Содержание

Введение

В рамках подготовки к работе над Модульным Рефалом А.В. Коноваловым был разработан язык, являющийся упрощённым подмножеством диалектов языка РЕФАЛ, названный Простым Рефалом. Для него был успешно реализован однопроходной компилятор, порождающий код на платформу императивного языка C++98. В последствии, результат работы над Простым Рефалом нашел применение в качестве одного из возможных back-end интерфейсов для Модульного Рефала и занял место в учебном процессе в рамках курса «Теория конструирования компиляторов» на кафедре ИУ-9 МГТУ им. Н.Э. Баумана.

В развитии компилятора принимали участие и студенты кафедры ИУ-9, выполняющие в виде курсовых работ поддержку и улучшение компилятора. Одним из таких улучшений является курсовая студента Сухарева Вадима [1] , в которой было предпринята оптимизация построения результатных выражений путём замены прямой кодогенерации на интерпретацию. Результатом работы был самоприменимый компилятор Простого Рефала, поддерживающий режим интерпретации, однако, только для результатных выражений.

В данной работе будет рассмотрена задача реализации замены прямой кодогенерации на интерпретацию для образцовой части предложений.

1 Краткий обзор и постановка задачи

1.1 Язык Простого Рефала

Прежде чем говорить об языке Простого Рефала рассмотрим некоторые общие положения «базисного рефала»[2], который можно рассматривать как его основу.

Язык рефал является одномерным знаковым языком. Его объекты — последовательности минимальных нерасчленимых синтаксических единиц — атомарных символов (здесь и далее, атомарные символы будут называться более коротко — атомами). Атомами могут быть:

- Символы (байты, ASCII-символы);
- Составные символы (то есть, имена, определённые в тексте программы) или, просто идентификаторы;
- Целые числа (десятичная запись без знаковых целых числе, обычно, в диапазоне $0 .. 2^{32} - 1$).

Составными термами являются последовательности символов, заключенные либо в структурные (круглые) скобки, либо в функциональные (угловые).

Исходный текст программы на языке рефал представляет собой набор определений программных сущностей. Для базисного рефала — программной сущностью является функция. Пример определения и вызова функции приведён в листинге 1. При этом, в единице трансляции не может быть определено более одной программной сущности с одним и тем же именем. Если тело функции не содержит предложений, то вызов такой функции приводит к аварийному останову.

Введем понятие выражения, как последовательности атомов и скобок, соответствующих следующим правилам:

- Пустое объект есть выражение;
- Атом есть выражение;
- Последовательность выражений есть выражение;
- Выражение в функциональных или структурных скобках есть выражение;
- Остальные объекты, к которым не применимы правила выше не являются выражениями.

Листинг 1 Пример определения и вызова функции на языке базисного рефала.

```
/* Определение функции WriteLine */
WriteLine {
  образец = результат;
  ...
}

/* Вызов функции WriteLine с аргументом 'Hello, World!' */
<WriteLine 'Hello, World'>
```

Легко заметить, что функции в языке рефал оперируют с выражениями и возвращают их же.

Рассмотрим структуру предложений на языке рефал. Предложения состоят из двух частей — образца и результата. Обе части состоят из выражений и разделены между собой символом '='. Предложение обязано оканчиваться символом ';'. Пример предложения: `'a' s.T 'c'= <WriteLine s.T>;`.

Переменные бывают трёх типов, соответствуя синтаксическому типу тех объектов, которые могут быть их значениями. Переменная обозначается парой знаков — признака переменной и произвольного объектного знака. Для изображения признака выражения используется символ — e, для признака терма — t, для признака символа — s. Пример переменных: eA, sT, tP.

Рассмотрим принимаемые значения переменных для соответствующих признаков:

- e-переменные могут принимать любое выражение;
- t-переменные могут принимать любой терм, например, P, (PQ);
- s-переменные могут принимать только атомы.

Выполнением алгоритма, представленного на языке рефал, занимается специальное вычислительное устройство — рефал-машина. В общем представлении, рефал-машина это устройство пошагового выполнения, состоящие из двух потенциально бесконечных запоминающих устройств: привычной памяти и поля зрения. Полем зрения называют область памяти, содержащее выражение, требующее конкретизации. Во многих реализациях рефал-машин принято использовать реализацию поля зрения на основе двусвязного списка узлов, описывающих либо атом, либо скобку.

Теперь перейдем к рассмотрению особенностей языка Простого Рефала и его отличий от базисного.

Во-первых, в Простом Рефале помимо функций в качестве программных сущностей также используются идентификаторы. Исходный текст программ расширяется до набора

объявлений и определений программных сущностей. У программных сущностей появился тип доступа — локальный и глобальный. Локальные функции имеют область видимости в рамках файла, в котором они объявлены. Глобальные же функции (они же entry-функции) могут быть видны в любом файле, при условии добавления её в область видимости файла директивой $EXTERN. Такие функции должны иметь уникальные имена для всей программы.

Идентификаторы в Простом Рефале состоят из последовательности латинских букв, цифр и знаков '-', '_', начинающихся с заглавной латинской буквы. Переменные, в отличии от базисного рефала, уже могут обозначаться минимум двумя знаками. Первым знаком всегда идёт признак переменной, а далее, идёт идентификатор, структура которого соответствует описанному чуть ранее принципу. Для идентификаторов понятия типа доступа отсутствует. Идентификатор объявляется следующим образом `$LABEL Идентификатор1, ... , ИдентификаторN;`.

Ещё одним существенным отличием Простого Рефала является необходимость объявления или определения функциональной сущности до её применения. Для этого предусмотрены директивы $FORWARD и $EXTERN, для локальных и глобальных функций соответственно.

Для сохранения некоторого состояния текущего поля зрения существует понятие статических ящиков, которые помещают перечисленные имена в пространство имён как определённые имена локальных (директива $SWAP) или entry-функций (директива $ESWAP). Статические ящики представляют собой функции с состоянием. При вызове аргумент своего предыдущего вызова, при первом вызове возвращает пустое выражение. В некотором роде статический ящик можно рассматривать в качестве механизма реализации глобальной переменной в программе. Для чтения значения из статического ящика необходимо сначала вызвать ящик с любым аргументом, запомнить то, что он возвратил, снова вызвать с тем, что он возвратил за первый вызов и отбросить результат второго вызова — первый вызов извлекает хранимое значение, второй его восстанавливает.

Ещё одним преимуществом Простого Рефала является наличие вложенных функций. Пример использования вложенной функции приведён в листинге 2.

Более подробно об особенностях языка Простого Рефала, его отличии от таких диалектов, как Рефал-5, Рефал-7 и познакомиться со стандартной библиотекой можно на ресурсах [3, 4].

Листинг 2 Пример вложенной функции на языке Простого Рефала.

```
/* Определение функции без вложенной функции */
DeNumerate {
        (s.Num e.Value) = e.Value;
}
DoSomething {
  e.NumeratedSet = <Map DeNumerate e.NumiratedSet>;
}
/* Определение функции со вложенной функции */
DoSomething {
  e.NumSet = <Map { (s.Num e.Value) = e.Value; } e.NumSet>;
}
```

1.2 RASL

В различных реализациях Рефала достаточно часто встречается понятие промежуточного языка сборки, названного RASL (Refal ASsembly Language). По своей сути, RASL представляет собой систему команд для языка рефал, описанную в символическом виде с использованием мнемонических названий операций, меток и т. п. Предполагается, что язык сборки должен реализовываться посредством интерпретации. Хотя, изначально язык сборки предполагалось не интерпретировать, а компилировать [2]. К непосредственной реализации языка сборки у каждого подмножества диалекта языка Рефал применялись разные подходы. Рассмотрим некоторые из них на примере таких диалектов, как Рефал-2, Рефал-5 и Рефал-6.

1.2.1 RASL Рефал-2

Исходный текст в Рефал-2 компилируется в язык сборки, который встраивается в объектный файл. Затем, полученные объектные файлы со сборками с помощью компоновщика соединяются с файлами среды выполнения компилятора Рефал-2 и внешними функциями языка С. При этом, для обеспечения правильной работы интерпретатора необходимо использовать при компиляции С-функций ключ выравнивания указателей и целых на границу байта для непрерывного размещения в памяти.

Функции в языке сборки описываются в виде структуры:

- имя функции в формате записи ASCII;
- длина имени функции (1 байт);
- коды операций (диапазон принимаемых значений маркирующего байта команды - 0..81 ($0_8..121_8$)).

Для работы со внешней функцией на Си порождается специальный адаптер:

- имя функции в формате записи ASCII;
- длина имени функции (1 байт);
- код операции вызова С-функции (код = 82 (122_8), размер — 1 байт);
- указатель на тело функции внешней Си-функции.

При этом, для выполнения требований выравнивания указателя функции на чётность или четырем байтам (PDP-11 и VAX-11 машины, соответственно) к имени добавлялись в конец выравнивающие байты, содержащие символ пробела ('\032'). При считывании имени они не учитывались и отбрасывались. Также, при вычислении длины имени функции выравнивающие элементы не включались. На листинге 3 приведён пример сгенерированого представления внешней функции для последующей линковки со средой выполнения.

Листинг 3 Пример генерации адаптера внешней функции и её описания для RASL языка Рефал-2.

```
static cproc_ (refpt)
  REFAL *refpt;
{
  <тело функции>
}
static char cproc_0[] = {'c','p','r','o','c','\005'};
char cproc = '\122';
static int (*cproc_1)() = cproc_;
```

1.2.2 RASL Рефал-5

В Рефал-5 язык сборки используется в качестве двоичного представления в файлах с расширением .rsl. Данное представление считывается в память и далее подается интерпретатору в виде потока мнемонических команд. Такое представление имеет очень важное преимущество — переносимость сгенерированных сборок. Отличным примером может служить результат переноса реализации Рефал-5 на операционную систему Windows Mobile 5.0 [6]. Все rsl-сборки для языка Рефал-5 могут быть запущены в данной среде.

Кратко рассмотрим структуру представления, а более подробно ознакомиться с представлением формата RASL для языка Рефал-5 и полным перечнем команд, аргументов и их описанием можно в соответствующем курсовом проекте [5].

Структура .rsl-файла состоит из заголовка, служебной информации и байт-кода. Порядок байт — little-endian.

Представление заголовка:

- имя модуля в формате записи строки ASCIIZ;
- размер секции кода (4 байта);
- число ENTRY-функций в модуле (4 байта);
- число (Extern count) ссылок на внешние функции (4 байта);
- число использованных идентификаторов (4 байта);
- число функций, определённых в модуле (4 байта).

Секция служебной информации хранит в себе данные в следующем порядке:

- список ENTRY-функций;
- список ссылок на внешние функции в формате записи строки ASCIIZ;
- список идентификаторов в формате записи строки ASCIIZ;
- список смещений функций в модуле.

Entry-функции при этом описываются в виде пары значений — именем функции в формате записи строки ASCIIZ и её смещением в коде.

1.2.3 RASL Рефал-6

В реализации диалекта Рефал-6 RASL называется R-кодом и выполняет его R-машина, или по другому — интерпретатор R-кода. Скомпилированные сборки представляют собой сериализованное промежуточное представление в текстовой форме и имеют расширение .rex. Вместе с компилятором предоставляется API для применения сериализации / десериализации данного типа. В данной работе подробно не будет описано преобразование и формат R-кода, лишь приведён пример программы (см. листинг 4), вычисляющей факториал (верхний блок) и её R-код представление (нижний блок).

Каждый модуль начинается с ключевого слова START, за которым следует имя модуля с приписанным расширением «.rex». Затем, тело модуля заключается теги `<BlockBegin>` и `<BlockEnd>`. В атрибутах тега `<BlockBegin>` описываются имена всех сущностей, которые определяются в данном модуле, или же должны быть подгружены из других. Внутри описанных тегов располагаются последовательно определения функций в виде `*{ИмяФункции}=F{Длина R-кода}({Последовательность команд})`.

Подробную информацию можно найти в [7].

Листинг 4 Пример программы вычисления факториала на языке Refal-6 и полученный R-код.

```
$ENTRY FACT e1 = <Fact e1>;
Fact {
  0 = 1;
  s1 = <MUL s1 <Fact <SUB s1 1>>>;
};
```

```
START "fact.rex"
<BlockBegin FACT FACT1 SUB MUL ADD DIV>
*FACT=F7(K P k *Fact dD lB p)
*Fact=F24(u 7 C 0 P i 1 p K P k *MULdD
          k *Fact k *SUB dD i 1 lB lB lB p)
<BlockEnd>
```

1.2.4 Простой Рефал и его RASL

В реализации компилятора Простого Рефала на момент начала данной работы уже существовал ограниченный интерпретатор и генератор псевдокода, по которому во время выполнения строились результаты. Напомним, что режим интерпретации был доступен только для вычисления результатной части предложения. Образцовая часть предложения генерировалась в виде последовательности условий, объявлений и вызовов соответствующих функций среды выполнения компилятора. Сами же команды языка сборки представлялись в виде структуры ResultAction (см. листинг 5). Интерпретируемое тело функции транслируется в виде инициализации массива структуры ResultAction и его заполнения соответствующими операциями на языке C++. Переключение между режимом интерпретации и компиляции должно задаваться директивой условной компиляции Си-кода INTERPRET. Во время компиляции исходного текста рефал-программы и происходит генерация для линковки либо си-кода с включенными оттранслированными представлениями языка сборки, либо без них.

Рассмотрим организацию языка сборки. Поле cmd представляет собой код команды интерпретатора. Возможные значения и описание команд будет приведены ниже по тексту. Поля ptr_value1, ptr_value2 используются для передачи таких аргументов, как ссылки на идентификаторы, функции, объекты в поле зрения. Поле value служит для передачи числовых значений, например, символов или чисел.

Листинг 5 ResultAction

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

Рассмотрим подробнее возможные команды языка сборки. Их можно объединить в следующие группы:

- размещение атомов (icChar, icInt, icFunc, icIdent, icString, icBracket);
- операции переноса фрагмента (icSpliceSTVar, icSpliceEVar);
- операции копирования фрагмента (icCopySTVar, icCopyEVar);
- маркер конца тела (icEnd).

Построении результата из переменных образца и элементов осуществляется при помощи операций над двусвязными списками. Набор таких операций не должен нарушать инвариант двусвязного списка (то есть, не должны появляться «висячие» фрагменты и список должен оставаться правильно связан). Поэтому, появляется необходимость в переносе части цепочки из одного места в другое с исключением из исходного местоположения (splicing). Операции icSpliceSTVar и icSpliceEVar осуществляют перенос для s,t-перменных и e-переменных соответственно.

Пример сгенерированного Си-кода для небольшой функции компилятором до внесения изменений можно найти в приложении 1 на с. 24. Легко заметить, что развилка кода с использованием директивы INTERPRET позволяет в удобном виде наблюдать и изучать изменение генерируемого кода. Данный способ разделения на интерпретируемую и компилируемую часть кодогенерации будет оставлен и в улучшенной версии компилятора, написание которого и является результат данной работы.

В работе [1] описано, что порождение команд языка сборки происходит в GeneralizeResult в файле Algorithm.sref и приводится её тело. В дорабатываемой версии в рамках текущей работы данная операция вынесена в GeneralizeResult (см. листинг 6), а функция MakeInterpCommands (преобразование оригинальных команд псевдокода в команды языка сборки) осталась без изменений. Из листинга видно, что добавилась улучшенная обработка открытых e-перменных, вычисляются неиспользуемые переменные, которые не участвуют в генерации кода и после происходит генерация команд операций языка сборки. Также, из строчек 22 и 23 можно заметить, что обработка образцовой части предложения не транслируется в команды языка сборки.

Вызов выполнения команд языка сборки осуществляется через вызов функции interpret_array, в качестве аргументов которой передаётся массив команд RASL'a, указатель на выделенную память для размещение дополнительных данных, указатели на границы поля зрения. Результатом работы функции является один из следующих кодов:

- cSuccess — код успешного построения результата;
- cRecognitionImpossible — код ошибки сопоставления;
- cNoMemory — код ошибки при выделении памяти.

Более подробное описание деталей реализации и примеры кода дорабатываемой в рамках текущей работы версии компиялтора можно найти в работе Сухарева Вадима [1], а также в ветке master проекта на GitHub [4] с историей изменений вплоть до 19 октября 2015 года.

Таким образом, в данной работе необходимо расширить операции языка сборки, добавить их обработку в интерпретатор RASL'a, чтобы интерпретироваться могла и образцовая часть предложения. Также, необходимо произвести актуализацию кода обработки команд и их порождения.

Листинг 6 GenerateResult-Interp Algorithm.sref

```
1  GeneralizeResult {
2    (e.PatternVars) (e.PatternCommands) (e.ResultVars)
3    (e.ResultAllocCommands) (e.ResultCommands) =
4      <Fetch
5        (<ReplicateVars e.PatternVars>) (<ReplicateVars e.ResultVars>)
6        (<Map RepeatedEVariables e.PatternCommands>)
7        <Map ClosedEVariables e.PatternCommands>
8        <Seq
9          {
10           (e.PatternVars^) (e.ResultVars^) (e.RepeatedEs) e.ClosedEs =
11             (<VarSetUnion (e.PatternVars) (e.ResultVars)>)
12             (<VarSetDifference (e.ResultVars) (e.PatternVars)>)
13             <VarSetDifference (e.ClosedEs) (e.RepeatedEs e.ResultVars)
                  >;
14         } {
15          (e.CommonVars) (e.CopiedVars) e.UnusedClosedEs =
16             (<VarSetDifference (e.CommonVars) (e.UnusedClosedEs)>)
17             (e.CopiedVars)
18             <Map (FilterUnusedCmdClosedE e.UnusedClosedEs) e.
                  PatternCommands>;
19         } {
20           (e.CommonVars) (e.CopiedVars) e.PatternCommands^ =
21             <GenerateResult-OpenELoops
22               <Map MakeDeclaration e.CommonVars>
23               e.PatternCommands
24               (CmdIfDef)
25               (CmdInitRAA)
26               <Map
27                 (MakeInterpCommands e.CopiedVars)
28                 <FoldAllocCommands
29                   (e.ResultAllocCommands)
30                   e.ResultCommands>
31               >
32               (CmdFinRAA)
33               (CmdElse)
34               (CmdEmptyResult)
35               <Map MakeCopyVar e.CopiedVars>
36               e.ResultAllocCommands
37               e.ResultCommands
38               (CmdReturnResult)
39               (CmdEndIf)>;
40         }
41       >
42     >;
43 }
```

2 Реализация

При изучении различных реализаций RASL языков диалекта Рефал можно заметить, что большинство реализаций так или иначе позволяют описывать сборки в файловом представлении. В дорабатываемой версии языка в генерируемом коде активно используются переменные типа Iter, которые служат как ссылки на элементы поля зрения, то есть для описания сопоставляемых сущностей. Они часто передаются по ссылке в качестве аргументов для функций среды выполнения или команд интерпретатора. Одним из предлагаемых улучшений в рамках данной работы является переписывание выделения переменных Iter в отдельный массив памяти context и использование в качестве аргумента уже положение ссылки на узел в этом массиве, а не указатель. Также, все объявления скобок аналогично переносятся в массив context. Введём следующее соглашение: в начале массива context первые 2*n элементов выделяется для n скобок, остальное же место предназначено для описания сопоставляемых сущностей.

Будем хранить e-перменные и скобки в виде двух последовательных элементов массива context. Таким образом, по индексу скобки или e-переменной мы получаем описание для её начала, а по индексу + 1 получаем её конец. Код обращения и объявления выделенных переменных и скобок приведён в листинге 8. Она же, оттранслированная на версии компилятора до внесения улучшений, доступна для сравнения в приложении 2 на с. 29. Для повышения читаемости генерируемого кода и его наглядности будем объявлять индексы выделенных переменных по следующему соглашению: `enum { __{тип}{имя}_{номер-вхождения}_{номер-скобки} = {индекс} };`. Тип переменной представляет собой ascii символ 's', 't', 'e' для соответствующего типа рефал данных. Для скобок же сохраним их численное представление. По индексу скобки легко восстановить её номер, ведь $index = num * 2$.

В оригинальной версии компилятора не было предусмотрено нумерации переменных, что усложняет внедрение работы с сущностями по индексу в массиве context. Поэтому, первым шагом к реализации описанной выше модификации является добавление в генератор нумерации выделяемых переменных и скобок. В файле Algorithm.sref в функции GeneralizeResult (которая вкратце рассматривалась на предшествующей странице) происходит вычленение неиспользуемых переменных и генерация сопоставления с образцом, построения результата и команд языка сборки. Воспользуемся тем фактом, что в генерации участвуют уже отфильтрованные сущности. Для них и будем строить индексы. Объект (e.CommonVars) представляет собой список переменных, который содержит не только переменные образцовой части, но и копируемые переменные результатной. В листинге 7 приведён код функции, осуществляющей нумерацию переменных в порядке возрастания. Здесь же, разбирается тип переменной. Все переменные, кроме e-переменных должны занимать 1 слот, а e-переменные — 2.

Нумерация скобок актуализируется достаточно тривиально. В функции DoGenPattern при построении объекта уже присутствует нумерация. Необходимо лишь заменить инкремент номера на увлечение на 2. В силу того, что переменные в массиве располагаются после выделения памяти для описания скобок, то необходимо в качестве начального значения для первого вызова функции NumerateVars передать диапазон на начало области переменных в массиве context, которое вычисляется, как `(число скобок + 1) * 2`.

Листинг 7 Нумерация переменных

```
NumerateVars {
  s.Number ( s.Usings 'e' e.Index ) e.Etc =
    (s.Number s.Usings 'e' e.Index)
    <NumerateVars <Add s.Number 2> e.Etc>;
  s.Number ( s.Usings s.Mode e.Index ) e.Etc =
    (s.Number s.Usings s.Mode e.Index)
    <NumerateVars <Inc s.Number> e.Etc>;
  s.Number = (s.Number);
}
```

Данное решение позволяет получить более читабельный и компактный генерируемый код, а также позволяет сократить размер структуры, описывающей команду языка сборки, о чем будет написано ниже по тексту.

В качестве дополнительной оптимизации А.В. Коноваловым было предложено сокращение размера элементарной команды языка сборки до 4 байт. Предполагается, что команда будет описана структурой, приведённой в листинге 9. Данный подход требует ввода дополнительного ограничения на максимальный индекс переменных и скобок в диапазоне до 255 в рамках одной порождаемой функции. Данное ограничение считается вполне приемлемым. Также, например, в реализации диалекта Рефал-5 введено аналогичное ограничение.

Структура ResultAction была переименована на более актуальное имя RASLCommand, что точнее передаёт её смысл. Поле cmd описывает тип операции, поля val1 и val2 служат для передачи индексов аргументов или некоторых значений. При этом индекс скобки обычно передаётся через поле bracket. Приведённая модификация уменьшает потребление памяти при выполнении в режиме интерпретации, также, позволяет современным процессорам более эффективно обрабатывать данные, так как вероятность попадания в кеш процессора цепочки команд, выровненных по границе в 4 байта достаточно существенна.

Листинг 8 Оптимизация с использованием массива context

```
refalrts::Iter context[13];
refalrts::zeros( context, 13 );
enum { __ePattern_1_1 = 6 };
enum { __eVars_2_1 = 8 };
enum { __sNumRanges_2_1 = 10 };
enum { __eCommands_2_1 = 11 };
context[0] = arg_begin;
context[1] = arg_end;
...
static refalrts::RASLCommand raa[] = {
  {refalrts::icBracketLeft, 0, 2, 0},
  {refalrts::icBracketLeft, 0, 4, 0},
  {refalrts::icContextSet, 0, __ePattern_1_1, 2},
  {refalrts::icContextSet, 0, __eVars_2_1, 4},
  {refalrts::icsVarLeft, 0, __sNumRanges_2_1, 0},
  {refalrts::icContextSet, 0, __eCommands_2_1, 0},
  {refalrts::icEmptyResult, 0, 0, 0},
  ...
  {refalrts::icBracket, 0, refalrts::ibCloseBracket, 0},
  {refalrts::icEnd}
};
...
if( ! refalrts::brackets_left( context[4], context[5], context
   [0], context[1] ) )
 break;
context[__ePattern_1_1] = context[2];
context[__ePattern_1_1 + 1] = context[3];
context[__eVars_2_1] = context[4];
context[__eVars_2_1 + 1] = context[5];
...
```

Листинг 9 Описание структуры RASLCommand

```
typedef struct RASLCommand {
  unsigned char cmd;
  unsigned char val1;
  unsigned char val2;
  unsigned char bracket;
} RASLCommand;
```

Модель интерпретатора, описанная в [1] при реализации новых команд сохранит часть своей кодовой базы. Только существующие операции (icChar, icInt, icFunc, icIdent, icString, icBracket, icSpliceSTVar, icSpliceEVar, icCopySTVar, icCopyEVar, icEnd) изменят свой фор-

мат на более актуальный — четырехбайтный.

Рассмотрим полное множество команд для языка сборки Простого Рефала.

Команды интерпретатора можно разделить на несколько групп по их функциональности. В первую группу можем отнести команды выделения памяти для сущности. В нее входят команды для таких типов данных, как символ (icChar), без знаковое целое число (icInt, icHugeInt), объект типа функция (icFunc), идентификатор (icIdent), строка (icString), скобка (icBracket). Данная группа имеет следующий вид представления в RASL-представления: {код_операции, 0, аргумент, 0}.

Стоит отметить, что при представлении команды в четырёхбайтной структуры передача в прямом виде в качестве аргумента указателя на функцию, идентификатор или значения целого без знакового числа, представление которого превышает размер 1 байт, не представляется возможным при таком подходе. Поэтому, будем передавать в качестве аргумента не значение, а индекс данного элемента в некотором хранилище. Выделим следующие виды хранилищ данных:

- хранилище представлений функций (`const RefalFunction functions[]`);
- хранилище представлений идентификаторов (`const RefalIdentifier labels[]`);
- хранилище чисел (`const RefalNumber numbers[]`).

Таким образом, для команд icHugeInt, icIdent, icFunc в качестве аргумента передаётся положение сущности в соответствующем хранилище. В для остальных команд первой группы аргументом является значение сущности. Рассмотрим подробнее команду выделения скобки icBracket. В качестве её аргумента выступает одно из значений перечисления BracketType, описывающее какой тип скобки должен быть помещён в память и открытая ли или закрытая ли скобка. Скобки представлены следующими значениями:

- ibOpenADT — открытая скобка АТД-терма;
- ibCloseADT — закрытая скобка АТД-терма;
- ibOpenBracket — открытая структурная скобка;
- ibCloseBracket — закрытая структурная скобка;
- ibOpenCall — открытая функциональная скобка;
- ibCloseCall — закрытая функциональная скобка.

Во вторую группу вынесем ряд команд, осуществлявшие перенос или копирование переменных в результатной части предложения (icSpliceSTVar, icSpliceEVar, icCopySTVar,

icCopyEVar). RASL-представление данной группы схоже с первой группой, только в качестве аргумента всегда используется положение сущности в хранилище сущностей context. Представление — `{код_операции, 0, аргумент, 0}`. Благодаря тому, что для e-переменных по их индексу в context можно однозначно определить начало и конец сущности, представление стало более лаконичнее, в отличии от ранее используемого `{код_операции, указатель_начала, указатель_конца, 0}`.

Рассмотрим команды, отвечающие за образцовую часть предложения. Многие команды имеют суффикс Left\Right — обозначающий направление сопоставления. Большая часть команд соответствуют RASL-представлению `{код_операции, 0, аргумент, номер_скобки}`.

Данному описанию соответствуют команды:

- icBracketLeft, icBracketRight — сопоставление со скобкой;
- icsVarLeft, icsVarRight — сопоставление s-переменной;
- ictVarLeft, ictVarRight — сопоставление t-переменной;
- icNumLeft, icNumRight, icHugeNumLeft, icHugeNumRight — сопоставление с целым беззнаковым числом;
- icIdentLeft, icIdenrRight — сопоставление с идентификатором;
- icFuncLeft, icFuncRight — сопоставление с сущностью, описывающую функцию ;
- icCharLeft, icCharRight — сопоставление с символом;
- icSave — присвоение значения одной e-перменной или скобки в другую;
- icEStart — сопоставление открытых e-переменных;
- icEPrepare — подготовка к циклу удлинения открытых e-переменных.

Команды icBracket*, icsVar*, ictVar* в качестве аргумента принимают позицию сопоставляемой сущности в хранилище context. icNum*, icChar* принимает непосредственно значение сущности (целое беззнаковое до 255 или символ). Аналогично с командами построения результата для сущностей, представляющих собой описание функции, идентификатор, целое безнаковое больше 255 в качестве аргумента передается позиция сущности в соответствующем хранилище (functions, labels, numbers соответственно).

Команде icSave более точно соответствует RASL-представление: `{код_операции, 0, цель, источник}`.

Рассмотрим команды сопоставления открытых e-переменных: icEPrepare, icEStart. Сопоставление с открытой e-переменной генерируется в виде двух команд — — подготовка

к удлинению (icEPrepare) и само непосредственное удлинению (icEStart). Подготовка к удлинению инициализирует значение сущности открытой переменной пустым выражением. Затем кладет на стек истории удлинения указатель на следующую команду (команду удлинения) в виде индекса команды в потоке RASL-команд. При этом, следующая команда пропускается на следующем шаге выполнения потока команд языка сборки. В случае, если текущее сопоставление не удалось, то интерпретатором языка сборки происходит проверка стека откатов удлинения. Если стек пуст, значит произошла ошибка сопоставления всей образцовой части предложения. Тогда просто результатом выполнения блока команд для данного предложения является код ошибки `cRecognitionImpossible`. В противном случае, снимаем с вершины стека индекс команды для отката, то есть на команды удлинения, соответствующей текущей сопоставляемой открытой e-переменной. На данном шаге интерпретатор пытается удлинить её. Если удлинение прошло успешно, то сохраняем на стек возвратов положение текущей команды удлинения и проводим выполнение последующих команд. В противоположном исходе, повторяем процедуру проверки стека откатов. Аналогичный способ обработки в языке сборки сопоставления открытых переменных был применён в [2].

Теперь, перейдем к описанию последней группы RASL-команд — сопоставление повторных переменных iceRepeatRight, iceRepeatLeft, icsRepeatRight, ictRepeatRight, icsRepeatLeft, ictRepeatLeft. Им соответствует RASL-представление: `{код_операции, повторная_переменная, оригнальная_переменная, номер_скобки}`.

Данные операции производят сопоставление с повторными переменными, устанавливая копии значений.

3 Тестирование

В тестировании были задействованы несколько машин, на одной из которых также проводились измерения в рамках работы [1]. Характеристики общей машины для курсовой работы 2009 года и данной приведены в таблице 3.1. Системные характеристики остальных тестовых машин приведены в таблицах 3.2, 3.3.

Таблица 3.1: Системные характеристики: ноутбук Pentium MMX

CPU	Intel® Pentium® Processor with MMX™ Technology 233 MHz, 66 MHz FSB, L2 Cache 512 KB
RAM	64 Mb DRAM
OS	Windows XP x86

Таблица 3.2: Системные характеристики тестовой машины №1 для тестирования

CPU	Intel 5-2430M, 2.4 GHz (up to 3.0GHz) 3 MB Smart Cache (L1 — 128KB, L2 — 512KB), 2/4 cores/threads
RAM	2×4 GB, 1333 MHz SODIMM
OS	Windows 10 x86-64

Таблица 3.3: Системные характеристики тестовой машины №2 для тестирования

CPU	Intel i5 3570k, 4.4 GHz 6 MB Smart Cache (L1 — 4x32KB, L2 — 4x256KB), 4/4 cores/threads
RAM	2x8GB, 1866 MHz
OS	OpenSuse 13.2, x86-64 Linux kernel — 3.16.7-29-desktop

Тестирование проводилось на примере самоприменения компилятора Простого Рефала. Рассматривались режимы прямой компиляции исходных текстов и непосредственной генерации и выполнения интерпретируемого кода. Замеры происходили в виде нескольких выборок для каждой конфигурации тестовой машины и используемого компилятора. Исходный текст скриптов, выполняющих измерения приведён в приложении 3 на с. 30.

Рассмотрим результаты тестирования для машины №1. В качестве исходных данных использовались исходные тексты компилятора, по пять прогонов для каждого файла. Также, сделан девятикратный прогон каждого компилятора C++ с файлами в папке bootstrap. Таким образом, можно определить, на сколько быстрее целевой компилятор обрабатывает инициализацию массива кодов операций языка сборки по сравнению с построением в режиме компиляции.

Используемые компиляторы:

- Visual C++ Express 2013, исходные тексты транслировались как на x86 и x86-64 архитектуры;
- Borland C++ Compiler 5.5;
- MinGW builds 4.8.1 rev 5
  - x86 с использованием потоков POSIX, механизм исключений — DWARF;
  - x86-64 с использованием потоков POSIX, механизм исключений — SEH;
- Open Watcom 1.9;
- Clang 3.7.0, исходные тексты транслировались как на x86 и x86-64 архитектуры, библиотеки в сборке MinGW builds 4.8.1 rev 5.

Результаты измерений приведены в таблицах 3.5, 3.4, 3.6. Данные приводятся в формате: общее время выполнения одного прохода, время сопоставления, время построения результата, время выполнения внешних функций, время выполнения рефал-функций. Размерность — секунды. Виды тестирования, приведённые в соответствующей таблице:

1. Среднее время самоприменения в режиме компиляции;
2. Среднее время самоприменения в режиме интерпретации;
3. Среднее время проверки целевого компилятора в режиме компиляции;
4. Среднее время проверки целевого компилятора в режиме интерпретации.

Таблица 3.4: Результаты ноутбука Pentium MMX

	Тестирование	
Конфигурация	*1	*2
MinGW x86	319.175/118.377/ 102.719/221.096/98.079	370.841/145.55/ 132.023/277.573/93.268
Borland C++ 5.5	235.086/79.515/ 86.176/165.691/69.395	258.958/95.062/ 101.382/196.444/62.514

	Тестирование	
Конфигурация	*3	*4
MinGW x86	320.637/0.0/0.0/0.0/320.577	96.859/0.0/0.0/0.0/96.795
Borland C++ 5.5	26.083/0.0/0.0/0.0/26.045	13.672/0.0/0.0/0.0/13.628

Таблица 3.5: Результаты тестовой машины №1

	Тестирование	
Конфигурация	*1	*2
Visual C++ x86	21.344/6.728/6.855/13.583/7.761	26.576/8.806/10.25/19.056/7.52
Visual C++ x86-64	21.032/6.88/7.045/13.925/7.107	25.534/8.156/10.321/18.478/7.056
Borland C++ 5.5	16.618/5.082/6.915/11.996/4.622	18.505/5.929/8.181/14.111/4.394
MinGW x86,	20.44/6.618/6.995/13.613/6.827	25.976/8.817/10.222/19.039/6.937
MinGW x86-64	20.648/6.579/7.54/14.119/6.529	26.45/8.779/11.086/19.865/6.585
Open Watcom	55.16/19.577/27.274/46.851/8.308	58.413/20.856/29.588/50.444/7.969
Clang 3.7.0 x86	23.832/7.941/8.841/16.781/7.05	29.357/10.275/11.903/22.178/7.179
Clang 3.7.0 x86-64	20.06/6.816/6.84/13.656/6.404	25.437/8.495/10.439/18.934/6.504

	Тестирование	
Конфигурация	*3	*4
Visual C++ 2013 x86	1.592/0.0/0.0/0.0/1.601	1.468/0.0/0.0/0.0/1.468
Visual C++ 2013 x86-64	0.821/0.0/0.0/0.0/0.819	1.565/0.0/0.0/0.0/1.565
Borland C++ 5.5	0.821/0.0/0.0/0.0/0.819	0.536/0.0/0.0/0.0/0.533
MinGW x86, DWARF	13.699/0.0/0.0/0.0/13.696	4.746/0.0/0.0/0.0/4.742
MinGW x86-64, SEH	13.472/0.0/0.0/0.0/13.47	4.608/0.0/0.0/0.0/4.602
Open Watcom	13.54/0.0/0.0/0.0/13.531	1.998/0.0/0.0/0.0/1.993
Clang 3.7.0 x86	11.471/0.0/0.0/0.0/11.471	5.619/0.0/0.0/0.0/5.617
Clang 3.7.0 x86-64	11.727/0.0/0.0/0.0/11.727	5.551/0.0/0.0/0.0/5.549

Таблица 3.6: Результаты тестовой машины №2

	Тестирование	
Конфигурация	*1	*2
GCC 4.8.3 x86	16.768/5.061/6.475/13.088/4.798	19.870/6.091/8.359/15.232/5.394
GCC 4.8.3 x86-64	17.123/4.885/6.382/13.952/5.088	20.652/6.128/8.514/16.651/5.425
Clang 3.5.0 x86	18.538/5.671/7.355/13.422/5.581	21.442/6.955/9.716/16.289/6.181
Clang 3.5.0 x86-64	18.665/6.104/7.489/14.109/5.791	22.115/7.475/9.519/17.314/6.560

	Тестирование	
Конфигурация	*3	*4
GCC 4.8.3 x86	9.272/0.0/0.0/0.0/9.320	3.111/0.0/0.0/0.0/3.209
GCC 4.8.3 x86-64	8.892/0.0/0.0/0.0/9.012	2.982/0.0/0.0/0.0/2.997
Clang 3.5.0 x86	10.781/0.0/0.0/0.0/10.892	3.687/0.0/0.0/0.0/3.619
Clang 3.5.0 x86-64	10.976/0.0/0.0/0.0/11.027	3.814/0.0/0.0/0.0/3.756

Необходимо отметить, что в силу ограниченной вычислительной мощности и ресурсов тесты на ноутбуке Pentium MMX проводились с меньшим числом выборки (а именно, 3 выборки для самоприменения и 5 выборок для тестирования целевых компиляторов). Также, в связи с архитектурой процессора (80486) были доступны лишь компиляторы MinGW x86 и Borland C++ 5.5. Из аналогичных соображений, доступный на данной вычислительной машине компилятор C++ Watcom не участвовал в замерах, как зарекомендовавший

себя в замерах на тестовой машине №1 самым медленным компилятором.

На тестовой машине №2 проводилось тестирование на операционной системе Gnu Linux Opensuse 13.2 с выбором в качестве целевых компиляторов GCC 4.8.3 (g++ (SUSE Linux) 4.8.3 20140627 [gcc-4_8-branch revision 212064]) и clang version 3.5.0 (tags/RELEASE_350/final 216961). В качестве целевых платформ использовались платформы x86 и x86-64.

Как видно из результатов тестирования, сравнивать их достаточно сложно, так как не малое влияние оказывает время компиляции целевым компилятором и его выбор. По результатам анализа – самым быстрым компилятором оказался BCC 5.5, самым медленным — Watcom (на ос семейства Windows). В среде Gnu Linux разница между GCC и Clang не столь значительна, как в среде Windows. Среднее время общее в режиме интерпретации время возросло на 21% относительно времени выполнения в режиме компиляции. При этом, в среднем, на 24.6% возросло время на процесс сопоставления с образцом. А время построения результата выражения возросло на 38.3 %. В таблице 3.7 приведена усреднённая статистика времени самораскрутки в режиме компиляции и интерпретации команд RASL с указанием среднего квадратичного отклонения.

Таблица 3.7: Усреднённая статистика снижения производительности в режиме интерпретации

	Среднее/минимальное/Максимальное значение снижения производительности при интерпритации (%)	Среднее квадратичное отклонение (%)
Общее время компиляции	21.002 / 5.280/ 29.162	8.762
Время сопоставления	24.635 / 6.861/ 34.035	9.534
Время построения результата	38.320 / 8.927 / 55.727	15.975

Аналогичная ситуация произошла и при оценке размера получаемого исполнительного файла. Это вполне объясняется тем, что каждый компилятор имеет разные алгоритмы оптимизации и генерации кода. Если усреднить полученные значения и сравнивать их в рамках фиксированной конфигурации целевого компилятора и платформы, то разница между размером исполнимых файлов для режима интерпретации языка сборки от режима прямой компиляции составляет от 2 до 3 раз.

Заключение

Таким образом, в ходе выполнения данной работы был улучшен интерпретатор для языка сборки языка Простого Рефала. Была получена разница в 2-3 раза в объеме исполнительного файла при генерации в режиме интерпретации RASL. Что больше, чем результат, полученный в работе [1] (уменьшение размера в 35%). При этом, потеря производительности в режиме компиляции лежит в диапазоне 7-29%. Был получен опыт работы с чужим кодом, а также, открыты для автора данной работы диалекты языка Рефал.

Список литературы

[1] Сухарев В.И. «Оптимизация компилятора Простого Рефала в C++ путём замены прямой кодогенерации на интерпретацию». Курсовой проект, МГТУ им. Н.Э.Баумана, кафедра «Теоретическая информатика и компьютерные технологии (ИУ9)», 2009.

[2] С.А. Романенко. МАШИННО-НЕЗАВИСИМЫЙ КОМПИЛЯТОР С ЯЗЫКА РЕКУРСИВНЫХ ФУНКЦИЙ. Дисертация на соискание ученой степени кандидата физико-математических наук. Руководитель: В.С. Штаркман. Москва. 1978г.

[3] Recursive functions algorithmic language [электронный ресурс], — Режим доступа: http://www.refal.ru/.

[4] Документация и заметки к языку и компилятору Простого Рефала, исходные тексты программы [электронный ресурс], — Режим доступа: https://github.com/Mazdaywik/simple-refal/.

[5] Мансурова М.Г. «Компилятор из RASL в C++». Курсовой проект, МГТУ им. Н.Э.Баумана, кафедра «Теоретическая информатика и компьютерные технологии (ИУ9)», 2015.

[6] А.П. Немытых. Заметка о переносе реализации Рефала-5 на операционную систему Windows Mobile 5.0 [электронный ресурс], — Режим доступа: http://skif.pereslavl.ru/psi-info/rcms/rcms-publications/2014-rus/refal-16.pdf.

[7] Арк.В.Климов. Программирование на языке Рефал. 2004 [электронный ресурс], — Режим доступа: http://www.refal.net/∼arklimov/refal6/manual.htm.

Приложение

1. Пример трансляции функции до доработки компилятора

```
Func {
  ((e.FilePath)) (e.Text) e.A =
    <WriteLine e.FilePath>
    <WriteLine e.Text>
    <WriteToHandle <FOpen 'w' e.FilePath> e.Text>
     ;
}
```

```
static refalrts::FnResult Func(refalrts::Iter arg_begin, refalrts
   ::Iter arg_end) {
  refalrts::this_is_generated_function();
  do {
    refalrts::Iter bb_0 = arg_begin;
    refalrts::Iter be_0 = arg_end;
    refalrts::move_left( bb_0, be_0 );
    refalrts::move_left( bb_0, be_0 );
    refalrts::move_right( bb_0, be_0 );
    static refalrts::Iter eFilePath_1_b_1;
    static refalrts::Iter eFilePath_1_e_1;
    static refalrts::Iter eText_1_b_1;
    static refalrts::Iter eText_1_e_1;
    static refalrts::Iter eFilePath_1_b_2;
    static refalrts::Iter eFilePath_1_e_2;
    static refalrts::Iter eText_1_b_2;
    static refalrts::Iter eText_1_e_2;
    // ( ( e.FilePath#1 ) ) ( e.Text#1 ) e.A#1
    refalrts::Iter bb_1 = 0;
    refalrts::Iter be_1 = 0;
    if( ! refalrts::brackets_left( bb_1, be_1, bb_0, be_0 ) )
      break;
    refalrts::Iter bb_2 = 0;
    refalrts::Iter be_2 = 0;
    if( ! refalrts::brackets_left( bb_2, be_2, bb_1, be_1 ) )
      break;
```

```
26      refalrts::Iter bb_3 = 0;
27      refalrts::Iter be_3 = 0;
28      if( ! refalrts::brackets_left( bb_3, be_3, bb_0, be_0 ) )
29        break;
30      if( ! refalrts::empty_seq( bb_1, be_1 ) )
31        break;
32      eFilePath_1_b_1 = bb_2;
33      eFilePath_1_e_1 = be_2;
34      eText_1_b_1 = bb_3;
35      eText_1_e_1 = be_3;
36      // Unused closed variable e.A#1
37  #ifdef INTERPRET
38      const static refalrts::ResultAction raa[] = {
39        {refalrts::icBracket, 0, 0, refalrts::ibOpenCall},
40        {refalrts::icFunc, (void*) & WriteLine, (void*) "WriteLine
            "},
41        {refalrts::icSpliceEVar, & eFilePath_1_b_1, &
            eFilePath_1_e_1},
42        {refalrts::icBracket, 0, 0, refalrts::ibCloseCall},
43        {refalrts::icBracket, 0, 0, refalrts::ibOpenCall},
44        {refalrts::icFunc, (void*) & WriteLine, (void*) "WriteLine
            "},
45        {refalrts::icSpliceEVar, & eText_1_b_1, & eText_1_e_1},
46        {refalrts::icBracket, 0, 0, refalrts::ibCloseCall},
47        {refalrts::icBracket, 0, 0, refalrts::ibOpenCall},
48        {refalrts::icFunc, (void*) & WriteToHandle, (void*) "
            WriteToHandle"},
49        {refalrts::icBracket, 0, 0, refalrts::ibOpenCall},
50        {refalrts::icFunc, (void*) & FOpen, (void*) "FOpen"},
51        {refalrts::icChar, 0, 0, 'w'},
52        {refalrts::icCopyEVar, & eFilePath_1_b_1, & eFilePath_1_e_1
            },
53        {refalrts::icBracket, 0, 0, refalrts::ibCloseCall},
54        {refalrts::icCopyEVar, & eText_1_b_1, & eText_1_e_1},
55        {refalrts::icBracket, 0, 0, refalrts::ibCloseCall},
56        {refalrts::icEnd}
57    };
```

```
58      refalrts::Iter allocs[2*sizeof(raa)/sizeof(raa[0])];
59      refalrts::FnResult res = refalrts::interpret_array( raa,
           allocs, arg_begin, arg_end );
60      return res;
61  #else
62      refalrts::reset_allocator();
63      refalrts::Iter res = arg_begin;
64      if( ! refalrts::copy_evar( eFilePath_1_b_2, eFilePath_1_e_2,
           eFilePath_1_b_1, eFilePath_1_e_1 ) )
65        return refalrts::cNoMemory;
66      if( ! refalrts::copy_evar( eText_1_b_2, eText_1_e_2,
           eText_1_b_1, eText_1_e_1 ) )
67        return refalrts::cNoMemory;
68      refalrts::Iter n0 = 0;     if( ! refalrts::alloc_open_call(
           n0 ) )
69        return refalrts::cNoMemory;
70      refalrts::Iter n1 = 0;
71      if( ! refalrts::alloc_name( n1, & WriteLine, "WriteLine" ) )
72        return refalrts::cNoMemory;
73      refalrts::Iter n2 = 0;
74      if( ! refalrts::alloc_close_call( n2 ) )
75        return refalrts::cNoMemory;
76      refalrts::Iter n3 = 0;
77      if( ! refalrts::alloc_open_call( n3 ) )
78        return refalrts::cNoMemory;
79      refalrts::Iter n4 = 0;
80      if( ! refalrts::alloc_name( n4, & WriteLine, "WriteLine" ) )
81        return refalrts::cNoMemory;
82      refalrts::Iter n5 = 0;
83      if( ! refalrts::alloc_close_call( n5 ) )
84        return refalrts::cNoMemory;
85      refalrts::Iter n6 = 0;
86      if( ! refalrts::alloc_open_call( n6 ) )
87        return refalrts::cNoMemory;
88      refalrts::Iter n7 = 0;
89      if( ! refalrts::alloc_name( n7, & WriteToHandle, "
           WriteToHandle" ) )
```

```
90        return refalrts::cNoMemory;
91      refalrts::Iter n8 = 0;
92       if( ! refalrts::alloc_open_call( n8 ) )
93         return refalrts::cNoMemory;
94       refalrts::Iter n9 = 0;
95       if( ! refalrts::alloc_name( n9, & FOpen, "FOpen" ) )
96         return refalrts::cNoMemory;
97       refalrts::Iter n10 = 0;
98       if( ! refalrts::alloc_char( n10, 'w' ) )
99         return refalrts::cNoMemory;
100      refalrts::Iter n11 = 0;
101      if( ! refalrts::alloc_close_call( n11 ) )
102        return refalrts::cNoMemory;
103      refalrts::Iter n12 = 0;
104      if( ! refalrts::alloc_close_call( n12 ) )
105        return refalrts::cNoMemory;
106      refalrts::push_stack( n12 );
107      refalrts::push_stack( n6 );
108      res = refalrts::splice_elem( res, n12 );
109      res = refalrts::splice_evar( res, eText_1_b_2, eText_1_e_2 );
110      refalrts::push_stack( n11 );
111      refalrts::push_stack( n8 );
112      res = refalrts::splice_elem( res, n11 );
113      res = refalrts::splice_evar( res, eFilePath_1_b_2,
           eFilePath_1_e_2 );
114      res = refalrts::splice_elem( res, n10 );
115      res = refalrts::splice_elem( res, n9 );
116      res = refalrts::splice_elem( res, n8 );
117      res = refalrts::splice_elem( res, n7 );
118      res = refalrts::splice_elem( res, n6 );
119      refalrts::push_stack( n5 );
120      refalrts::push_stack( n3 );
121      res = refalrts::splice_elem( res, n5 );
122      res = refalrts::splice_evar( res, eText_1_b_1, eText_1_e_1 );
123      res = refalrts::splice_elem( res, n4 );
124      res = refalrts::splice_elem( res, n3 );
125      refalrts::push_stack( n2 );
```

```
126     refalrts::push_stack( n0 );
127     res = refalrts::splice_elem( res, n2 );
128     res = refalrts::splice_evar( res, eFilePath_1_b_1,
          eFilePath_1_e_1 );
129     res = refalrts::splice_elem( res, n1 );
130     res = refalrts::splice_elem( res, n0 );
131     refalrts::use( res );
132     refalrts::splice_to_freelist( arg_begin, arg_end );
133     return refalrts::cSuccess;
134 #endif
135   } while ( 0 );
136   return refalrts::FnResult(
137    refalrts::cRecognitionImpossible | (__LINE__ << 8)
138   );
139 }
```

2. Пример трансляции тестовой функции до доработки

Так выглядит листинг 8 до внесения улучшений в компилятор.

```
refalrts::Iter bb_0 = arg_begin;
refalrts::Iter be_0 = arg_end;
...
static refalrts::Iter ePattern_1_b_1;
static refalrts::Iter ePattern_1_e_1;
static refalrts::Iter sNumRanges_2_1;
static refalrts::Iter eVars_2_b_1;
static refalrts::Iter eVars_2_e_1;
static refalrts::Iter eCommands_2_b_1;
static refalrts::Iter eCommands_2_e_1;
refalrts::Iter bb_1 = 0;
refalrts::Iter be_1 = 0;
if( ! refalrts::brackets_left( bb_1, be_1, bb_0, be_0 ) )
  break;
ePattern_1_b_1 = bb_1;
ePattern_1_e_1 = be_1;
refalrts::Iter bb_2 = 0;
refalrts::Iter be_2 = 0;
eVars_2_b_1 = bb_2;
eVars_2_e_1 = be_2;
eCommands_2_b_1 = bb_0;
eCommands_2_e_1 = be_0;
...
const static refalrts::ResultAction raa[] = {
  {refalrts::icBracket, 0, 0, refalrts::ibOpenBracket},
  {refalrts::icSpliceEVar, & eVars_2_b_1, & eVars_2_e_1},
  {refalrts::icBracket, 0, 0, refalrts::ibCloseBracket},
  {refalrts::icBracket, 0, 0, refalrts::ibOpenBracket},
  {refalrts::icBracket, 0, 0, refalrts::ibOpenBracket},
  {refalrts::icFunc, (void*) & CmdComment, (void*) "CmdComment"},
  ...
  {refalrts::icBracket, 0, 0, refalrts::ibCloseBracket},
  {refalrts::icEnd}
};
```

3. Исходный текст скрипта для тестирования

bootstrap/benchmark.bat

```
setlocal
  call ..\c-plus-plus.conf.bat
  del int-%1 comp-%1
  set FILES=srefc Algorithm Context Error FindFile Generator
     Lexer ParseCmdLine Parser^
    SymTable refalrts Library LibraryEx
  for /L %%i in (1, 1, 9) do (
    echo Interpret %%i
    ..\compiler\srefc -c "%CPPLINE% -I../srlib -DINTERPRET" %
       FILES% 2>> int-%1
  )
  for /L %%i in (1, 1, 9) do (
    echo Compile %%i
    ..\compiler\srefc -c "%CPPLINE% -I../srlib" %FILES% 2>> comp
       -%1
  )
  del *.exe
  if exist *.obj del *.obj
  if exist *.o del *.o
  if exist *.tds del *.tds endlocal
```

compiler/benchmark.bat

```
del %1
for /L %%i in (1, 1, 9) do echo %%i & ..\compiler\srefc
   @benchmark_data.lst 2>> %1
del *.cpp
echo.>> %1
echo.>> %1
for %%f in (srefc.exe) do
  echo File size %%~zf bytes >> %1
```

compiler/benchmark_data.lst

```
Algorithm Context Error
FindFile Generator Lexer
ParseCmdLine Parser srefc
SymTable Algorithm Context
```

```
Error FindFile Generator
Lexer ParseCmdLine Parser
srefc SymTable Algorithm
Context Error FindFile
Generator Lexer ParseCmdLine
Parser srefc SymTable
Algorithm Context Error
FindFile Generator Lexer
ParseCmdLine Parser srefc
SymTable Algorithm Context
Error FindFile Generator
Lexer ParseCmdLine Parser
srefc SymTable
```

compiler/mean.awk

```
{
  key = FILENAME "␣||";
  for (i = 1; i < NF; ++i) {
    n = i+1;
    if ($n ~ /seconds/) {
     metric[key] += $i;
     count[key] += 1;
     squares[key] += $i * $i;
    }
    key = key "␣" $i;
  }
}
/^File size/ {   filesize[FILENAME] = $3; }
function size_format(bytes) {
  return bytes / 1024.0 "␣Kb"
}
END {
  print "␣PARAM␣||␣interpreted␣||␣compiled␣||␣%";
  for (key in metric) {
    mean = metric[key] / count[key];
    mean_sq = squares[key] / count[key];
    sq_mean = mean * mean;
    suf = ""
```

```
    if (key ~ /int/) {
      comp_key = key
      gsub("int", "comp", comp_key);
      if (count[comp_key] ) {
        mean_comp = metric[comp_key] / count[comp_key];
        mean_sq_comp = squares[comp_key] / count[comp_key];
        sq_mean_comp = mean_comp * mean_comp;
        if (mean_comp) {
          suf = "␣||␣" (100.0 * mean / mean_comp - 100.0) "␣%"
       }
        line_int = mean "␣+-␣" sqrt(mean_sq - sq_mean);
        line_comp = mean_comp "␣+-␣" sqrt(mean_sq_comp -
           sq_mean_comp);
        printf("%s␣=␣%s␣|␣%s␣%s\n", key, line_int, line_comp, suf
           );
      }
    }
  }
  for (mode_int in filesize) {
    if (mode_int ~ /int/) {
      mode_comp = mode_int;
      gsub("int", "comp", mode_comp);
      if (filesize[mode_comp]) {
        int_sz = filesize[mode_int]
        comp_sz = filesize[mode_comp]
        percent = 100.0 * int_sz / comp_sz;
        ratio = 100.0 / percent
        int_sz_fmt = size_format(int_sz);
        comp_sz_fmt = size_format(comp_sz);
        print "zFILE␣SIZE␣" mode_int "␣||␣" int_sz_fmt "␣|␣"
           comp_sz_fmt "␣||␣" percent - 100 "␣%,␣" ratio
     }
    }
  }
}
```

compiler/mean.bat

```
@echo off "c:\Program␣Files␣(x86)\Git\bin\gawk.exe" -f ../
   compiler/mean.awk *.txt | sort
```