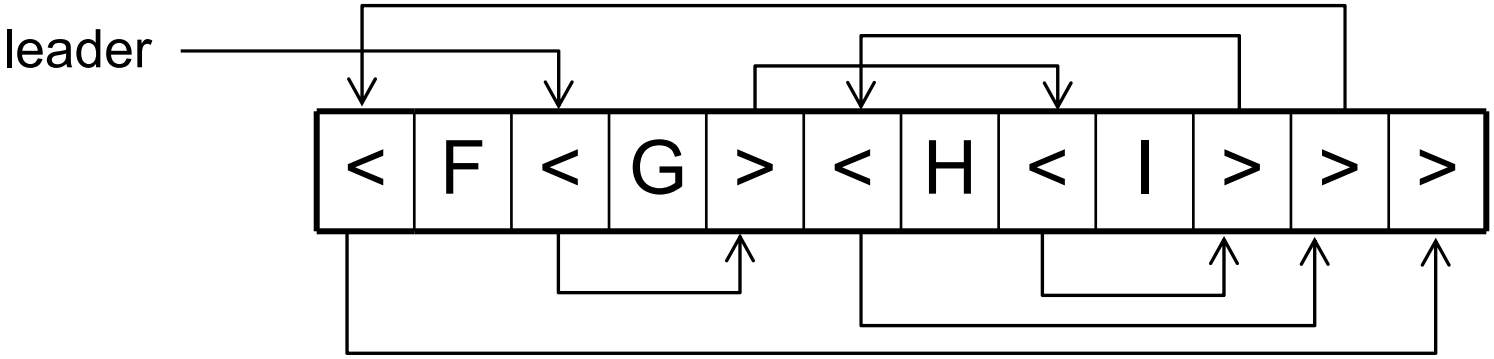
Лекция Ы

Компилятор Простого Рефала

Структура поля зрения и псевдокод основного цикла



Выполнение предложений

Рассмотрим пример:

```
Fib {
  0 = 1;
  1 = 1;
  s.N = <Add <Fib <Sub s.N 1>> <Fib <Sub s.N 2>>>;
}
```

Эта функция вычисляет n-е число Фибоначчи. Проанализируем её с точки зрения списковой реализации.

1. Сопоставление с образцом может завершиться как успешно, так и неуспешно. В первом случае должны быть сохранены указатели на переменные, которые затем будут использоваться при построении результатного выражения. Во втором случае управление передаётся на следующее предложение, при этом аргумент не должен изменяться.
2. В результатной части предложения могут находиться термы, отсутствующие в образцовой части. Для них необходимо выделить память.
3. Часть содержимого скобок конкретизации может оказаться ненужной — её необходимо удалить из поля зрения.

Выполнение предложений (продолжение)

Поэтому выполнение предложения делится на 3 стадии:

1. Стадия сопоставления с образцом. На этой стадии аргумент функции не меняется. В случае успешного сопоставления выполняется вторая стадия, в противном случае управление передаётся на начало предложения следующей функции. В случае последнего предложения происходит возврат из функции кода refalrts::cRecognitionImpossible.
2. Стадия распределения памяти для новых элементов поля зрения. На этом этапе внутри списка свободных блоков формируются новые узлы списка, а также копируются переменные, входящие в результат большее число раз, чем в образец. Аргумент функции при этом не меняется. При отсутствии памяти функция возвращает код refalrts::cNoMemory.
3. На этой стадии формируется результат возврата из функции, скобки конкретизации связываются в стек вызовов. Все эти списковые операции всегда завершаются успешно. На заключительном этапе функция возвращает код refalrts::cSuccess, означающий, что функция успешно завершилась.

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;
}
```

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) =
          <SaveFile
            (e.FileName '.out')
            <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 результат выполнения замыкания (активное выражение) 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.

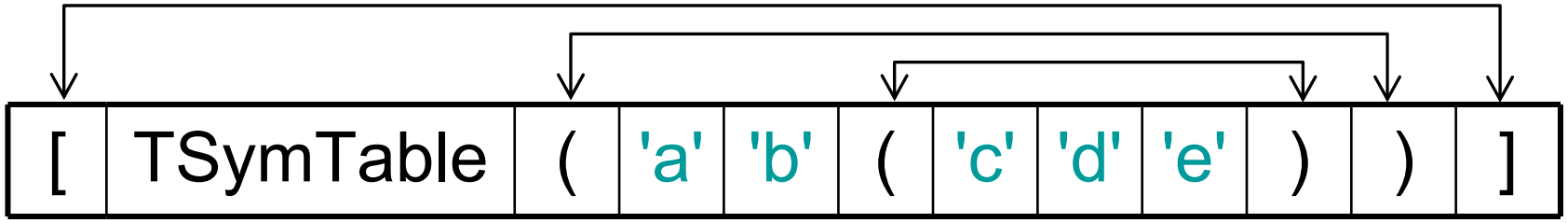
```
<Go>
<WriteLine '6! = ' <Fact 6>>
<WriteLine '6! = ' <Mul 6 <Fact <Dec 6>>>>
<WriteLine '6! = ' <Mul 6 <Fact 5>>>
<WriteLine '6! = ' <Mul 6 <Mul 5 <Fact <Dec 5>>>>>
<WriteLine '6! = ' <Mul 6 <Mul 5 <Fact 4>>>>
<WriteLine '6! = ' <Mul 6 <Mul 5 <Mul 4 <Fact <Dec 4>>>>>>
<WriteLine '6! = ' <Mul 6 <Mul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

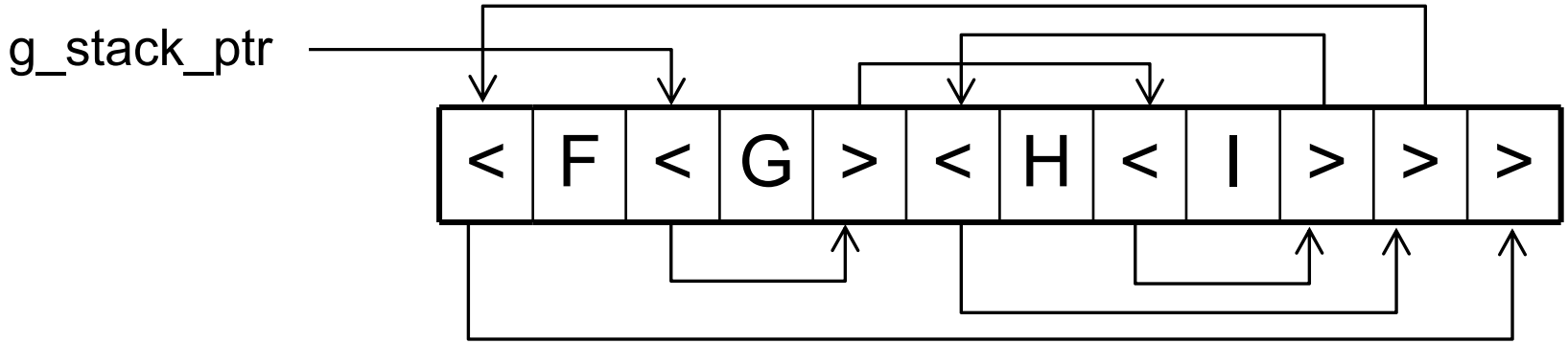
Представление структурных и именованных скобок

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)

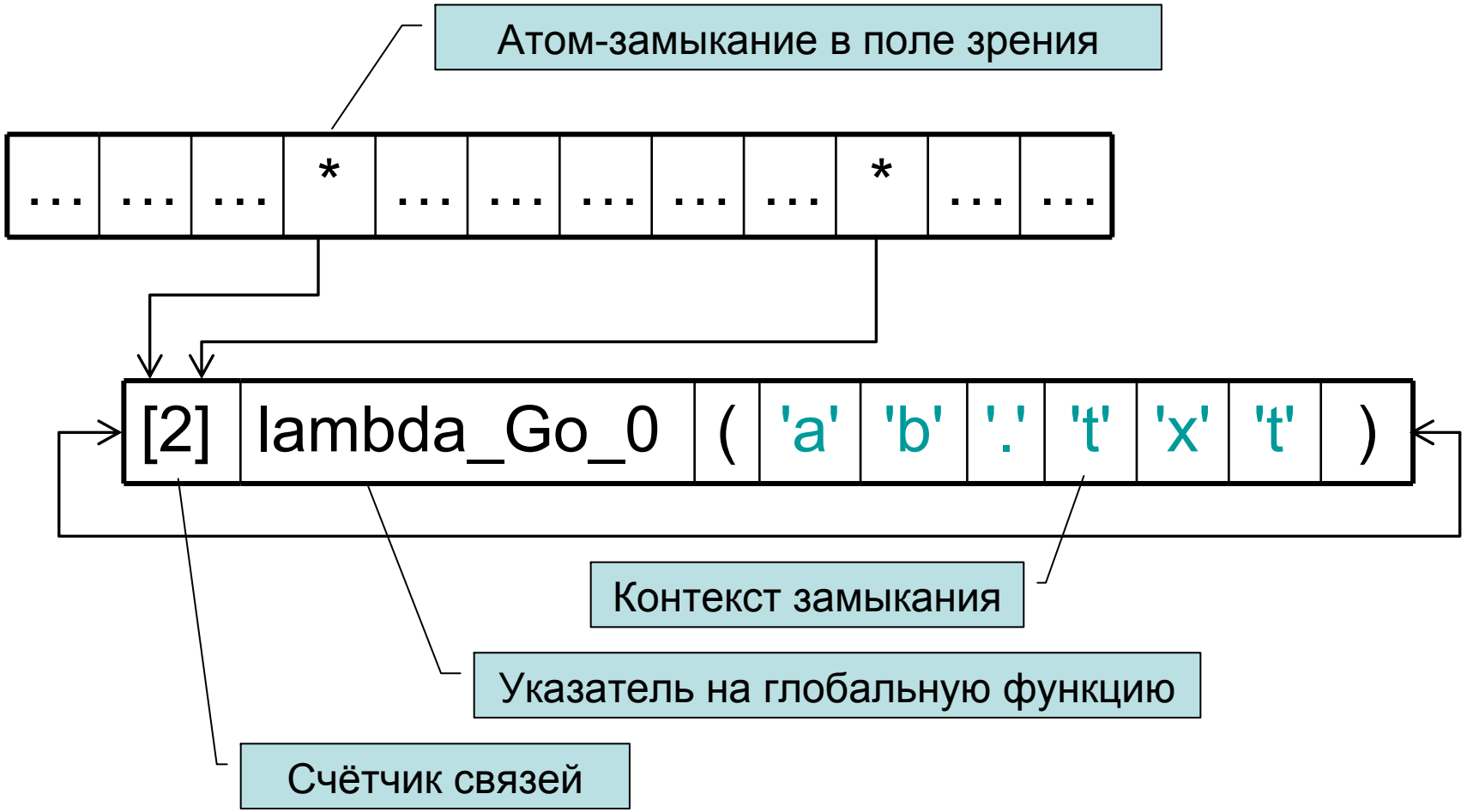
```
$EXTERN ArgList, LoadFile, SaveFile, MapReduce, Map, Inc;

lambda_Go_0 {
  (e.FileName) s.Number (e.Line) =
    <Inc s.Number> (e.FileName ':' s.Number ':' e.Line);
}

lambda_Go_1  {
  (e.FileName) =
    <SaveFile
      (e.FileName '.out')
      <MapReduce
        <refalrts::bind_left lambda_Go_0 (e.FileName)>
        1 <LoadFile e.FileName>
      >
    >;
}

lambda_Go_2 { (e.ProgName) e.Args = e.Args; }

$ENTRY Go {
  = <Map lambda_Go_1 <lambda_Go_2 <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
```