Министерство образования и науки Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана (национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ: ИНФОРМАТИКА И СИСТЕМЫ УПРАВЛЕНИЯ

КАФЕДРА: ТЕОРЕТИЧЕСКАЯ ИНФОРМАТИКА И КОМПЬЮТЕРНЫЕ ТЕХНОЛОГИИ

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
К ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЕ
НА ТЕМУ:

Оптимизация построения результатных выражений в компиляторе языка Рефал

Студент _____ _____ _____
(Группа) (Подпись, дата) (И.О.Фамилия)

Руководитель ВКР _____ _____
(Подпись, дата) (И.О.Фамилия)

Консультант _____ _____
(Подпись, дата) (И.О.Фамилия)

Консультант _____ _____
(Подпись, дата) (И.О.Фамилия)

Нормоконтролер _____ _____
(Подпись, дата) (И.О.Фамилия)

2016 г.

Аннотация

Темой работы является «Оптимизация построения результатных выражений в компиляторе языка Рефал». Объем дипломной работы 55 страницы. В ней размещены 7 рисунков, 20 листингов и 3 таблицы. При написании работы использовалось 6 источников литературы.

Объектом исследования при написании работы послужил компилятор Простого Рефала. Предметом исследования стала оптимизация построения результатных выражений.

В дипломную работу входят пять глав. Первая посвящена описанию предметной области, во второй формулируется расширенная постановка задачи, а оставшиеся три посвящены разработке, реализации и тестированию.

Во введении постановка задачи формулируется абстрактно, на математическом уровне. Расширенная постановка задачи описывается более формально, с учетом всех нюансов и особенностей предметной области.

Заключение посвящено основным выводам и идеям, о дальнейшем развитии темы, поднятой в данной работе.

Содержание

Введение

Функциональное программирование — парадигма программирования, в которой вычисления выполняются над неизменяемыми переменными посредством вызова функций. В данный период времени широко известны такие языки программирования как Haskell, Scala, Lisp.

Несмотря на то, что программы, написанные в функциональном стиле, как правило, требуют больше вычислительных ресурсов и работают медленнее, у данного подхода, несомненно, есть и свои плюсы.

Так как в функциональном программировании каждый символ является неизменяемым, то функции не имеют побочных действий. Благодаря этому можно протестировать каждую функцию в программе, используя только лишь нужные аргументы. Нет необходимости вызывать функции в специальном порядке или восстанавливать сложное состояние системы. Очевидно, что это значительно упрощает подготовку unit-тестов и отладку программ.

Функциональная программа сразу готова к распараллеливанию без каких-либо изменений. Нет необходимости задумываться о deadlock-ах или состояниях гонки (race conditions). Синхронизация попросту не нужна, так как ни один фрагмент данных в функциональной программе не изменяется дважды. Это означает, что присутствует возможность добавить новый поток в любую программу, даже не задумываясь при этом о проблемах, присущих императивным языкам.

Еще одно интересное свойство функциональных языков программирования состоит в том, что их можно изучать с математической точки зрения. Так как функциональный язык — это реализация формальной системы, то все математические операции, могут быть применены и к функциональным программам. Компилятор, например, может

конвертировать участок кода в эквивалентный, более эффективный фрагмент, используя математическое обоснование их эквивалентности.

РЕФАЛ — один из старейших функциональных языков программирования. Создателем РЕФАЛа является Валентин Турчинов. Язык был впервые реализован в 1968 году в России. В МГТУ имени Н.Э. Баумана, на кафедре ИУ9, компилятор этого языка используется в качестве учебного полигона для курсовых и дипломных работ.

В рамках данной работы предлагается ознакомиться с компилятором одного из диалектов РЕФАЛа, а именно с компилятором Простого Рефала. Интерес, прежде всего, вызывает возможность использовать математический аппарат на практике — для улучшения качества кода. В рамках данного компилятора допустимы оптимизации различных видов, предлагается остановиться на одной из таких.

Целью данной работы является оптимизация построения результатных выражений в компиляторе Простого Рефала. Такая оптимизация увеличит скорость выполнения и уменьшит количество потребляемой памяти для кода, генерируемого компилятором.

Постановка задачи может быть сделана следующим образом: необходимо переработать алгоритм преобразования одного списка в другой, минимизировав при этом вычислительные затраты.

Для того, чтобы описать задачу с меньшей степенью абстракции в работе присутствуют разделы, посвященные синтаксису РЕФАЛа, а также структуре компилятора Простого Рефала. В последующих главах четко формулируется расширенная постановка задачи, после чего описывается разработка алгоритма и его реализация в компиляторе.

Практическая значимость данной работы очевидна. Опция компиляции в режиме оптимизации может использоваться для программ, в которых существенна скорость выполнения и объем используемых вычислительных ресурсов. Кроме того, теоретические результаты исследования могут быть использованы для внедрения данной оптимизации в другие диалекты

РЕФАЛа и другие функциональные языки программирования со схожей структурой реализации.

Заинтересованность в данной теме также вызывает тот факт, что РЕФАЛ является отечественным языком программирования. По этой причине данная работа является некоторым вкладом в развитие информационных технологий Российской Федерации.

Глава 1. Обзор предметной области

1.1 Общие понятия

РЕФАЛ — РЕкурсивный Функциональный АЛгоритмический язык, язык функционального программирования, ориентированный на символьные вычисления, обработку и преобразование текстов. В 1978 году С. А. Романенко опубликовал диссертацию «Машинно-независимый компилятор с языка рекурсивных функций» [1]. В течение нескольких следующих лет были реализованы несколько различных диалектов этого языка.

Одним из диалектов языка РЕФАЛ является РЕФАЛ-5 [2]. Как видно на листинге 1, функция в РЕФАЛе-5 состоит из предложений, разделенных точкой с запятой. Каждое предложение состоит из левой и правой части, называемых образцовым выражением (образцом) и результатным выражением (результатом) соответственно.

В данном примере (см. листинг 1) приведена функция, определяющая является ли заданное выражение палиндромом. Палиндром — это выражение, читающееся одинаково, начиная с начала и с конца. Например, XYX является палиндромом, а XYZ не является.

Листинг 1, Функция на языке РЕФАЛ-5.

```
Pal {
  /*empty*/ = True;
  s.1 = True;
  s.1 e.2 s.1 = <Pal e.2>;
  e.1 = False;
}
```

В документации языка РЕФАЛ-5 определен набор базовых понятий (понятий Базисного Рефала), использующихся во многих других диалектах. РЕФАЛ-машиной называется абстрактное устройство, которое выполняет РЕФАЛ-программы. Она имеет два хранилища информации: поле программы и

поле зрения. Поле программы содержит РЕФАЛ-программу, которая загружается в машину перед запуском и не изменяется в ходе выполнения. Поле зрения хранит выражение без свободных переменных, такие выражения называются определенными. Свободные переменные — это s-переменные (атомы), t-переменные (термы) и e-переменные (выражения). Терм — это или атом, или выражение в круглых (структурных) скобках. Виды атомов, как правило, определяются диалектом. В языке РЕФАЛ-5 — это идентификаторы, числа, символы. Объектное выражение — это выражение, содержащее только атомы и структурные скобки. Образцовое выражение — это выражение, содержащее атомы, структурные скобки и свободные переменные. Результатное выражение — это выражение, содержащее атомы, структурные скобки, свободные переменные и скобки конкретизации.

В процессе работы РЕФАЛ-машины выполняется процедура конкретизации, которая обозначается угловыми скобками. Под процедурой конкретизации подразумевается вызов функции, расположенной после открывающейся угловой скобки, в которую передаются все остальные символы между именем функции и закрывающейся угловой скобкой. Такую конструкцию называют термом конкретизации. В результате работы конкретизации, функция возвращает некоторое определенное выражение (т.е. в нем отсутствуют свободные переменные и скобки конкретизации). Это выражение заменяет терм конкретизации в поле зрения. В случае, если скобки конкретизации вложены в друг друга, РЕФАЛ-машина вызывает их поочередно, слева направо, начиная с самой внутренней.

Еще одним важным понятием является процедура сопоставления с образцом. При запуске функции, выполняется попытка сопоставить входное выражение с левой частью первого предложения. Если сопоставление удается, то возвращается правая часть, иначе происходит переход к следующему предложению. Если ни одно предложение не может быть сопоставлено с аргументом функции, то программа завершает свою работу с ошибкой.

Далее более подробно рассматривается процесс сопоставления с образцом для некоторого предложения. При сопоставлении с образцом аргумент функции проецируется на образец (отображается). Проецирование выполняется по следующим правилам [2]:

1. После того, как отображена открывающаяся структурная скобка, следующей отображается соответствующая ей закрывающаяся скобка.
2. Если в результате предыдущих шагов оба конца вхождения некоторой e-переменной уже отображены, но эта переменная еще не имеет значения (ни одно другое ее вхождение не было отображено), то эта переменная отображается следующей. Такие вхождения называются закрытыми e-переменными. Две закрытые e-переменные могут появиться одновременно; в этом случае та, что слева, отображается первой.
3. Вхождение переменной, которая уже получила значение, является повторным. Скобки, символы, s-переменные, t-переменные и повторные вхождения e-переменных являются жесткими элементами. Если один из концов жесткого элемента отображен, проекция второго конца определена однозначно. Если пункты 1 и 2 неприменимы и имеется несколько жестких элементов с одним спроектированным концом, то из них выбирается самый левый. Если возможно отобразить этот элемент, тогда он отображается, и процесс продолжается дальше. В противном случае объявляется тупиковая ситуация.
4. Если пункты 1-3 неприменимы и имеются несколько e-переменных с отображенным левым концом, то выбирается самая левая из них. Она называется открытой e-переменной. Первоначально она получает пустое значение, то есть ее правый конец проектируется на тот же узел, что и левый. Другие значения могут присваиваться открытым переменным через удлинение (см. пункт 6).

5. Если все элементы образца отображены, то это значит, что процесс сопоставления успешно завершен.
6. В тупиковой ситуации процесс возвращается назад к последней открытой e-переменной (то есть к той, что имеет максимальный номер проекции), и ее значение удлиняется (то есть проекция ее правого конца сдвигается на один объект вправо). После этого процесс возобновляется. Если переменную нельзя удлинить, удлиняется предшествующая открытая переменная, и так далее. Если не имеется подлежащих удлинению открытых переменных, процесс сопоставления не удался.

Рассмотрим пример того, как происходит проецирование аргумента на образец (см. листинг 2). Здесь `Argument` — это аргумент, передаваемый на вход некоторой функции, в левой части предложения которой содержится образец `Pattern`. Последовательность цифр под образцом показывает то, в каком порядке происходит отображение объектов выражения. Сначала отображаются скобки, потом открытой e-переменной `e.1` присваивается пустое выражение, вычисляются закрытые переменные `s.X` и `e.2`. Далее открытая переменная `e.3` становится пустой, после этого при попытке отобразить `s.X`, возникает тупиковая ситуация так как `'M'` не равен `'X'`. Выполняется удлинение `e.3`, но это не помогает избежать тупиковой ситуации потому, что ‘M’ не равен ‘Y’. По этой причине происходит еще одно удлинение и `e.3` становится равным `'XYZ'`. Однако, отображение `s.X` все еще невозможно, ведь `'M'` не равно `'Z'`. Поэтому происходит удлинение `e.1`, обновляется значение `s.X` и цикл `'XYZ'` повторяется. После еще четырех удлинений, `e.1` становится равным `'METAS'`, а `s.X` равным `'Y'`. Далее, уже после первого удлинения `e.3`, будет получено корректное отображение. В листинге 2 финальные значения переменных указаны под меткой `Variables`.

Листинг 2, Проецирование аргумента функции на образец.

```
Argument:
 ('METASYSTEM INDEX')'XYZ'
Pattern:
 (    e.1   s.X   e.2   )    e.3   s.X   e.4
 1    3     4     5     2    6     7     8
Variables:
 e.1 = 'METAS'
 s.X = 'Y'
 e.2 = 'STEM_INDEX'
 e.3 = 'X'
 e.4 = 'Z'
```

1.2 Простой Рефал

Простой Рефал [3] — является одним из диалектов Базисного Рефала. Компилятор Простого Рефала транслирует исходные тексты на Рефале в исходные тексты на C++, получившийся набор файлов транслируется компилятором C++ (совместимым со стандартом ANSI/ISO C++ 1998 года).

Простой Рефал, бесспорно, имеет много общих черт с РЕФАЛом-5. Функции внешне имеют сходный синтаксис, но есть небольшие различия. Функции оперируют объектными выражениями: принимают в качестве аргумента одно объектное выражение и возвращают его в качестве результата. Семантика выполнения программы тоже определяется в терминах абстрактной рефал-машины. Идентична семантика сопоставления с образцом. Структура результатного выражения незначительно отличается. Затраты времени на операции схожи. Реализация Простого Рефала использует классическое списковое представление, поэтому особенности стоимости отдельных операций (сопоставление с открытыми и повторными переменными, стоимость конкатенации и копирования) те же, что и в реализации РЕФАЛа-5.

Функция Простого Рефала определяется грамматикой, представленной в листинге 3. В рассматриваемом языке атомами (атомарными термами) являются символы, числа, указатели на функции, идентификаторы

(глобальные константы). Идентификаторы начинаются с символа решетки. Скобки бывают трех видов: структурные (круглые), конкретизационные (угловые), абстрактные (квадратные). Абстрактные скобки — особый вид скобок, позволяющий помечать их именами функций, делая при этом уникальными. Как правило, для таких целей используются функции с пустым телом. Символ `'^'` позволяет помечать переменные в образце, сообщая компилятору о том, что такие переменные перезаписываются и предыдущие их значения игнорируются.

Листинг 3, Грамматика для функции в Простом Рефале.

```
FunctionDefinition ::= [ "$ENTRY" ] NAME Block.
Block ::= "{" { Sentence } "}".
Sentence ::= Pattern "=" Result ";".
Pattern ::= { PatternTerm }.
Result ::= { ResultTerm }.
PatternTerm ::=
  CommonTerm |
  "(" Pattern ")" |
  "[" NAME Pattern "]" |
  RedefinitionVariable.
RedefinitionVariable ::= VARIABLE "^".
ResultTerm ::=
  CommonTerm |
  "(" Result ")" |
  "[" NAME Result "]" |
  "<" Result ">" |
  Block.
CommonTerm ::=
  CHAR |
  NUMBER |
  NAME |
  VARIABLE |
  "#" NAME.
```

В Простом Рефале есть возможность объявлять анонимные функции, располагая блок объявления функции (предложения внутри фигурных скобок) вместо указателя на нее.

Для Простого Рефала существует набор библиотечных функций, расположенных в файлах «Library.sref» и «LibraryEx.sref». Далее определяются некоторые из них:

1. `WriteLine` выводит свой аргумент в консоль, возвращает пустое выражение.
2. `Fetch` принимает на вход два аргумента: некоторое выражение и указатель на функцию. Возвращает результат, полученный путем применения функции, из второго аргумента к выражению из первого.
3. `Map` принимает на вход указатель на функцию и последовательность термов. Возвращает последовательность объектов, полученных путем поэлементного применения функции, из первого аргумента, ко всем термам из второго.
4. `Reduce` принимает на вход указатель на функцию, аккумулятор и последовательность термов. Под сверткой будет пониматься вызов функции, из первого аргумента, с передачей в нее аккумулятора и терма последовательности. Свертка выполняется для всех термов из последовательности, в порядке слева направо, при этом на каждом шаге аккумулятору присваивается результат свертки. Результатом работы функции является значения аккумулятора после выполнения всех сверток.
5. `Inc` принимает на вход число, а возвращает число на единицу больше.
6. `Dec` принимает на вход число, а возвращает число на единицу меньше.
7. `Add` принимает два числа, а возвращает их сумму.
8. `Sub` принимает два числа, а возвращает разность первого и второго.
9. `ArgList` возвращает аргументы командной строки.
10. `Compare` сравнивает два терма. Если первый терм больше второго,

то возвращается ‘>’, если термы равны возвращается ‘=’, иначе возвращается ‘<’. Ранжирование термов задается следующей последовательностью типов объектов в порядке убывания значимости: скобочный терм, число, литеральный символ, функция, идентификатор. Замыкания с контекстом и абстрактные типы данных не поддерживаются. Скобочные термы сравниваются рекурсивно в лексикографическом порядке.

1.3 Структура компилятора Простого Рефала

В рамках данного раздела будут рассмотрены детали, связанные с построением результатных выражений, необходимые для строгой постановки задачи. Для упрощения понимания, будут последовательно рассмотрены все стадии компиляции и функции, отвечающие за их выполнение. Далее в схемах будут фигурировать следующие обозначения: прямоугольник — это данные, овал — это процесс, ромб — это условие.

Первая стадия — это анализ, состоящий из фаз лексического, синтаксического и семантического анализов.

Лексический анализ, в рассматриваемом компиляторе, выполняется с помощью функции `LexFolding` из файла «Lexer.sref» (см. рисунок 1). На вход функция принимает название файла. Далее из файла считывается текст программы с помощью функции `ReadFile`. Текст программы обрабатывается с помощью функции `Root`, генерирующей последовательность токенов. Далее эта последовательность передается в `NormilizeTokens`. В ней последовательность приводится к общему виду, в котором впоследствии будет использоваться на других стадиях компиляции. Токены, полученные на выходе функции `LexdFolding`, имеют следующую структуру (см. листинг 4). Здесь s.TokType — тип токена, `s.LineNumber` — номер строки, `e.Info` — значение токена, в зависимости от типа.

Листинг 4, Последовательность токенов.

```
<LexFolding e.FileName> == e.Tokens
e.Tokens ::= (s.TokType s.LineNumber e.Info)*
```

Синтаксический анализ выполняется с помощью функции `ParseProgram` из файла «Parser.sref» (см. рисунок 2). Она принимает на вход список ошибок и последовательность токенов, а возвращает расширенный список ошибок, таблицу символов и абстрактное

синтаксическое дерево. Вызываемая из `ParseProgramm` функция `ParseElements` выполняет синтаксический анализ методом рекурсивного спуска. Абстрактное синтаксическое дерева (Abstract Syntax Tree, аббревиатура AST) представляет собой древовидную структуру, в которой хранится информация о функциях и предложениях, входящих в эти функции (см. листинг 5).

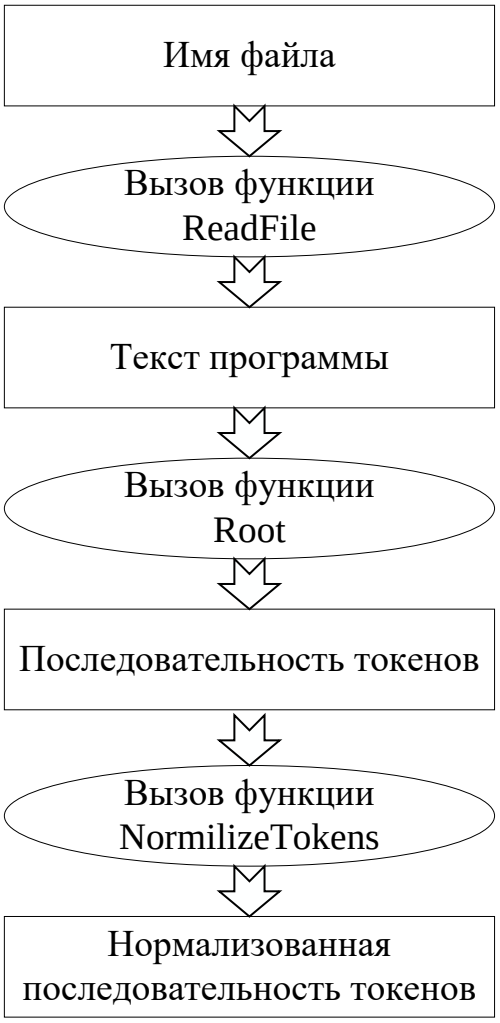

Рисунок 1, Лексический анализ, выполняемый функцией LexFolding.

Далее по абстрактному синтаксическому дереву строится промежуточное представление, называемое RASL. RASL представляет собой последовательность команд различного типа. За его генерацию отвечает функция `HighLevelRASL`, расположенная в файле «HighLevelRASL.sref» (см. рисунок 3). В ней для каждой функции из дерева вызывается `HighLevelRASL-Function`, для каждого предложения в функции

HighLevelRASL-Sentence, для левой и правой части предложения GenPattern и GenResult соответственно. Эти функции отвечают за генерацию команд RASL для функций, предложений, образца и результата соответственно. В полученном промежуточном представлении каждой функции соответствует блок команд RASL.

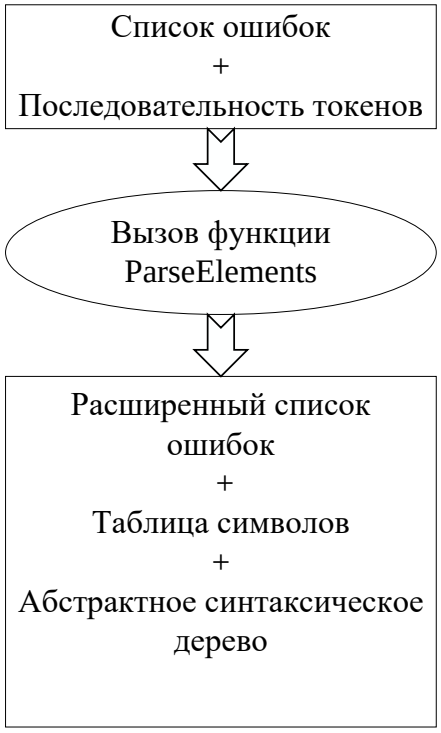

Рисунок 2, Синтаксический анализ, выполняемый функцией ParseProgram.

Промежуточный код RASL можно считать некоторым аналогом ассемблера (абстрактным ассемблером), в роли памяти для которого выступает массив context. Этот массив используется в сгенерированном коде на языке C++. В исходной версии компилятора в нем содержится информация о контексте: указатели на диапазоны, используемые в процессе сопоставления, указатели на переменные, полученные в ходе сопоставления с образцом, и указатели на новые элементы, полученные в ходе построения результата. Регистрами такого ассемблера можно считать переменные: arg_begin (начало диапазона), arg_end (конец диапазона) и res (позиция в списке для вставки переносимого фрагмента).

Модель команд RASL можно описать следующим образом. Последовательность команд внутри функций, обозначаемых #Function,

должна начинаться с #CmdIssueMem, которая резервирует массив context. Далее располагаются команды сопоставления с образцом (например, #CmdChar) и вложенные предложения #CmdSentence (содержит информацию о допустимых альтернативах в случае ошибки сопоставления). Затем либо должна следовать составная команда #CmdOpenELoop — цикл по открытой e-переменной, за которой следует #CmdFail (возврат из функции значения «сопоставление невозможно»), либо команды создания новых узлов и команды сборки результатного выражения, завершающиеся #CmdReturnResult. Составные команды внутри себя должны иметь точно такую же структуру, за исключением того, что внутри #CmdOpenELoop не может располагаться #CmdFail и не должно быть #CmdSentence.

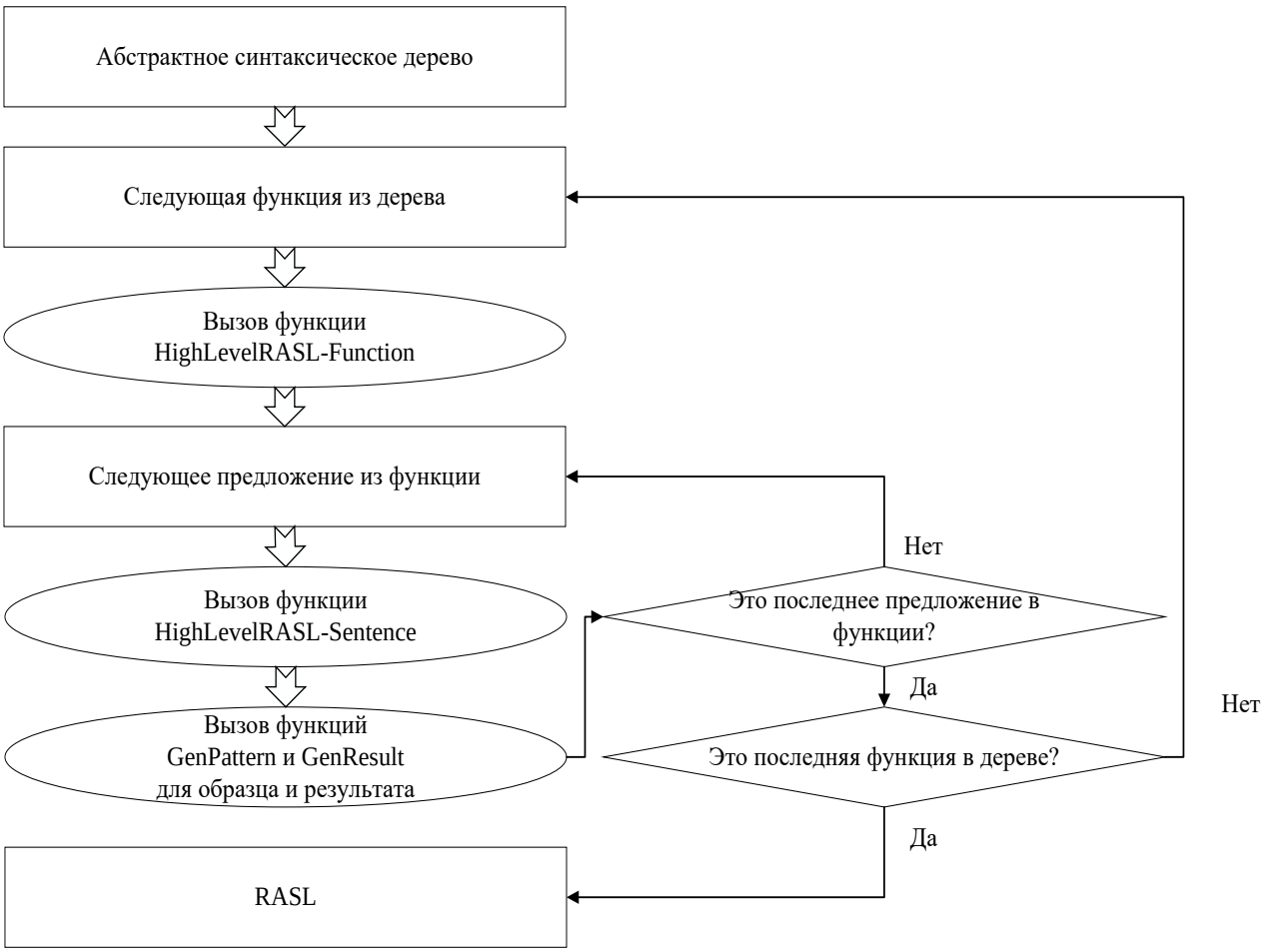

Рисунок 3, Преобразование абстрактного синтаксического дерева в RASL, выполняемое функцией HighLevelRASL.

Многие команды принимают смещение в массиве `context`, например, такими являются команды, команды аллокаций, команды сопоставления с образцом. Для компактных объектов (атомарные значения, s-переменные, t-переменные, отдельные скобки при создании результата) в массиве используется только одна ячейка. Для протяжённых (диапазоны сопоставления, e-переменные, а также последовательности символов — строки) используются две соседние ячейки. Для них в команде указывается номер младшей ячейки N, вторая ячейка N+1 подразумевается. Команд сопоставления с образцом для закрытых e-переменных не генерируется. В роли смещения закрытой e-переменной выступают границы диапазона, в котором она находится.

Стоит подробнее рассмотреть упомянутые ранее команды переноса, аллокаций и сопоставления с образцом, ввиду их значимости для оптимизации. Команда переноса принимает, как аргумент, переносимый объект, определенный смещением в массиве, и переносит его в позицию, заданную регистром `res`. Команда аллокации принимает на вход создаваемый объект и положение в массиве context, в которое будет помещен указатель на объект. Команда сопоставления с образцом принимает диапазон и сопоставляемый объект (в случае сопоставления переменной также принимается ее смещение).

Листинг 5, Абстрактное синтаксическое дерево.

```
<ParseProgram t.ErrorList e.Tokens>
  == t.ErrorList e.AST
e.AST ::= t.ProgramElement*
t.ProgramElement ::=
    (#Function s.ScopeClass (e.Name) e.Sentences)
  | (#Enum s.ScopeClass e.Name)
  | (#Swap s.ScopeClass e.Name)
  | (#Stub s.ScopeClass e.Name)
  | (#Declaration s.ScopeClass e.Name)
  | (#Ident e.Name)
  | (#Separator)
s.ScopeClass ::= #GN-Entry | #GN-Local
e.Sentences ::= t.Sentence*
```

```
t.Sentence ::= ((e.Pattern) (e.Result))
e.Pattern ::= e.Expression
e.Result ::= e.Expression
e.Expression ::= t.Term*
t.Term ::=
    (#TkChar s.Char)
  | (#TkNumber s.Number)
  | (#TkName e.Name)
  | (#TkIdentifier e.Name)
  | (#Brackets e.Expression)
  | (#ADT-Brackets (e.Name) e.Expression)
  | (#CallBrackets e.Expression)
  | (#TkVariable s.Mode e.Index s.Depth)
```

Задачей функции LowLevelRASL (см. рисунок 4), из файла «LowLevelRASL.sref», является развёртывание составных инструкций промежуточного кода (#CmdSentence и #CmdOpenELoop) в более примитивные команды и формирование интерпретируемого кода. LowLevelRASL, в зависимости от переданной опции, осуществляет прямую кодогенерацию, генерирует интерпретируемый код с вызовом интерпретатора, или выполняет и то, и другое. В последнем случае для выбора нужной ветки используется макрос препроцессора C++ INTERPRET. Процедура развертывания команд напрямую зависит от выбранной опции. Стоит отметить принципиальную разницу прямого режима кодогенерации и интерпретации: в первом случае функция на Рефале компилируется в функцию, в которой команды промежуточного кода соответствуют фрагментам выполняемого кода на C++, во втором случае функция состоит из определения константного статического массива команд, вспомогательных массивов литералов и вызова функции интерпретатора.

Оптимизация, рассматриваемая в данной работе, выполняется при генерации высокоуровневого RASL (то есть на выходе HighLevelRASL имеется уже оптимизированное промежуточное представление). В целях упрощения в режиме оптимизации интерпретируемый код генерироваться не

будет (чтобы для промежуточных команд не создавать аналогов режима интерпретации).

Последняя стадия — это синтез. В ней, на основе RASL, генерируется целевой код на языке C++, с помощью функции `GenProgramm` из файла «Generator.sref» (см. рисунок 5). Функция `GenCommand` генерирует код для отдельной команды. Функции, используемые в сгенерированном коде, объявляются в файлах «refalrts.h» и «refalrts.cpp».

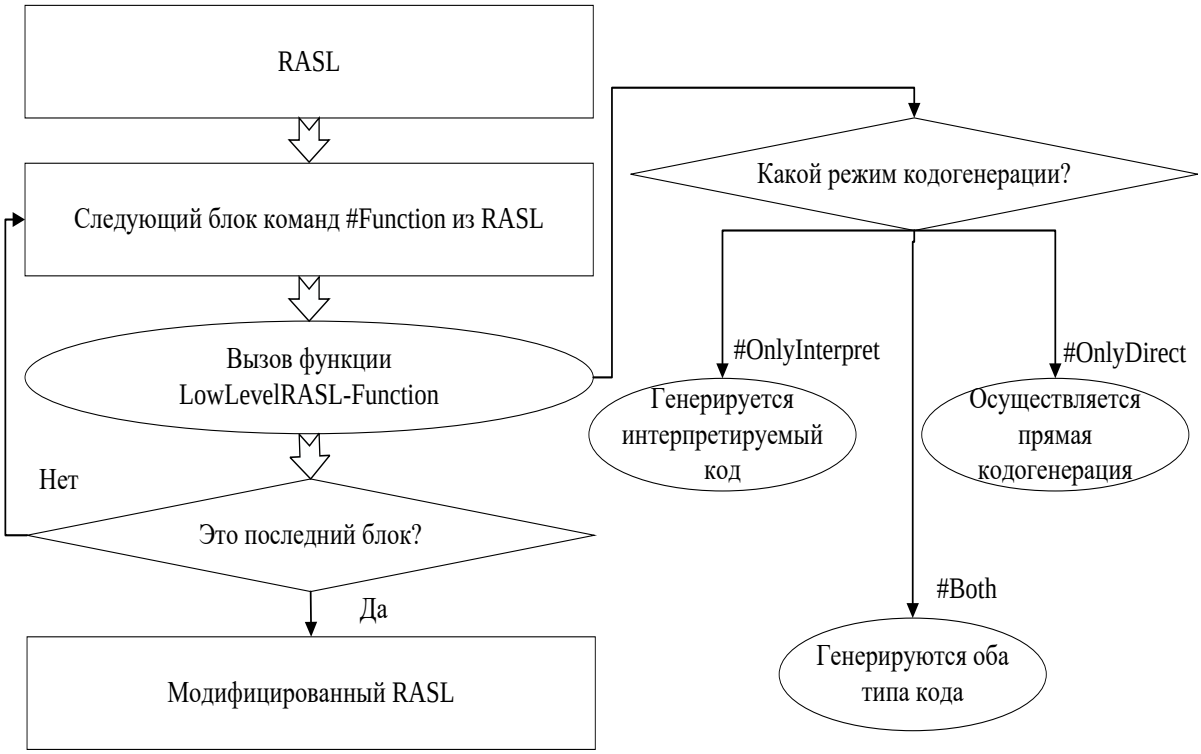

Рисунок 4, Развертывание составных инструкций, выполняемое функцией LowLevelRASL.

Подводя итоги, можно сказать, что компилятор Простого Рефала является многопроходным. Компиляция выполняется за 5 проходов (см. рисунок 6). Текст программы преобразуется в поток токенов с помощью лексического анализатора. Из потока токенов, в результате синтаксического анализа, получается абстрактное синтаксическое дерево. Абстрактное синтаксическое дерево преобразуется в промежуточное представление, называемое RASL. RASL модифицируется в соответствии с режимом

кодогенерации. Далее по модифицированному RASL генерируется целевой код на языке C++.

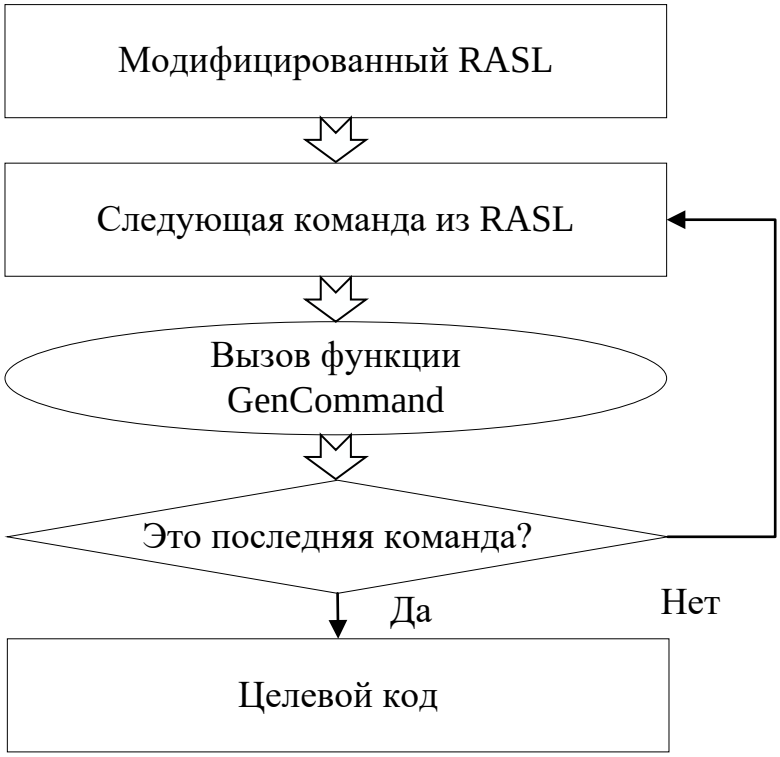

Рисунок 5, Генерация целевого кода, выполняемая функцией GenProgram.

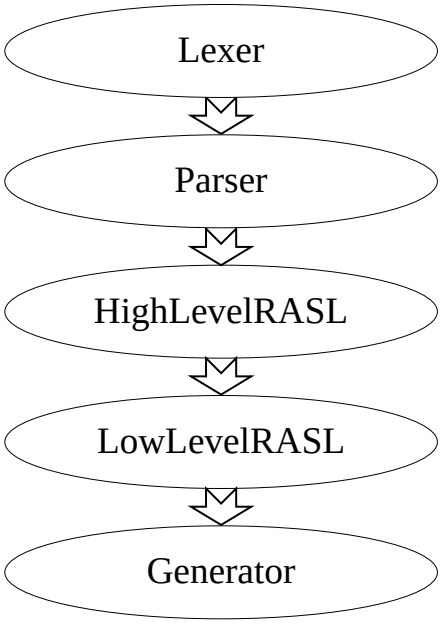

Рисунок 6, Проходы, осуществляемые компилятором.

1.4 Процесс выполнения целевого кода

Следующая тема, требующая обсуждения, — это процесс исполнения целевого кода, имитирующего работу рефал-программы. Другими словами, можно сказать, что темой данной главы являются основные идеи того, как реализована рефал-машина в Простом Рефале.

Поле зрения в Простом Рефале представлено в виде двусвязного списка, элементами которого являются структуры `Node` (см. листинг 6). Здесь `prev` и `next` — указатели на предыдущий и последующий элементы списка; `tag` — тэг типа узла; поля объединения — содержимое узла, в зависимости от его тэга. В поле `char_info` хранятся значения символов, в поле `number_info` значения чисел, в поле `function_info` указатель на функцию и ее имя, в поле `ident_info` указатель на объект, представляющий собой идентификатор.

Особое внимание нужно уделить тому факту, что в узлах открывающих скобок всех видов содержится указатель на закрывающие. Более того скобки конкретизации перевязаны между собой таким образом, чтобы, переходя из одной в другую, можно было обойти все скобки конкретизации (см. рисунок 7) в той последовательности, в которой их должен обрабатывать компилятор (то есть в той последовательности, в которой вызываются функции, соответствующие этим скобкам, а именно слева на право, начиная с самой вложенной). В узлах такие указатели хранятся в поле объединения, под названием `link_info`.

Поле зрения постоянно модифицируется в процессах сопоставления с образцом и конкретизации. После того как аргумент функции был сопоставлен с некоторым образцом, возникает необходимость преобразовать терм конкретизации, расположенный в поле зрения, в результат. Бесспорно, это можно сделать различными способами.

Рассмотрим то, как реализуется эта процедура в целевом коде, полученном с помощью текущей версии компилятора Простого Рефала. При

создании нового узла, он генерируется в списке свободных узлов. После этого необходимо выполнить операцию переноса этого узла в поле зрения. Результатное выражение генерируется, используя узлы переменных из левой части правила. Но все остальные узлы (элементы двусвязного списка) в левой части правила игнорируются. По этой причине, в процессе построения результата, используется большое количество ресурсоемких команд, генерации которых можно было бы избежать, путем повторного использования уже существующих узлов из образца.

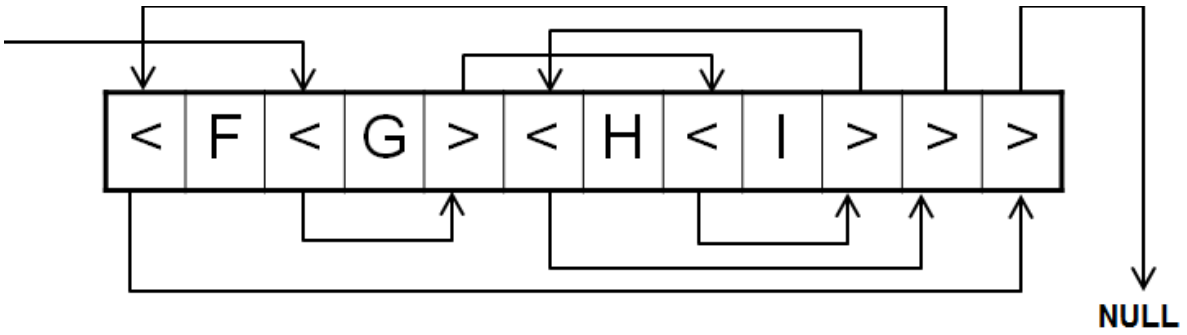

Рисунок 7, Перевязка скобок конкретизации. [3]

Для примера рассмотрим функцию Fab из листинга 7, заменяющую все вхождения символа `'a'` на символ `'b'`. Если ей на вход подать строку `'acc'`, то поле зрения будет изменяться так, как указанно в листинге 8. Безусловно, такой ряд преобразований описывает лишь результат перезаписи терма конкретизации, но не сам процесс. Для более подробной иллюстрации процесса далее рассматривается первый шаг, на котором возникает необходимость в преобразовании выражения `<Fab 'a' 'c' 'c'>` в `'b' <Fab 'c' 'c'>`. В текущей реализации (см. листинг 9), несмотря на наличие совпадающих участков, создается много новых узлов. Более того в ней перенос новых элементов из списка свободных узлов в поле зрения осуществляется посимвольно, в то время как допустим перенос диапазонов. Стоит уточнить, что в рассматриваемых листингах: VF — поле зрения (View Field), FL — список свободных узлов (Free List).

Листинг 6, Узел двусвязного списка, объявленный в «refalrts.h».

```
typedef struct Node {
  NodePtr prev;
  NodePtr next;
  DataTag tag;
  union {
    char char_info;
    RefalNumber number_info;
    RefalFunction function_info;
    RefalIdentifier ident_info;
    NodePtr link_info;
    void *file_info;
    RefalSwapHead swap_info;
  };
} Node;
```

Листинг 7, Функция Fab.

```
Fab {
  'a' e.String = 'b' <Fab e.String>;
  s.Sym e.String = s.Sym <Fab e.String>;
  /*пусто*/ = /*пусто*/;
}
```

Листинг 8, Процесс изменения поля зрения при обработке строки 'acc' функцией Fab.

$VF: \boxed{<} \leftrightarrow \boxed{Fab} \leftrightarrow \boxed{'a'} \leftrightarrow \boxed{'c'} \leftrightarrow \boxed{'c'} \leftrightarrow \boxed{>}$

$VF: \boxed{'b'} \leftrightarrow \boxed{<} \leftrightarrow \boxed{Fab} \leftrightarrow \boxed{'c'} \leftrightarrow \boxed{'c'} \leftrightarrow \boxed{>}$

$VF: \boxed{'b'} \leftrightarrow \boxed{'c'} \leftrightarrow \boxed{<} \leftrightarrow \boxed{Fab} \leftrightarrow \boxed{'c'} \leftrightarrow \boxed{>}$

$VF: \boxed{'b'} \leftrightarrow \boxed{'c'} \leftrightarrow \boxed{'c'} \leftrightarrow \boxed{<} \leftrightarrow \boxed{Fab} \leftrightarrow \boxed{>}$

$VF: \boxed{'b'} \leftrightarrow \boxed{'c'} \leftrightarrow \boxed{'c'}$

Подводя итоги можно утверждать, что представленная схема генерации результатных выражений является неоптимальной и, как следствие, нуждается в оптимизации. Другими словами, необходимо внести ряд изменений в процесс генерации RASL, то есть сгенерировать более

подходящий набор команд. Ввиду того, что некоторые из необходимых команд отсутствуют в текущей версии компилятора, необходимо добавить обработку этих команд в функцию `GenCommand`, генерирующую целевой код.

Листинг 9, Текущая реализация преобразования.

VF: $\boxed{<} \leftrightarrow \boxed{Fab} \leftrightarrow \boxed{`a'} \leftrightarrow \boxed{`c'} \leftrightarrow \boxed{`c'} \leftrightarrow \boxed{>}$

FL:

VF: $\boxed{<} \leftrightarrow \boxed{Fab} \leftrightarrow \boxed{`a'} \leftrightarrow \boxed{`c'} \leftrightarrow \boxed{`c'} \leftrightarrow \boxed{>}$

FL: $\boxed{'b'}$

VF: $\boxed{<} \leftrightarrow \boxed{Fab} \leftrightarrow \boxed{`a'} \leftrightarrow \boxed{`c'} \leftrightarrow \boxed{`c'} \leftrightarrow \boxed{>}$

FL: $\boxed{'b'} \leftrightarrow \boxed{<}$

VF: $\boxed{<} \leftrightarrow \boxed{Fab} \leftrightarrow \boxed{`a'} \leftrightarrow \boxed{`c'} \leftrightarrow \boxed{`c'} \leftrightarrow \boxed{>}$

FL: $\boxed{'b'} \leftrightarrow \boxed{<} \leftrightarrow \boxed{Fab}$

VF: $\boxed{<} \leftrightarrow \boxed{Fab} \leftrightarrow \boxed{`a'} \leftrightarrow \boxed{`c'} \leftrightarrow \boxed{`c'} \leftrightarrow \boxed{>}$

FL: $\boxed{'b'} \leftrightarrow \boxed{<} \leftrightarrow \boxed{Fab} \leftrightarrow \boxed{>}$

VF: $\boxed{>} \leftrightarrow \boxed{<} \leftrightarrow \boxed{Fab} \leftrightarrow \boxed{`a'} \leftrightarrow \boxed{`c'} \leftrightarrow \boxed{`c'} \leftrightarrow \boxed{>}$

FL: $\boxed{'b'} \leftrightarrow \boxed{<} \leftrightarrow \boxed{Fab}$

VF: $\boxed{`c'} \leftrightarrow \boxed{`c'} \leftrightarrow \boxed{>} \leftrightarrow \boxed{<} \leftrightarrow \boxed{Fab} \leftrightarrow \boxed{`a'} \leftrightarrow \boxed{>}$

FL: $\boxed{'b'} \leftrightarrow \boxed{<} \leftrightarrow \boxed{Fab}$

VF: $\boxed{Fab} \leftrightarrow \boxed{`c'} \leftrightarrow \boxed{`c'} \leftrightarrow \boxed{>} \leftrightarrow \boxed{<} \leftrightarrow \boxed{Fab} \leftrightarrow \boxed{`a'} \leftrightarrow \boxed{>}$

FL: $\boxed{'b'} \leftrightarrow \boxed{<}$

VF: $\boxed{<} \leftrightarrow \boxed{Fab} \leftrightarrow \boxed{'c'} \leftrightarrow \boxed{'c'} \leftrightarrow \boxed{>} \leftrightarrow \boxed{<} \leftrightarrow \boxed{Fab} \leftrightarrow \boxed{'a'} \leftrightarrow \boxed{>}$

FL: $\boxed{'b'}$

VF: $\boxed{'b'} \leftrightarrow \boxed{<} \leftrightarrow \boxed{Fab} \leftrightarrow \boxed{'c'} \leftrightarrow \boxed{'c'} \leftrightarrow \boxed{>} \leftrightarrow \boxed{<} \leftrightarrow \boxed{Fab} \leftrightarrow \boxed{'a'} \leftrightarrow \boxed{>}$

FL:

VF: $\boxed{'b'} \leftrightarrow \boxed{<} \leftrightarrow \boxed{Fab} \leftrightarrow \boxed{'c'} \leftrightarrow \boxed{'c'} \leftrightarrow \boxed{>}$

FL: $\boxed{<} \leftrightarrow \boxed{Fab} \leftrightarrow \boxed{'a'} \leftrightarrow \boxed{>}$

Глава 2. Расширенная постановка задачи

После краткого описания предметной области, можно четко формализовать задачу данной работы. Её можно сформулировать следующим образом — необходимо ускорить процесс преобразования терма конкретизации в результатное выражение, то есть преобразовать один двусвязный список в другой с минимальными вычислительными затратами.

В текущей версии компилятора результатные выражения генерируются с нуля, в то время как в терме конкретизации, расположенном в поле зрения, как правило, уже присутствуют некоторые из необходимых узлов. После создания новых элементов, они по одному переносятся из списка свободных узлов в поле зрения, в то время как можно было бы выполнять переносы не отдельно выбранных узлов, а сразу некоторых диапазонов. Предлагается реализовать оптимизацию, основанную на этих двух фактах (см. листинг 10). В рассматриваемом случае вместо того, чтобы генерировать новые элементы, сначала ищется некоторый наиболее схожий диапазон в образце и результате, а потом происходит переопределение содержимого узлов образца для повторного использования в результате. Таким образом, уменьшается количество операций создания новых элементов и операций переноса, что должно хорошо сказаться на скорости выполнения целевого кода. Поиск максимально схожих диапазонов необходим для того, чтобы минимизировать вычислительные затраты на переопределение.

Листинг 10, Оптимизированная реализация преобразования.

VF: $\boxed{<} \leftrightarrow \boxed{Fab} \leftrightarrow \boxed{`a'} \leftrightarrow \boxed{`c'} \leftrightarrow \boxed{`c'} \leftrightarrow \boxed{>}$

FL:

VF: $\boxed{'b'} \leftrightarrow \boxed{Fab} \leftrightarrow \boxed{`a'} \leftrightarrow \boxed{`c'} \leftrightarrow \boxed{`c'} \leftrightarrow \boxed{>}$

FL:

VF: $\boxed{'b'} \leftrightarrow \boxed{<} \leftrightarrow \boxed{'a'} \leftrightarrow \boxed{'c'} \leftrightarrow \boxed{'c'} \leftrightarrow \boxed{>}$

FL:

VF: $\boxed{'b'} \leftrightarrow \boxed{<} \leftrightarrow \boxed{Fab} \leftrightarrow \boxed{'c'} \leftrightarrow \boxed{'c'} \leftrightarrow \boxed{>}$

FL:

Под размеченным образцом (результатом) будем подразумевать образец (результат), каждому элементу которого будет сопоставлено смещение в массиве `context`. Имея информацию о существующих узлах, необходимо разметить результат (добавить каждому элементу выражения поле, со значением смещения в массиве `context`) так, чтобы максимальным образом задействовать уже созданные узлы. При этом предполагается, что узлы можно будет повторно использовать следующим образом:

- без изменения;
- с изменением значения;
- с изменением тэга и значения.

Если вернуться к листингу 10, то в нем скобка конкретизации была повторно использована как символ, то есть имели место изменения тэга и значения. Таким же способом были использованы другие два узла, а именно указатель на функцию использовали как открывающую скобку конкретизации, а символ использовали как указатель на функцию.

Для реализации такой оптимизации необходимо уметь искать общие участки для выражения, находящегося в поле зрения (то есть для образца), и для результатного выражения. В рассмотренном примере (см. листинг 10) повторно используется весь образец, но это не всегда возможно. Более того, могут встречаться случаи, когда в результате больше элементов, чем в образце. В таких ситуациях крайне важно правильно наложить образец на

результат, чтобы значительно уменьшить время выполнения целевого кода. Таким образом, необходимо подобрать алгоритм, удовлетворяющий требованиям данной предметной области.

Глава 3. Разработка алгоритма

3.1 Поиск схожих диапазонов

Сейчас активно развивается область, связанная с поиском сходства между двумя последовательностями, одна из которых, предположительно, содержит крупные фрагменты другой, взятые, возможно, с небольшими изменениями — это поиск плагиата.

В реальных рефал-программах типичной является ситуация, когда левая часть L и правая часть R похожи друг на друга (т.е. L содержит такие участки, которые без изменений или с небольшими отличиями переходят в R). В таких случаях, эти участки можно не генерировать заново, а взять их из L в готовом виде. Возникает проблема: как найти наилучшую последовательность преобразований, для коррекции L в R? Последовательность операторов преобразований можно найти конечным перебором. Действительно, перебрав все варианты доступных преобразований, мы можем за конечное число шагов получить результат, сложность таких алгоритмов экспоненциальная. К сожалению, такое «решение» невозможно использовать в компиляторе, так как при увеличении длин строк необходимое для вычислений время слишком быстро растет, и необходимо искать более быстрые алгоритмы [1].

В литературе указанная проблема известна как проблема коррекции строки в строку (string to string correction problem) и формулируется следующим образом. Пусть даны две строки и несколько видов элементарных преобразований строк. Каждому элементарному преобразованию ставится в соответствие его стоимость. Под стоимостью понимается некоторое числовое значение, обозначающее степень вычислительной сложности выполнения задачи. Стоимостью композиции элементарных преобразований называется сумма стоимостей всех

преобразований, входящих в нее. Необходимо определить преобразование для коррекции первой строки во вторую. При этом такое преобразование должна иметь наименьшую стоимость.

Само собой, решение проблемы коррекции строки в строку самым непосредственным образом зависит от рассматриваемого набора типов элементарных преобразований. Левенштейн предлагал использовать следующий набор операций [4]: вставка элемента, удаление элемента, замена одного элемента на другой. Минимальное количество операций Левенштейна необходимых для преобразования строки P в строку T называют расстоянием Левенштейна (дистанцией редактирования) между строками P и T. Иногда также рассматривают расстояние Дамерау-Левенштейна [5]. В этом случае рассматриваются операции вставки, удаления, и транспозиции (перестановка двух соседних символов).

В диссертации Романенко [1] был предложен алгоритм, использующий три преобразования: вставка одного символа в строку, удаления символа из строки и операция взаимной перестановки двух участков строки. Такой алгоритм принимает на вход две строки A и B, а возвращает последовательность операций редактирования (см. листинг 11). Множество операций вставки обозначено SI, а перестановки двух участков SS. Множеством всех операций редактирования считается S. Функция r определяет сходство строк следующим образом. Если ей на вход подать равные строки, то функция вернет бесконечность, иначе функция вернет максимальную длину совпадающего префикса двух рассматриваемых строк.

Листинг 11, Алгоритм коррекции строки из диссертации Романенко.

```
OPS = {}
X = A
while X != B:
    S0 = {s ∈ S | r(s(X),B) > r(X,B)}
    M = -1
    SM = {}
    for s in S0:
      if r(s(X),B) == M:
```

```
        SM += s
      if r(s(X),B) > M:
        M = r(s(X),B)
        SM = {s}
    SMI = SI ∩ SM
    SMS = SS ∩ SM
    If SMI != {}:
      cur_s = SMI[0]
    elif SMS != {}:
      cur_s = SMI[0]
    else:
      cur_s = SM[0]
    X = cur_s(X)
    OPS += cur_s
return OPS
```

К сожалению, такие подходы не позволяет использовать идею о повторном использовании «неточно совпадающих» диапазонов элементов (то есть диапазонов, в которых некоторые элементы совпадают, а некоторые нет). В рассматриваемом случае необходимо, чтобы множество операций редактирования включало в себя:

- операции вставки элемента;
- операции удаления;
- операции взаимной перестановки двух участков;
- операции обновления.

Новой является операция обновления, она подразумевает изменения некоторых полей узла. Наличие такой операции кардинальным образом меняет суть алгоритма, так как вводиться метрика степени схожести «неточно совпадающих» диапазонов. С помощью такой метрики можно сравнивать вычислительную стоимость обновления для разных типов узлов.

3.2 Модификация алгоритма жадного строкового замощения

Предлагается использовать алгоритм, находящий субоптимальное, применимое на практике решение, а именно модификацию алгоритма

жадного строкового замощения (Greedy String Tiling) [6]. Такой алгоритм первым делом находит самые длинные совпадающие участки, а это уменьшает количество операций переноса.

Изначально алгоритм жадного строкового замощения формулировался для двух строк. На вход алгоритму подаются две строки P и T (см. листинг 12). Функция `scan patterns` ищет совпадающие подстроки длины не меньше s и сохраняет их в виде списка классов `Match`. Класс типа `Match` описывает диапазон (плитку), в нем хранятся: индексы начала подстрок и их длины. Функция `mark_strings` последовательно перебирает все элементы списка. Если рассматриваемый диапазон не перекрывается с уже размеченными, то функция маркирует элементы строк из этого диапазона и добавляет его в возвращаемый массив. Список диапазонов, полученный после работы функции `mark_strings` конкатенируется со списком `Tiles`. После этого значение s уменьшается, и процедура повторяется. Результатом работы алгоритма является список `Tiles`, содержащий неперекрывающиеся диапазоны совпадающих элементов.

Листинг 12, Алгоритм жадного строкового замощения.

```
class Match:
  int indexT
  int indexP
  int len
P,T = #INPUT
Tiles = []
for s=max_len; s>=min_len; s--:
  Matches = scan_patterns(P, T, s)
  Tiles += mark_strings(P, T, Matches)
return Tiles
```

После модификации данного алгоритма (см. листинг 13) его можно использовать для поиска наиболее подходящих фрагментов в поле зрения, для последующего использования. Теперь P и T — результатные выражения. Введем функцию сравнения двух узлов результатного выражения `Compare`, возвращающую вес перекрытия для двух элементов. В случае полного

совпадения узлов эта функция возвращает значение 3. Если у узлов совпадают типы, то функция возвращает 2. Иначе функция возвращает 1. Эти значения были выбраны таким образом, чтобы максимально задействовать участки, которые имеют наименьшую стоимость повторного использования.

В классе, описывающем диапазон, необходимо ввести новое поле, отвечающее за вес. Вес — это значение, полученное в результате сканирования подстрок. Вес описывает степень схожести двух диапазонов. Он обратно пропорционален вычислительной стоимости, то есть чем больше вес, тем меньше вычислительная стоимость коррекции одного выражения в другое. Для компилятора это означает то, что чем больше вес, тем меньше ресурсов будет задействовано для повторного использования рассматриваемого диапазона из терма конкретизации в результате. Для вычисления веса плитки используется формула (1).

$$\text{weight} = \sum_{i=0}^{len} \text{Compare}(T[\text{indexT} + i], P[\text{indexP} + i]), \qquad (1)$$

где weight — вес;

Compare — функция сравнения символов результатного выражения;

P и T — рассматриваемые объектные выражения;

indexP — индекс начала диапазона в P;

indexT — индекс начала диапазона в T;

len — длина диапазона.

В модифицированном алгоритме функция `scan_patterns` ищет все подстроки длины не меньше s в результатных выражениях P и T, после чего генерирует все возможные диапазоны, с вычисленным весом каждого из них, и помещает в список. Полученный на выходе функции `scan_patterns` список сортируется по убыванию поля `weight`. Поведение функции `mark_strings` ничем не отличается от случая со строками, она последовательно маркирует неперекрывающиеся диапазоны и сохраняет их в возвращаемый список.

Листинг 13, Модифицированный алгоритм жадного строкового замощения.

```
class Match:
  int indexT
  int indexP
  int len
  int weight
P,T = #INPUT
Tiles = []
for s=max_len; s>=1; s--:
  Matches = scan_patterns(P, T, s)
  Sort(Matches)
  Tiles += mark_strings(P, T, Matches)
return Tiles
```

К сожалению, почти для любого участка (с фиксированной длиной и весом) можно найти участок, имеющий меньшую длину, но больший вес. Это делает подобный поиск перекрытий малоэффективным. Более того в объектных выражениях языка Рефал, существуют объекты, которые не могут попарно перекрываться, а это нужно учитывать. В таблице 1 указаны веса перекрытий при попытке использовать X как Y. Переменные не могут повторно использоваться ни с изменением типа, ни с изменением значения ввиду специфичности их обработки. Ввиду того, что e-переменные и t-переменные сами по себе являются диапазонами, то есть последовательностью узлов, их повторное использование значительно усложнит и без того ресурсоемкий алгоритм. Хоть s-переменные и являются обычными узлами, переопределение такого узла грозит потерей данных. Дело в том, что переменная, узел которой переопределяется, может служить оригиналом для создания другой одноименной переменной (для создания повторной переменной или для повторного использования какого-то другого узла). Например, при преобразовании поля зрения вида `<F s.X s.Y> → <G s.Y s.X>` после предполагаемого преобразования `s.X` в `s.Y` исходное значение `s.X` будет потеряно и преобразовать `s.Y` в `s.X` уже не удастся. Анализ таких ситуаций сильно усложнит алгоритм, при этом, вряд ли будет добавлять существенный прирост производительности. Поэтому

преобразования узлов s-переменных разумно запретить. Стоит отметить, что обратная процедура допустима, и атом или скобку можно преобразовать в s-переменную, что не верно для других типов переменных. Так же стоит упомянуть о том, что плитка может состоять из единственной e-переменной, может содержать ее внутри, но в начале или в конце плитки e-переменная встречаться не должна. Для большей ясности требуется уяснить, что в данном контексте, под элементами плитки понимаются лексемы диапазона из результата, которым сопоставлены конкретные узлы из поля зрения.

Для получения более хорошего результата предлагается каждый раз выбирать не замаркированную плитку наибольшего веса, и выполнять ее маркировку. Это значительно увеличит алгоритмическую сложность, но в данной прикладной области такая проблема не столь существенна, так как оптимизируется время выполнения скомпилированного кода, а не время компиляции. На листинге 14 приведен псевдокод такой модификации. Функция `find_tile` ищет плитку с наибольшим весом. Она возвращает экземпляр класса `Match`, а также `true` или `false` в зависимости от успешности поиска. Функция `mark_tile` маркирует эту плитку.

Таблица 1, Веса поэлементных перекрытий.

X	**Y**			
	Атом	e, t	< > ( ) [ ]	s
Атом	3, 2, 1	0	1	1
s, t, e	0	0, 3	0	0, 3
< > ( ) [ ]	1	0	3, 1	1

Листинг 14, Модифицированный алгоритм жадного строкового замощения с выбором элемента, имеющего наибольший вес.

```
Tiles = []
while tile_exists:
```

```
  tile, tile_exists = find_tile(P, T)
  if tile_exists:
    mark_tile(P, T, tile)
    Tiles += tile
return Tiles
```

Поиск подвыражения с максимальным весом можно выполнять по алгоритму, представленному в листинге 15. Верхняя асимптотическая оценка сложность такого алгоритма $O(n^3)$. Данная реализация учитывает тот факт, что продление плитки при встрече перекрытия с нулевым весом невозможно.

Листинг 15, Поиск подвыражения с максимальным весом.

```
w_max = 0
for x in range(0, len(P)):
  for y in range(0, len(T)):
    l = 0
    w = 0
    while x+l < len(P) and
          y+l < len(T) and
          Compare(P[x+l], P[y+l]) > 0:
      w += Compare(P[x+l], T[y+l])
      l += 1
    if w > w_max
      w_max = w
      x_max = x
      y_max = y
      l_max = l
if w_max != 0:
  return (y_max, x_max, l_max, w_max), true
else:
  return Nothing, false
```

Верхняя асимптотическая оценка сложности такой модификации алгоритма жадного строкового замощения $O(n^4)$. Стоит еще раз отметить тот факт, что использование такой оптимизации значительно увеличит время компиляции программы, но при этом ускорит время выполнения скомпилированного кода.

Очевидно, что этот алгоритм не оптимален и может быть улучшен. При поиске элемента с наибольшим весом сохраняется информация лишь о максимуме, но в процессе поиска находятся и другие плитки, возможно неперекрывающиеся с максимальной. Эта информация теряется и алгоритм, в некоторых случаях, заново выполняет одну и ту же последовательность действий. Такая проблема может быть решена с помощью динамического программирования. Метод динамического программирования заключается в сохранении результатов решения подзадач для их дальнейшего использования. К сожалению, ввиду отсутствия индексации, выполняющейся за константное время, такая идея сложно реализуется в Простом Рефале.

Глава 4. Реализация

4.1 Разметка образца

Для реализации описанного выше алгоритма, первым делом необходимо подготовить входные данные, то есть разметить образцовое выражение. Для этого в функции `GenPattern`, расположенной в файле «HighLevelRASL.sref», были модифицированы соответствующие фрагменты кода. Каждому элементу (токену) образцового выражение было поставлено в соответствие смещение в массиве context, в узлах которого находятся указатели на соответствующие узлы двусвязного списка поля зрения. В данном контексте под элементами образцового выражения понимаются атомы (символы, идентификаторы, указатели на функции, числа) и скобки (конкретизационные, структурные, абстрактные). В процессе выполнения функции `GenPattern` генерируются команды RASL, которые впоследствии преобразуются в целевой код на языке C++. За генерацию целевого кода, по командам промежуточного представления, отвечает функция `GenCommand` из файла «Generator.sref». В файл «Generator.sref» была добавлена возможность обрабатывать команды, сохраняющие указатели на узлы в массиве context, а в файлы «refalrts.h» и «refalrts.cpp» соответствующие функции, вызываемые из сгенерированного по этим командам кода. Имена этих команд имеют суффикс `Save`. Например, `#CmdNumberSave`. Новые команды сопоставления принимают еще один дополнительный аргумент, а именно смещение в массиве `context`.

4.2 Реализация алгоритма жадного строкового замощения

Ввиду того, что алгоритм был реализован на функциональном языке, внешне он сильно отличается от псевдокода, приведенного ранее. Данная реализация принимает на вход размеченное образцовое выражение и не размеченное результатное выражение, а возвращает остатки образца и результат, в котором некоторые фрагменты были заменены плитками. Результат и образец, поданные на вход алгоритму, должны быть в плоском виде. Под плоским видом подразумевается формат выражения, в котором открывающиеся и закрывающиеся скобки заменены токенами, описывающими их. Данная оговорка присутствует по той причине, что изначально в функцию `HighLevelRASL-Sentence` выражения подаются в иерархическом (древовидном) формате.

Строковое представление плитки можно описать грамматикой из листинга 16. Далее представлены значения идентификаторов, использованных в грамматике: `#Tile` — начало плитки, `#AsIs` — токен используется без изменений, `#Reuse` — у токена нужно изменить значение, `#HalfReuse` — у токена нужно изменить тэг и значение.

Во всех токенах плитки присутствует смещение, взятое из размеченного образца. Именно это позволяет потом повторно использовать уже существующие узлы. В образце фрагменты, соответствующие плиткам в результате, заменяются на идентификаторы `#RemovedTile`.

Листинг 16, Строковое представление плитки.

```
t.Tile ::= (#Tile t.WrappedToken*).
t.WrappedToken ::= (s.Reuse t.Token).
s.Reuse ::= #AsIs | #Reuse | #HalfReuse.
```

В общем случае результатное выражение, которое получается на выходе алгоритма жадного строкового замощения, все еще может содержать некоторые токены без смещений, расположенные между плитками. Поэтому перед генерацией команд RASL, необходимо проставить эти смещения. Для

таких элементов впоследствии будут сгенерированы команды создания новых узлов или команды копирования переменных.

На листинге 17 приведен пример работы реализованного алгоритма. На вход алгоритм принял размеченный образец MARKED_PATTERN и неразмеченный результат RESULT. На основе входных данных был сгенерирован результат NEW_RESULT, с вставленными плитками, и остаток образца TRASH, с вырезанными. Для каждого элемента после символа прямой косой черты указывается смещение в массиве context. В данном примере были задействованы не все узлы из образца, несмотря на то, что между плитками в результате все еще остались некоторые объекты. Это произошло по той причине, что атомы не могут быть повторно использованы как t-переменные и вес такого перекрытия равен нулю. Именно по этой причине алгоритм не включил такие перекрытия в плитки.

Листинг 17, Пример работы алгоритма.

```
MARKED_PATTERN:
</0 & Test/4 t.X#1/5 t.Y#1/7 t.X#1/9 t.Y#1/11 >/1
RESULT:
t.X t.X t.Y t.Y t.X t.Y
TRASH:
{REMOVED TILE}
</0 & Test/4
{REMOVED TILE}
{REMOVED TILE}
>/1
{REMOVED TILE}
NEW_RESULT:
Tile{ [[ }
Tile{ AsIs: t.X#1/5 }
t.X#1/9/13 t.Y#1/11/15
Tile{ AsIs: t.Y#1/7 AsIs: t.X#1/9 AsIs: t.Y#1/11 }
Tile{ ]] }
```

4.3 Генерация команд RASL

За генерацию команд RASL по предложению отвечает функция HighLevelRASL-Sentence, расположенная в файле «HighLevelRASL-

OptResult.sref» (см. листинг 18). На вход она принимает имя функции, неразмеченную левую и правую части предложения, а возвращает набор команд, по которым потом генерируется целевой код, отвечающий за обработку данного предложения. Функция GenPattern предназначена для генерации команд сопоставления с аргументом функции. Она принимает на вход терм конкретизации (#CallBrackets (e.Name) e.Pattern), а возвращает таблицу переменных e.PatternVars, размеченный образец e.MarkedPattern, объем использованного пространства памяти на стеке s.ContextOffset и последовательность команд e.PatternCommands сопоставления в промежуточном коде. Функция GST выполняет поиск схожих диапазонов. На вход она принимает размеченный образец ((#LEFT-EDGE) e.MarkedPat (#RIGHT-EDGE)) и неразмеченный результат в плоском виде ((#LEFT-EDGE) <FlatRes e.Res> (#RIGHT-EDGE)), а возвращает остатки образца e.Trash и результат с вставленными плитками e.MarkedResult. Стоит отметить, что по краям образца и результата были добавлены специальные элементы #LEFT-EDGE и #RIGHT-EDGE, которые фиксируют начало и конец выражений. Элемент #LEFT-EDGE, как и RIGHT-EDGE, может перекрываться только с самим собой. По этой причине они всегда будут входить в плитки, расположенные в начале и в конце. Функция AddOffsets добавляет смещения в фрагменты результата, расположенные между плиток. Функция GenResult отвечает за генерацию команд изменения поля зрения в процессе построения результатного выражения из правой части правила. На вход она принимает остатки образца и полностью размеченный результат, а возвращает набор команд.

Листинг 18, Функция HighLevelRASL-Sentence.

```
HighLevelRASL-Sentence {
 e.Name (e.Pattern) (e.Result) =
  <Fetch
   <GenPattern (#CallBrackets (e.Name) e.Pattern)>
   {
    (e.PatternVars) (e.MarkedPattern)
```

```
    s.ContextOffset (e.PatternCommands) =
     <Fetch
      <GST
       ((#LEFT-EDGE) e.MarkedPat (#RIGHT-EDGE))
       ((#LEFT-EDGE) <FlatRes e.Res> (#RIGHT-EDGE))
      >
      {
       (e.Trash)(e.MarkedResult) =
        <Fetch
         <AddOffsets
          s.ContextOffset
          (e.PatternVars)
          e.MarkedResult
         >
         {
          s.ContextOffset^ e.MarkedResult^ =
           <FoldOpenELoops
            (#CmdIssueMem s.ContextOffset)
            (#CmdInitB0-Lite) e.PatternCommands
            (#CmdResetAllocator)
            <GenResult (e.Trash) (e.MarkedResult)>
           >;
         }
        >;
      }
     >;
   }
  >;
}
```

Для упрощения отладки, первым делом, были сгенерированы команды добавления комментариев в целевой код. Такие команды обозначаются идентификатором `#CmdComment`. Впоследствии эти команды были заменены на настоящие:

1. `#CmdCopyVar` — копирует переменную из образца в результат.
2. `#CmdAllocateElem` — создает новый элемент.
3. `#CmdReinitElem` — изменяет тэг и значение узла.
4. `#CmdUpdateElem` –изменяет значение узла.
5. `#CmdPushStack` — добавляет открывающую скобку в стек скобок.

6. `#CmdLinkBracket` — линкует закрывающую скобку со скобкой, снятой с вершины стека скобок.
7. `#CmdInsertTile` — переносит диапазон из списка свободных узлов в поле зрения.
8. `#CmdTrash` - обрабатывает неиспользованные узлы в образце и возвращает результат.
9. `#CmdReturnResult-NoTrash` - возвращает результат, при отсутствии неиспользованных узлов.

4.4 Генерация целевого кода

По командам промежуточного представления генерируется целевой код на языке C++. Проиллюстрировать работу программы можно с помощью листингов 19 и 20, в них показан целевой код, который генерируется для первого предложения функции Fab без оптимизации и с оптимизацией соответственно. В оптимизированном коде для каждого элемента из образца сохраняется указатель в массив `context`. Можно заметить, что в листинге 19 преобладают команды создания новых элементов `alloc` и переносов `splice`, в то время как в листинге 20 такие команды отсутствуют. Вместо них генерируются команды повторного использования `reinit`. Также можно заметить, что скобки конкретизации и указатель на функцию в текущей версии просто игнорируются и пропускаются посредством команд `move_left` и `move_right`. В оптимизированном компиляторе эти элементы используются как полноценные части образца, которые можно повторно использовать в результатном выражении.

Листинг 19, Целевой код первого предложения функции Fab полученный при компиляции без режима оптимизации.

```
do {
  refalrts::start_sentence();
  context[0] = arg_begin;
```

```
  context[1] = arg_end;
  refalrts::move_left(context[0], context[1]);
  refalrts::move_left(context[0], context[1]);
  refalrts::move_right(context[0], context[1]);
  if(!refalrts::char_left('A', context[0], context[1]))
    continue;
  refalrts::reset_allocator();
  refalrts::Iter res = arg_begin;
  if(!refalrts::alloc_char( context[2], 'B' ))
    return refalrts::cNoMemory;
  if(!refalrts::alloc_open_call( context[3] ))
    return refalrts::cNoMemory;
  if(!refalrts::alloc_name( context[4], Fab, "Fab" ))
    return refalrts::cNoMemory;
  if(! refalrts::alloc_close_call( context[5] ))
    return refalrts::cNoMemory;
  refalrts::push_stack(context[5]);
  refalrts::push_stack(context[3]);
  res = refalrts::splice_elem(res, context[5]);
  res = refalrts::splice_evar(
    res, context[0], context[1]);
  res = refalrts::splice_elem(res, context[4]);
  res = refalrts::splice_elem(res, context[3]);
  res = refalrts::splice_elem(res, context[2]);
  refalrts::use(res);
  refalrts::splice_to_freelist(arg_begin, arg_end);
  return refalrts::FnResult(
    refalrts::cSuccess | (__LINE__ << 8));
} while (0);
```

Отладка добавленных команд производилась на специально подобранных unit-тестах. Одним из таких тестов, и как следствие самым большим, можно считать исходный код компилятора, написанный на Простом Рефале. Подготовленные в процессе отладки тесты также были добавлены в проект компилятора. В дальнейшем они помогут избежать регрессий, при модификации написанного кода.

Листинг 20, Целевой код первого предложения функции Fab полученный при компиляции с режимом оптимизации.

```
do {
  refalrts::start_sentence();
```

```
  context[0] = arg_begin;
  context[1] = arg_end;
  // </0 & Fab/4 'A'/5 e.X#1/2 >/1
  context[2] = 0;
  context[3] = 0;
  context[4] = refalrts::call_left(
    context[2], context[3], Fab, context[0],
context[1]);
  context[5] = refalrts::char_left('A',context[2],
    context[3]);
  if(!context[5]) continue;
  refalrts::reset_allocator();
  //TRASH: {REMOVED TILE}
  //RESULT: Tile{ [[ HalfReuse: 'B'/0 HalfReuse:
  //</4 HalfReuse: & Fab/5 AsIs: e.X#1/2 AsIs: >/1 ]] }
  refalrts::reinit_char(context[0], 'B');
  refalrts::reinit_open_call(context[4]);
  refalrts::reinit_name( context[5], Fab, "Fab" );
  refalrts::push_stack( context[1] );
  refalrts::push_stack( context[4] );
  return refalrts::FnResult(refalrts::cSuccess |
    (__LINE__ << 8));
} while ( 0 );
```

Глава 5. Тестирование

Компилятор Простого Рефала является переносимым. Следовательно, замерять производительность можно как для операционных систем Linux, так и для Windows, при этом также допустимы тесты для разных компиляторов C++.

Для проверки прироста производительности можно воспользоваться лексическим анализатором компилятора Простого Рефала. Скомпилировав его с оптимизацией и без, а потом, запустив для одинаковых входных данных, можно получить наглядные результаты прироста производительности. Подав на вход лексическому анализатору все исходные файлы компилятора, были получены значения, указанные в таблице 2. Здесь «Opt» и «Non-opt» показывают, что тестировалась оптимизированная или неоптимизированная программа соответственно. Основываясь на полученных данных, можно сделать вывод, что оптимизированный лексический анализатор работает примерно на 14 процентов быстрее. Однако стоит отметить, что эта цифра, описывающая эффект оптимизации, не является инвариантом, то есть она напрямую зависит от того, какая программа компилировалась с оптимизацией.

Таблица 2, Результаты теста лексического анализатора.

Processor OS compiler	**Non-opt**	**Opt**
Core i7 Ubuntu 14.04 GCC	0.870 с	0.767 с
Core i5 Windows 10 GCC	0.769 с	0.651 с
Core i7 Ubuntu 14.04 Clang	0.940 с	0.796 с

Ввиду того, что компилятор является самоприменимым, его можно скомпилировать с оптимизацией и без оптимизации, оптимизированным компилятором и неоптимизированным. Результаты такого теста приведены в таблице 3. В ней «Mod = OR» означает тот факт, что компиляции происходила в режиме оптимизации результата. После такой компиляции получается оптимизированный компилятор, который работает быстрее. В данном случае уменьшение времени выполнения за счёт оптимизации примерно 8 процентов. При этом компиляция в режиме оптимизации выполняется почти в 8 раз дольше, что не удивительно, так как использовался алгоритм со сложностью $O(n^4)$, где n — количество значащих символов исходного кода.

Таблица 3, Результаты теста компиляции компилятора.

Processor **OS** **compiler**	**Non-opt**	**Non-opt** **+** **Mod = OR**	**Opt**	**Opt** **+** **Mod = OR**
Core i7 Ubuntu14.04 GCC	2.980 с	24.198 с	2.737 с	19.378 с
Core i5 Windows 10 GCC	2.163 с	18.031 с	1.897 с	13.278 с
Core i7 Ubuntu 14.04 Clang	4.358 с	34.173 с	4.102 с	29.033 с

Заключение

Подводя итоги, можно сказать, что в данной работе была подробно рассмотрена одна из возможных реализаций компилятора РЕФАЛа. Такой опыт, безусловно, может быть полезен при дальнейшей работе с компиляторами.

Задачи, поставленные ранее, были успешно решены, а именно: была разработана и реализована оптимизация построения результатных выражений для компилятора функционального языка программирования, называемого Простым Рефалом. Реализацию оптимизации можно найти в отдельной ветке «asadir-or» репозитория компилятора Простого Рефала [3].

В процессе разработки оптимизации были предложены разные алгоритмы. Некоторые из них обладали меньшей алгоритмической сложностью, другие давали более качественный результат. Реализованный алгоритм является некоторым компромиссом между качеством оптимизации и скоростью её выполнения. Уменьшение времени выполнения целевого кода, полученного с помощью оптимизированного алгоритма, зафиксировано и составляет от 8 до 14 процентов в зависимости от формата исходного кода. Компиляция в режиме оптимизации выполняется значительно дольше (приблизительно в 7-8 раз). Это происходит из-за использования алгоритма, имеющего полиномиальную сложность четвертой степени. Однако такая проблема не критична, так как объем исходного кода редко исчисляется в шестизначных числах. Поэтому можно утверждать, что полученная реализация оптимизации является применимой на практике.

Дальнейшее развитие поднятой в данной работе темы может иметь несколько различных направлений. Так как используемый метод, для поиска общих диапазонов, является субоптимальным, то допустим подбор более эффективного, в данной предметной области, алгоритма. Таким образом, можно добиться как увеличения скорости компиляции, так и улучшения качества целевого кода. Также есть возможность внедрить разработанную

оптимизацию в другие диалекты РЕФАЛа или в другие функциональные языки программирования, использующие списковую реализацию. Для этого, скорее всего, потребуется доработка алгоритма, однако главные идеи останутся теми же.

Литература

1. Романенко С. А. Машинно-независимый компилятор с языка рекурсивных функций, 1978.
2. Турчин В. Ф. Пользовательская документация для языка РЕФАЛ-5, URL: http://www.refal.net/rf5_frm.htm (дата обращения: 21.06.2016).
3. Коновалов А. В. Пользовательская документация для языка Простой Рефал, URL: https://github.com/Mazdaywik/simple-refal (дата обращения: 21.06.2016).
4. Nagy G. and Lopresti D. Form similarity via Levenshtein distance between ortho-filtered logarithmic ruling-gap ratios, Procs. SPIE/IST Document Recognition and Retrieval, Feb. 2001.
5. Damerau F. A technique for computer detection and correction of spelling errors. // Comm. of the ACM, 1964
6. Wise M.J. String similarity via greedy string tiling and running Karp-Rabin matching. // Dept. of CS, University of Sidney, December 1993.