Лекция Ы

Компилятор 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: 'a', 'c', 'ы';
  - Целые числа в диапазоне $0…(2^{32} - 1)$: 42, 121;
  - Замыкания — создаются из глобальных функций или безымянных вложенных функций: Fact, Go;
  - Идентификаторы: #True, #Success;
- Составные термы:
  - Структурные скобки: (#Found e.Info);
  - Именованные скобки (АТД): [SymTable e.SymTable].

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. Модуль с именованными скобками и идентификаторами

```
// «Пустая» функция – «тег» для именованной скобки
$ENUM TSymTable; // Эквивалентно TSymTable { }

// Объявления используемых идентификаторов
$LABEL Success, Fails;

// Конструктор
$ENTRY ST-Create {
  = [TSymTable];
}

//  Поиск имени в таблице символов
$ENTRY ST-Lookup {
  [TSymTable e.Names-B (e.Name (e.Value)) e.Names-E] e.Name =
    # Success e.Value;

  [TSymTable e.Names] e.Name = # Fails;
}

// Добавление имени в таблицу символов
$ENTRY ST-Append {
  [TSymTable e.Names-B (e.Name (e.OldValue)) e.Names-E]
  (e.Name) e.NewValue =
    [TSymTable e.Names-E (e.Name (e.NewValue)) e.Names-E];

  [TSymTable e.Names] (e.Name) e.Value =
    [TSymTable e.Names (e.Name (e.Value))];
}
```

Пример. Программа со вложенными функциями.
Эта программа в начале каждой строки файла пишет имя файла и номер строки. Список файлов берётся из командной строки. (реализация)

```
// Библиотечные функции
$EXTERN ArgList, LoadFile, SaveFile, MapReduce, Map, Inc;

// Точка входа в программу
$ENTRY Go {
  =
    <Map
      {
        (e.FileName) =
          <{ s.Number e.Lines = <SaveFile (e.FileName '.out') e.Lines>; }
            <MapReduce
              {
                s.Number (e.Line) =
                  <Inc s.Number>
                  (e.FileName ':' s.Number ':' e.Line);
              }
              1 <LoadFile e.FileName>
            >
          >;
      }
      // Пропуск первого аргумента – имени программы
      <{ (e.ProgName) e.Args = e.Args; } <ArgList>>
    >;
}
```

Абстрактная рефал-машина

Определение. *Рефал-машиной* называется абстрактное устройство, которое выполняет программы на Рефале.
Определение. *Активным выражением* называется выражение, содержащее скобки конкретизации, но при этом не содержащее переменных.
Определение. Активное выражение, обрабатываемое рефал-машиной, называется *полем зрения*.
Работа рефал-машины осуществляется в пошаговом режиме. За один шаг рефал-машина находит в поле зрения *ведущий активный терм* (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 активное 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.

Пример. Выполнение программы, вычисляющей факториал.

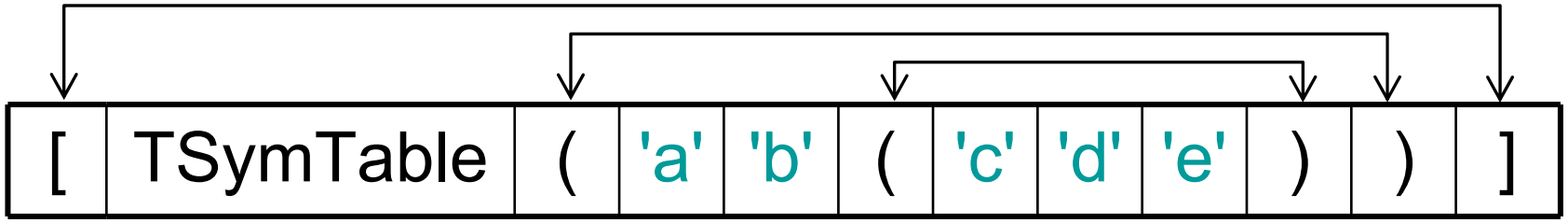
```
<Go>
<WriteLine '6! = ' <Fact 6>>
<WriteLine '6! = ' <Mul 6 <Fact <Dec 6>>>>
<WriteLine '6! = ' <Mul 6 <Fact 5>>>
<WriteLine '6! = ' <Mul 6 <Mul 5 <Fact <Dec 5>>>>>
<WriteLine '6! = ' <Mul 6 <Mul 5 <Fact 4>>>>
<WriteLine '6! = ' <Mul 6 <Mul 5 <Mul 4 <Fact <Dec 4>>>>>>
<WriteLine '6! = ' <Mul 6 <Mul 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.
```

§A. Структуры данных Простого Рефала

- Поле зрения представляется в виде двусвязного списка.
- Узлы списка содержат тег типа (type) 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 или именованную 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.

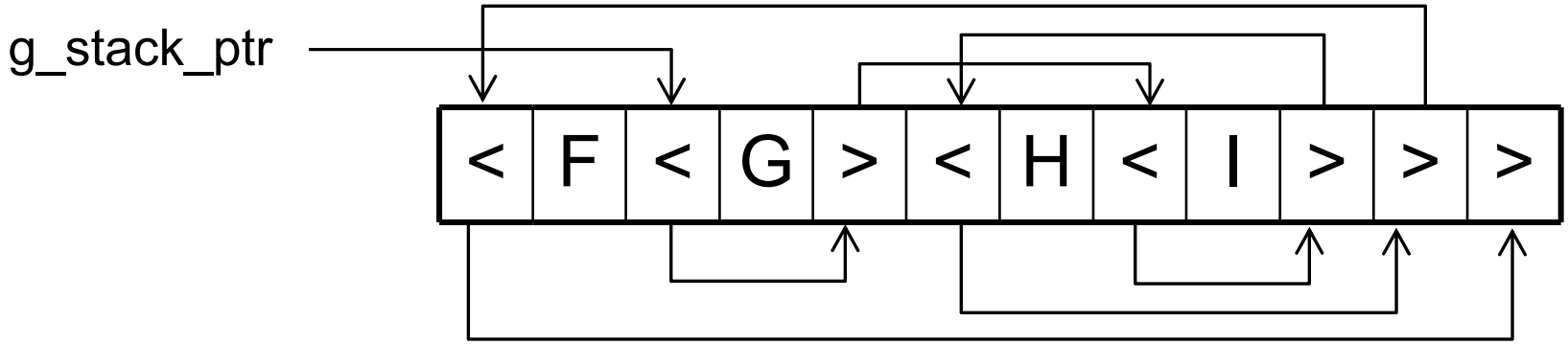
Представление структурных и именованных скобок

Корректность именованных скобок (наличие имени после []) гарантируется синтаксическим анализом, их представление в памяти отличается от структурных только тегом типа.



Представление угловых скобок

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



Представление замыканий с контекстом

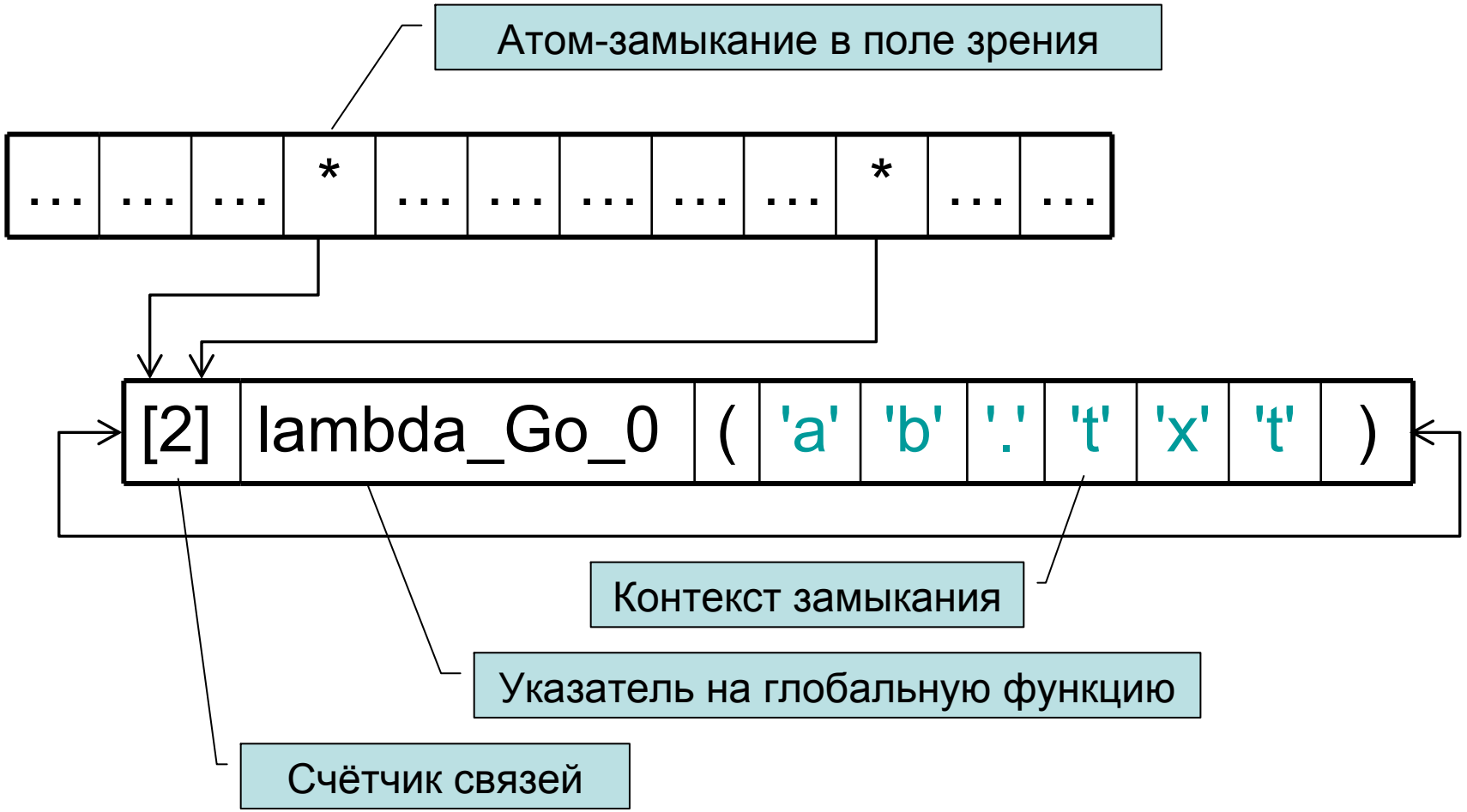
Определение. Контекстом замыкания называется множество свободных переменных внутри функционального блока. Глобальные функции контекста не имеют, у вложенных функций контекст как правило присутствует.

Пример. Контекстом внутренней вложенной функции является переменная e.FileName.

```
<Map
  {
    (e.FileName) =
      ...
      {
        s.Number (e.Line) =
          <Inc s.Number>
          (e.FileName ':' s.Number ':' e.Line);
      }
      ...
  }
  ...
>
```

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

```
$EXTERN ArgList, LoadFile, SaveFile, MapReduce, Map, Inc;

lambda_Go_0 {
  (e.FileName) s.Counter e.Lines =
    <SaveFile (e.FileName '.out') e.Lines>;
}

lambda_Go_1 {
  (e.FileName) s.Number (e.Line) =
    <Inc s.Number> (e.FileName ':' s.Number ':' e.Line);
}

lambda_Go_2  {
  (e.FileName) =
    <<refalrts::create_closure lambda_Go_0 (e.FileName)>
      <MapReduce
        <refalrts::create_closure lambda_Go_1 (e.FileName)>
        1 <LoadFile e.FileName>
      >
    >;
}

lambda_Go_3 { (e.ProgName) e.Args = e.Args; }

$ENTRY Go {
  = <Map lambda_Go_2 <lambda_Go_3 <ArgList>>>;
}
```



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.

§Б. Генерация кода

Крупные конструкции (объявления, отдельные предложения функций) компилируются непосредственно в C++.

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
```

Предложение как чёрный ящик

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.

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 (см. далее) 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;
}
```

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
- B1 → 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 );
...
```

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

Код на Рефале

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

  e.X = e.X;
}
```

Псевдокод

```
// первое предложение
Cycle (e.X1, B0)
  B0 → #A B0
  B0 → e.Y1
  n0 ← allocate(#B)
  n1 ← allocate(<)
  n2 ← allocate(Fab)
  n3 ← allocate(>)
  Push(n3)
  Push(n0)
  Build(e.X1, n0, n1, n2, e.Y1, n3)
  Free(arg_begin, arg_end)
  return cSuccess
End of cycle

// второе предложение
B0 → e.X1
Build(e.X1)
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`