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 (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 'bbrbcbdbbrb' <Fab '!!!'>>
<WriteLine 'bbrbcbdbbrb!!!'>
// Здесь происходит вывод на экран 'abrbcbdbbrb!!!',
// 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 <CartProd ('a' 'b' 'c') (1 2)>
//   == ('a' 1) ('a' 2)  ('b' 1) ('b' 2)  ('c' 1) ('c' 2)
$ENTRY CartProd {
  (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.

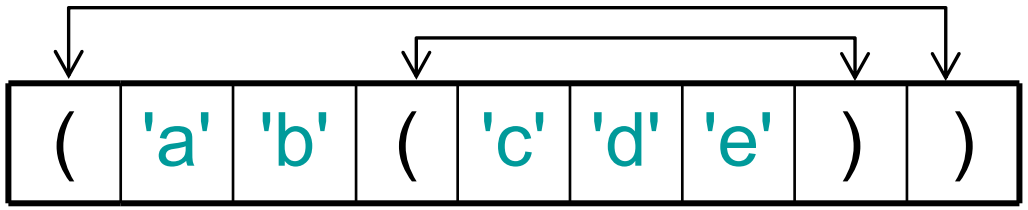
Структура узла

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
  );
```

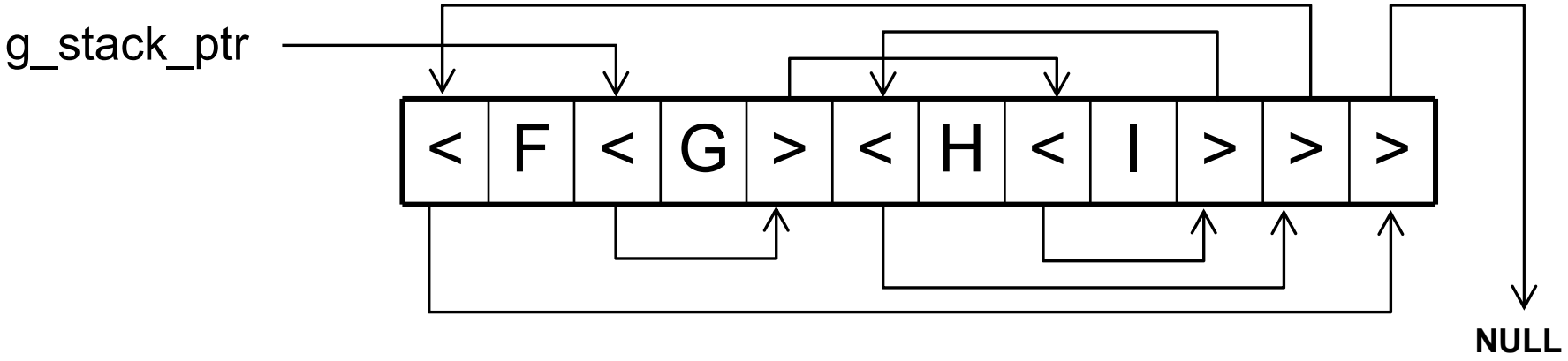
```
typedef struct RefalFunction {
  RefalFunctionPtr ptr;
  const char *name;
} RefalFunction;

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

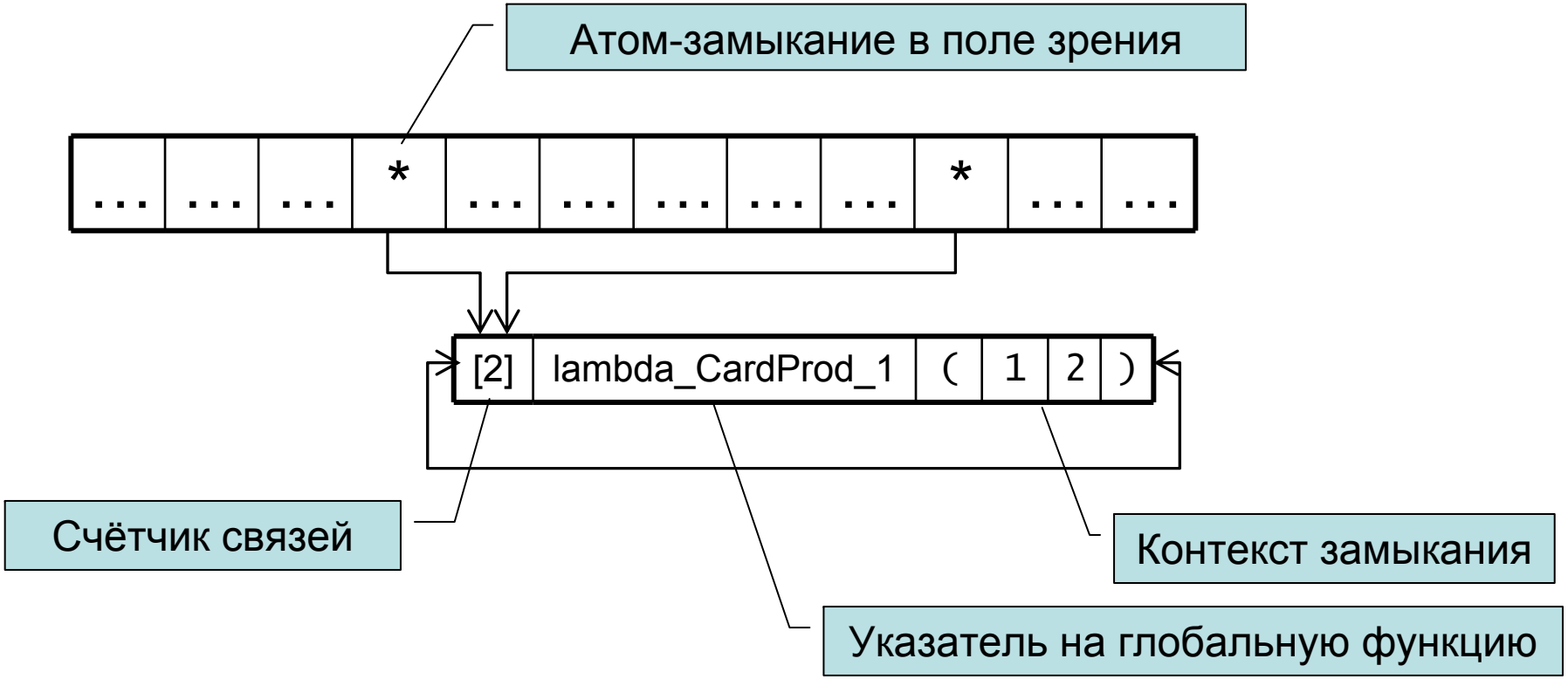
Определение. Контекстом замыкания вложенной функции называется множество переменных, связанных снаружи и используемых внутри функционального блока.

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 инстанцирования 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;
}
```