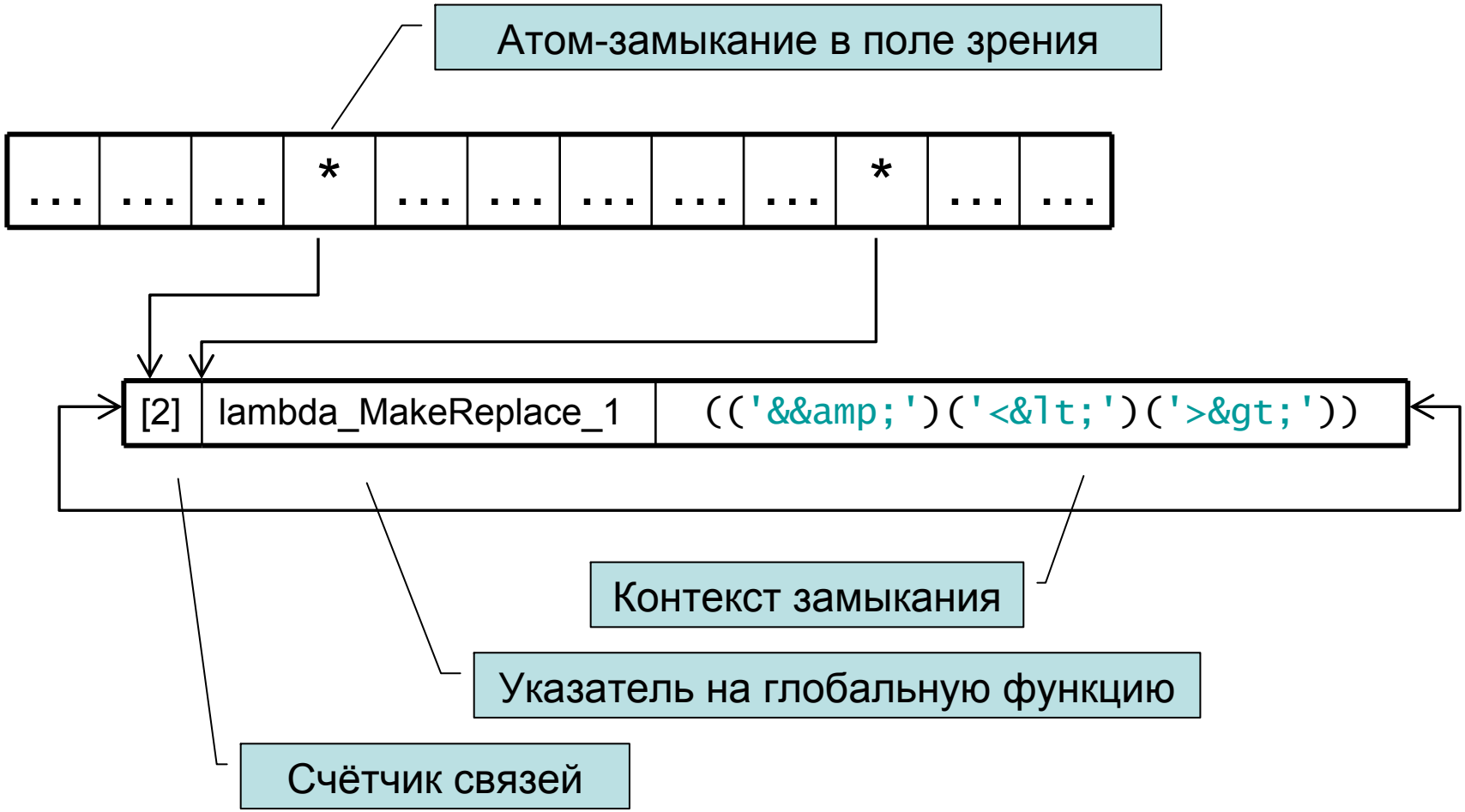
Определение. Контекстом замыкания вложенной функции называется множество переменных, связанных снаружи и используемых внутри функционального блока.

Пример. Контекстом вложенной функции является переменная e.Replaces.

```
MakeReplace {
  e.Replaces =
    { s.In = <Replace s.In e.Replaces>; };
}
```

Вложенные функции неявно преобразуются в глобальные функции и операции связывания с контекстом. (на слайд с примером)

```
$EXTERN ReadLine, WriteLine, Map;

 Replace {
  s.In e.Map-B (s.Symbol e.Out) e.Map-E = e.Out;
  s.Other e.Map = s.Other;
}

lambda_MakeReplace_0 {
  (e.Replaces) s.In = <Replace s.In e.Replaces>;
}

MakeReplace {
  e.Replaces =
    <refalrts::create_closure lambda_MakeReplace_0 (e.Replaces)>;
}

$ENTRY Go {
  =
    <WriteLine
      <Map
        <MakeReplace ('&' '&amp;') ('<' '&lt;') ('>' '&gt;')>
        <ReadLine>
      >
    >;
}
```



Замыкания 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.

§ 41. Генерация кода

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 конкретизации 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 непоследнего 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. *Распознавание образца.* 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  – распознавание образца
    if( распознавание 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;
}
```

Стадия распознавания образца

Левая часть предложения преобразуется в последовательность элементарных команд распознавания по достаточно сложному алгоритму. К элементарным командам распознавания относятся обнаружения с левого или правого конца объектного выражения заданного атома, скобочного терма, нераспознанной переменной, повторной переменной с правого или левого конца объектного выражения.

Особенности распознавания:

- Объектные выражения представляются парой указателей на диапазон узлов поля зрения.
- В начале распознавания мы имеем только пару указателей на диапазон аргумента.
- При успешном распознавании 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 элемента.

Стадия распознавания образца (продолжение)

- Успешное обнаружение атома приводит только смещению одного из указателей диапазона.
- Успешное обнаружение скобочного терма помимо изменения диапазона, приводит к инициализации новой пары указателей диапазона для подвыражения в скобках.
- Успешное обнаружение переменной (s-, t- или повторной) помимо изменения диапазона приводит к инициализации указателя на эту переменную (для e-переменных — пары указателей).
- Обнаружение закрытой e-переменной всегда выполняется успешно. В результате обнаружения пара указателей на e-переменную инициализируется парой указателей диапазона.
- Команда распознавания открытой e-переменной транслируется в цикл.
- При неуспешном обнаружении вне цикла по открытой e-переменной выполняется инструкция ***break;***.

Особенности распознавания открытых e-переменных

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

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

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

Код на Рефале

```
$LABEL A;
$LABEL B;

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

Псевдокод

- B0 → B0 #B
- B0 → (B1) B0
- B1 → B1 #A
- B1 → e.X
- B0 → 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 открыми e-переменными. В начале цикла происходит сохранение состояния вычислений. Вместо инструкции ***break*** используется инструкция ***continue***.
Для наглядности префикс *refalrts::* убран.

Код на Рефале

```
$LABEL A;
$LABEL B;

F {
  e.X #A e.Y = результат;
}
```

Псевдокод

- Cycle (e.X B0)
- B0 → #A B0
- B0 → 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 );
...
```

Стадии распределения памяти и построения результата

В отличии от первой стадии, правая часть преобразуется в последовательность элементарных команд по более простому алгоритму. К элементарным командам распределения памяти относятся команды создания литералов (атомы, скобки) 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 2-й и 3-й стадий:

- 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, распределяемые на 2-й стадии, распределяются в списке свободных узлов, где и хранятся до начала 3-й стадии.

Стадии 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 используемые в образце переменные. 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. Для наглядности префикс *refalrts::* убран. (Сгенерированный код.)

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_1, B0)**
- B0 → #A B0
- B0 → e.Y_1
- n0 ← allocate(#B)
- n1 ← allocate(<)
- n2 ← allocate(Fab)
- n3 ← allocate(>)
- Push(n3)
- Push(n0)
- Build(e.X_1, n0, n1, n2, e.Y_1, n3)
- Free(arg_begin, arg_end)
- **return** cSuccess
- **End of cycle**

- // второе предложение
- B0 → e.X_1
- Build(e.X_1)
- Free(arg_begin, arg_end)
- **return** cSuccess

Пример. Генерация распределения памяти и сборки результата. Для наглядности префикс *refalrts::* убран.

Код на Рефале

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

Псевдокод (начало)

- `// Первое предложение`
- `B0 → 0 B0`
- `B0 → empty`
- `n0 ← allocate(1)`
- `Build(n0)`
- `Free(arg_begin, arg_end)`
- **`return`** `cSuccess`

Псевдокод (продолжение)

- `// Второе предложение`
- $B0 \rightarrow s.Number_1\ B0$
- `B0 → empty`
- $s.Number_2 \leftarrow copy(s.Number_1)$
- `n0 ← allocate(<)`
- `n1 ← allocate(Mul)`
- `n2 ← allocate(<)`
- `n3 ← allocate(Fact)`
- `n4 ← allocate(<)`
- `n5 ← allocate(Dec)`
- `n6 ← allocate(>)`
- `n7 ← allocate(>)`
- `n8 ← allocate(>)`
- `Push(n8)`
- `Push(n0)`
- `Push(n7)`
- `Push(n2)`
- `Push(n6)`
- `Push(n4)`
- $Build(n0, n1, s.Number_1, n2, n3, n4, n5, s.Number_2, n6)$
- `Free(arg_begin, arg_end)`
- **`return`** `cSuccess`

Генерация кода двух последних стадий на C++. Прямая кодогенерация vs. 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 (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 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
```

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

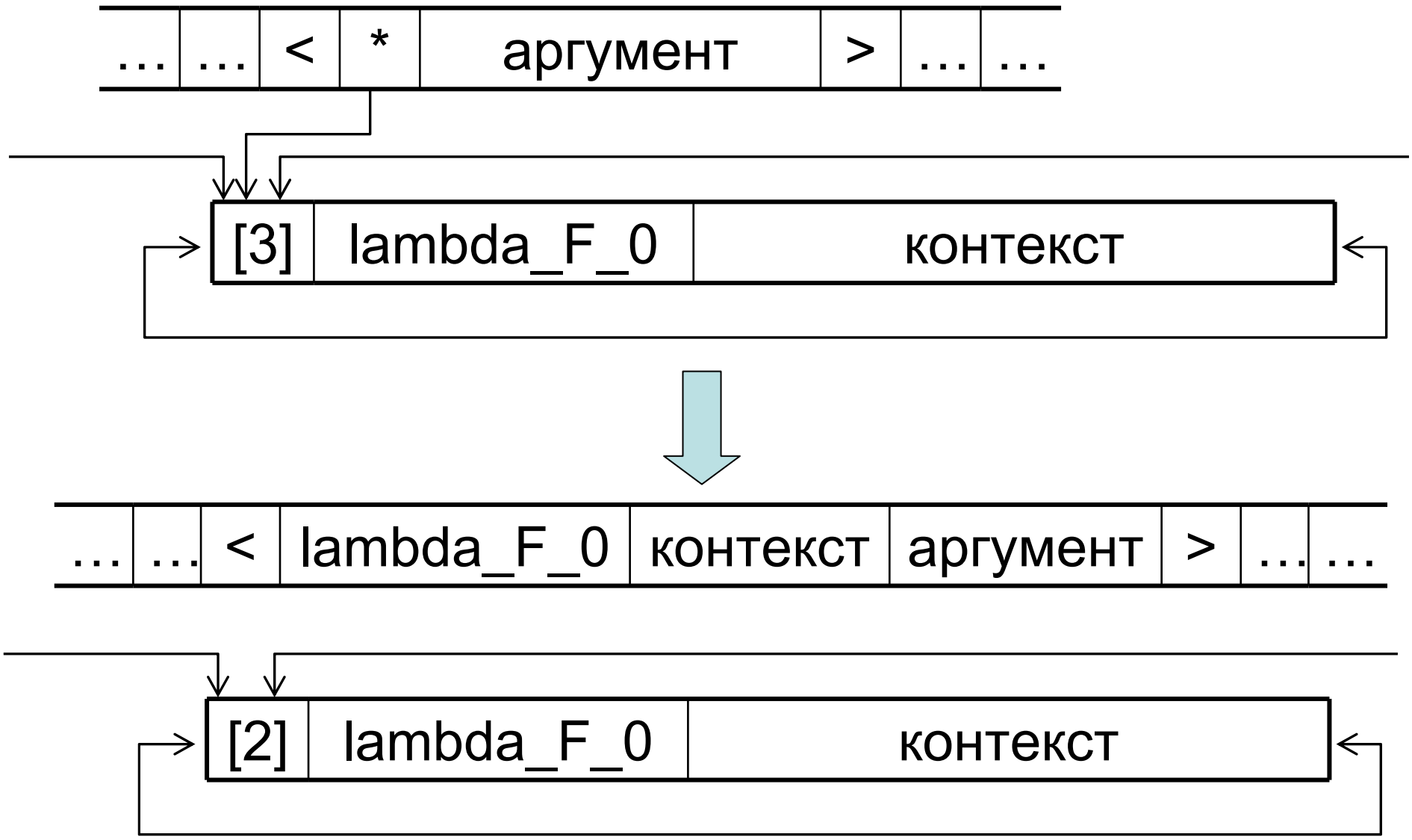
```
#ifdef INTERPRET
      ...
#else
      refalrts::reset_allocator();
      refalrts::Iter res = arg_begin;
      refalrts::Iter n0 = 0;
      if( ! refalrts::alloc_ident( n0, & BL_<int>::name ) ) return refalrts::cNoMemory;
      refalrts::Iter n1 = 0;
      if( ! refalrts::alloc_open_call( n1 ) ) 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
```

§ 42. Поддержка времени выполнения

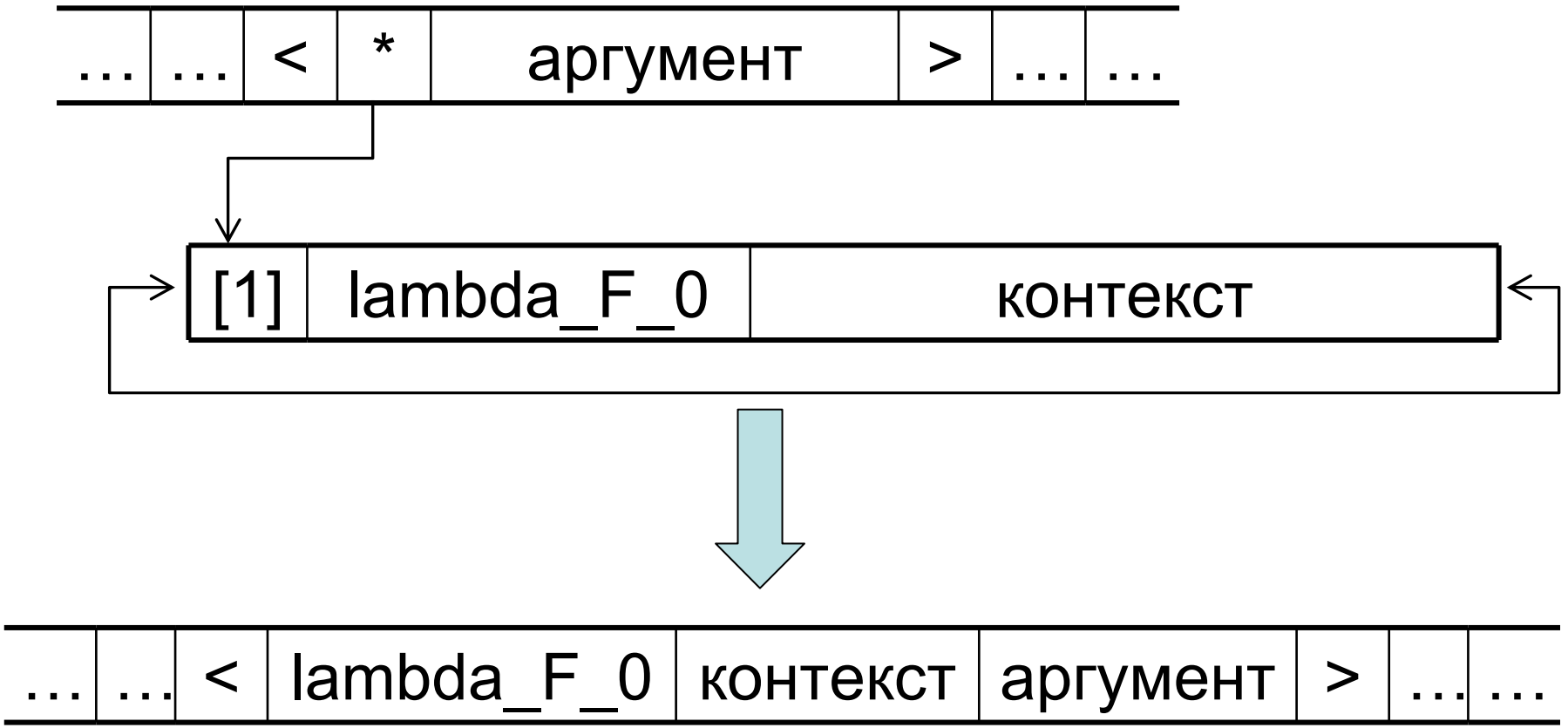
Основной цикл компилятора

```
while( g_stack_ptr != NULL )
{
  (arg_begin, arg_end) = снять со стека две угловые скобки;
  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;
  }
}
```

Вставка содержимого замыкания в поле зрения



Вставка содержимого замыкания в поле зрения (продолжение)



Контрольные вопросы

1. В чём смысл жизни?
2. Почему зимой дни короткие?