Аннотация.

Цель выпускной квалификационной работы — разработка и реализация алгоритма оптимизации операции сопоставления выражения с образцом в компиляторе языка Простой Рефал. В работе приведено описание языка Простой Рефал и общей структуры его компилятора, обоснована необходимость внедрения оптимизации. Предложен метод оптимизации, разработан и обоснован алгоритм. Созданы модули компилятора, реализующие разработанный алгоритм и проведены эксперименты, подтверждающие применимость и эффективность выбранного метода оптимизации.

Оглавление.

Обозначения и сокращения.

РЕФАЛ, Рефал — Рекурсивных Функций Алгоритмический язык.

AST — abstract syntax tree, абстрактное синтаксическое дерево.

RASL – Refal ASsembly Language, язык сборки Рефала, промежуточное представление при компиляции.

ЛСО — локальное сложнейшее обобщение.

ГСО — глобальное сложнейшее обобщение.

БО — быстрое обобщение.

Введение.

РЕФАЛ — РЕкурсивных Функций АЛгоритмический язык, один из старейших языков функционального программирования, созданный в 1966 году Валентином Турчиным в качестве метаязыка для описания семантики других языков. Язык ориентирован на символьные вычисления: преобразование текстов, обработку символьных строк, перевод с одного языка (искусственного или естественного) на другой, решение проблем, связанных с искусственным интеллектом. Он соединяет в себе математическую простоту с практической ориентацией на написание больших и сложных программ. В 1970-1990 были реализованы несколько различных диалектов этого языка. В МГТУ имени Н. Э. Баумана на кафедре ИУ9 компилятор этого языка используется в качестве учебного полигона для курсовых и дипломных работ.

Синтаксис РЕФАЛа не позволяет функциям принимать более одного аргумента-объектного выражения. Если в функцию необходимо передать несколько аргументов, то из них формируется единое объектное выражение, которое в теле функции разбирается на части, соответствующие входным аргументам. По этой причине часто предложения в одной функции имеют схожую структуру образцовых частей, из-за чего операции анализа аргумента функции выполняются несколько раз. Естественно, это не эффективно, особенно в случае рекурсивного вызова подобных функций.

Таким образом, цель данной работы — разработка и реализация алгоритма оптимизации обработки образцовых выражений. В первую очередь, нужно исследовать предметную область, выучить язык Рефал, провести поиск существующих решений поставленной проблемы. Для выбранного алгоритма необходимо обосновать применимость и доказать эффективность.

Алгоритм должен быть реализован как часть компилятора языка Рефал. Необходимо заменить генерацию команд сопоставления для каждого предложения в пределах одной функции на генерацию команд для списка всех предложений. Так можно добиться появления связи/зависимости между образцовыми частями предложений. Также следует выделить общие части

образцов или команд, избавиться от повторов. Это позволит избежать избыточных проверок и ускорить работу программы. После завершения реализации, следует провести тестирование и анализ его результатов. Эксперименты следует проводить на разных компьютерах, на разных операционных системах, на разных компиляторах языка C++. Помимо этого следует исследовать зависимость эффективности оптимизации от структуры компилируемого файла.

Глава 1. Обзор предметной области.

1.1. Общие понятия языка РЕФАЛ.

Существует несколько разновидностей языка Рефал. Некоторые из них похожи, например, Простой Рефал и РЕФАЛ-5. В документации диалекта РЕФАЛ-5 [2] определен набор базовых понятий (так же называемых понятиями Базисного Рефала), использующихся во многих других диалектах языка, поэтому общие понятия можно рассмотреть на примере этого диалекта.

Основные понятия Базисного Рефала:

- Атом — минимальная структурная единица, чей вид, как правило, определяется диалектом. Это может быть идентификатор, имя, число, символ и т. д.
- Терм – это или атом, или выражение в структурных (круглых) скобках.
- Свободные переменные – это s-переменные (атомы), t-переменные (термы) и e-переменные (выражения). Переменная в выражении состоит из 3 элементов: указателя типа, точки и индекса. Индекс переменной — последовательность букв и цифр. Переменные одного типа с одинаковыми индексами называются повторными и имеют одинаковые значения. Подробнее о переменных в пункте 1.2 настоящей работы.

  Примеры: s.Letter, e.Eval1, t.2016.
- Объектное выражение – это выражение, содержащее только атомы и структурные и абстрактные скобки (о видах скобок в пункте 1.2 настоящей работы).

  Примеры: ’Hello, world!’, (12+23)*34 – объектные выражения. ’A’ s.Mid ’B’ – НЕ объектное выражение.
- Образцовое выражение (образец) – это выражение, содержащее атомы, структурные и абстрактные скобки и свободные переменные. Образец может рассматриваться, как синтаксическая конструкция, описывающая множество объектных выражений. Фактически, образец является объектным выражением, где некоторые (или все) части заменены на свободные переменные.

Примеры: ’!’, 12 , ’A’ s.Mid ’B’, e.Begin (e.InBrackets) e.End

- Результатное выражение (результат) – это выражение, содержащее атомы, свободные переменные, структурные и абстрактные скобки и скобки конкретизации (скобки вызова функций).

  Примеры: ’!’, 12 , ’A’ s.Mid ’B’, <Func e.Arg1 s.Arg2>
- Конкретизация — вызов функции. При этом функция определяется именем, указанным сразу после открывающей скобки, а аргументом является последовательность символов после имени до закрывающей скобки. Скобки конкретизации могут присутствовать только в результатных выражениях.

На листинге 1 видно, что функция в РЕФАЛе-5 (как и в некоторых других диалектах) состоит из предложений, разделенных точкой с запятой. Каждое предложение состоит из левой, называемой образцом, и правой, называемой результатом, частей. Очевидно, образец может быть представлен только образцовым выражением, а результат — результатным.

Листинг 1. Пример программы на языке Рефал-5.

```
SameString {
  s.Letter e.Tail1 '$' s.Letter e.Tail2 =
    <SameString e.Tail1 '$' e.Tail2>;
  '$' = #Success;
}
```

В примере листинга 1 приведена функция, определяющая совпадают ли 2 строки, разделенные знаком доллара `'$'`. Если обе строки начинаются с одной буквы, функция вызывается рекурсивно для «хвостов» строк. Сравнение удачно, если в определенный момент получен только разделяющий символ, если же строки различаются, функция прервёт выполнение с ошибкой распознавания.

1.2. Простой Рефал.

Простой Рефал является одним из диалектов Базисного Рефала. Язык не поддерживает расширенные конструкции, такие как условия, за счет чего

является подмножеством Базисного Рефала. При этом существует возможность объявления безымянных вложенных функций (лямбда-функций). Компилятор Простого Рефала является кроссплатформенным за счёт своего устройства. Он транслирует исходные тексты на Рефале в исходные тексты на C++, а получившийся набор файлов преобразуется компилятором C++, установленном на используемой машине, в объектные и исполняемые файлы. Компилятор самоприменимый, написан на Простом Рефале, рантайм-библиотека написана на языке C++. Исходный код компилятора находится в открытом доступе[3]. За счет этого существует возможность разработки компилятора большим сообществом программистов, что, в свою очередь, говорит о перспективах развития.

Переменные в языке Простой Рефал делятся на 3 типа: s, t, и e. Они отличаются областями допустимых значений. S-переменная может заменять атом. В рассматриваемом диалекте атомами могут быть символы, числа, имена (указатели на функции) и идентификаторы (глобальные константы). Идентификаторы начинаются с символа решетки (`#Ident`), а имена определяются после директив `$ENUM` или являются именем локальной или внешней функции (`$ENUM EmptyFunc`).

T-переменная может являться термом: s-переменной, атомом или выражением в структурных (круглых) или абстрактных (квадратных) скобках. Абстрактные скобки обязательно помечены именем функции(`[FuncName e.Body]`), что делает их уникальными. В программе их можно использовать для выделения, например, особенных частей образца.

E-переменные заменяют последовательность атомов, термов или переменных, в том числе пустую последовательность — ε. Они являются самым гибким типом переменных, но потому и самым сложным при сопоставлении. В отличии от s- и t-переменных, длина которых всегда равна одному терму, e-переменная может быть любой длины.

Важным понятием Простого Рефала является процедура сопоставления

выражения с образцом, называемая так же проецированием выражения на образец. Сопоставление объектного выражения E с образцовым выражением P – это поиск значений переменных образца P, при замене переменных на которые будет получено выражение E. Если вариантов такого сопоставления несколько, то выбирается тот, в котором первая e-переменная принимает кратчайшее значение. Если это не решает конфликта, то первая e-переменная фиксируется в своем кратчайшем значении, и рассматривается вторая e-переменная и т. д. Это позволяет получить однозначное сопоставление.

Пример. Для образца $P = e.1\,()\,e.2\,()\,e.3$ и объектного выражения $E = 'a'\,()\,'b'\,()\,'c'\,()\,'d'$ проецирование произойдет следующим образом:

$e.1 \rightarrow$ ‘a’ ,

$e.2 \rightarrow$ ‘b’ ,

$e.3 \rightarrow$ ‘c’$()$‘d’ .

E-переменные могут быть открытыми и закрытыми. Закрытой называется e-переменная, значение которой при сопоставлении с любым выражением можно определить однозначно. Открытой, наоборот, называется e-переменная, значение которой нельзя определить однозначно при сопоставлении с каким-либо выражением. Например, в образцовом выражении $e.B\,s.M\,e.E$ переменная $e.B$ — открытая, потому что, например, при сопоставлении с выражением ABC, состоящим из 3 атомов, она может принимать 3 различных значения: $e.B = \varepsilon$ при $s.M = A, e.E = BC$, $e.B = A$ при $s.M = B, e.E = C$, $e.B = AB$ при $s.M = C, e.E = \varepsilon$. Та же переменная $e.B$ в образце $e.B\,s.M$ является закрытой, так как для любого образца будет являться результатом отделения справа атома от заданного выражения.

Проецирование для разных образцов производится за разное время и имеет разную сложность. Например, для атомов, пары скобок и s-переменных сопоставление выполняется за константное время, а для открытых e-переменных это время зависит от длины проецируемого выражения, при этом количество переменных в нем определяет степень полинома зависимости.

Повторные переменные требуют рекурсивного сравнения выражений, его сложность зависит линейно от количества термов, входящих в сравниваемые выражения.

Программа на Простом Рефале состоит из последовательности функций. Функция Простого Рефала определяется грамматикой, представленной в листинге 2. Функция состоит из предложений. При вызове функции, выполняется попытка сопоставить входное выражение (аргумент) с левой частью (с образцом) первого предложения. При удачном сопоставлении аргумента с образцом, функция возвращает результат, сформированный на основе переменных из образца. Если сопоставление не удалось, производится попытка сопоставить аргумент с образцом следующего предложения. Если следующего предложения нет, то есть не удалось сопоставление с последним предложением функции, программа останавливается и выводит сообщение об ошибке распознавания Recognition Impossible.

Листинг 2. Грамматика функции языка Простой Рефал

```
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 |
  "#" IDENT.
```

Указанная грамматика для упрощения не показывает, что конкретизационные скобки (скобки вызова функции) могут присутствовать только в результатном выражении, поэтому приведенную грамматику следует понимать с этой оговоркой.

Символ `'^'`, символ переопределения переменной, позволяет использовать ранее занятые названия (индексы) переменных. Помеченные этим символом в образце переменные перезаписываются при сопоставлении. Если же переменная имеет ранее использованное имя без символа переопределения, то при сопоставлении с выражением ее значение будет считаться известным, что может привести к неправильному распознаванию или даже к ошибке.

Компилятор поддерживает несколько режимов работы. Используемый режим определяется флагами, указанными при компиляции. При отсутствии флагов производится прямая кодогенерация. При указании флага `-INTERPRET` код генерируется только в интерпретируемом режиме, что уменьшает его размер примерно на треть, но замедляет выполнение программы[1]. Существует 2 флага оптимизаций, `-OP` и `-OR`. Они не совместимы с режимом интерпретации. Так как работа компилятора в режиме оптимизирования несколько отличается от стандартной, исходный код оптимизирующей версии несколько отличается. В частности, вместо модуля `HighLevelRASL.sref` используются модули `HighLevelRASL-OptPattern.sref` для флага `-OP` и `HighLevelRASL-OptResult.sref` для флага `-OR` (подробнее об устройстве компилятора в пункте 1.3. настоящей работы). В рамках данной работы будет производится написание модуля `HighLevelRASL-OptPattern.sref`.

Для Простого Рефала существует набор библиотечных функций, расположенных в файлах «Library.sref» и «LibraryEx.sref» [3], таких как `WriteLine` (выводит свой аргумент в консоль), `Add` (возвращает сумму своих аргументов) и так далее. Их назначение, чаще всего, понятно из названия.

1.3. Структура компилятора Простого Рефала.

Компилятор Простого Рефала многопроходный. Общая структура работы

компилятора изображена на рисунке 1.

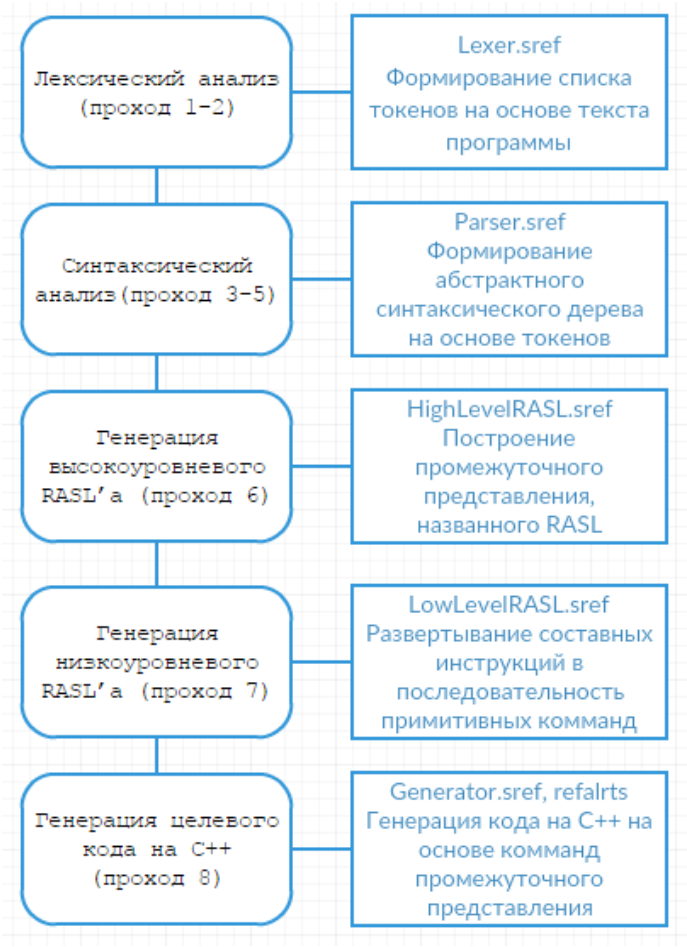

Рисунок 1: Структура работы компилятора Простого Рефала

Первыми проходами проводится лексический анализ. На этом этапе с помощью модуля `Lexer.sref` из файла считывается текст программы, генерируется последовательность токенов, которые приводятся к общему виду, используемом на других стадиях компиляции. Токены, полученные на выходе имеют структуру, показанную в листинге 3.

Листинг 3. Формат токенов лексического анализатора

```
e.Tokens ::= (s.TokType s.LineNumber e.Info)*
```

Здесь `s.TokType` – тип токена, `s.LineNumber` – номер строки, `e.Info` – значение токена, в зависимости от типа.

Вторым этапом является синтаксический анализ. Выполняемый модулем `Parser.sref`, синтаксический анализ методом рекурсивного спуска расширяет список ошибок, обнаруженных лексером, и строит таблицу символов и абстрактное синтаксическое дерево. Абстрактное синтаксическое дерево (Abstract Syntax Tree, аббревиатура AST) — древовидная структура, в которой хранится информация о функциях и предложениях, входящих в эти функции. Его структура показана в листинге 4. Образцовые и результатные выражения отображены, как одинаковые структурные элементы, чтобы избежать лишнего усложнения грамматики, но на практике образцы не могут содержать скобки вызова, то есть элемент `(#CallBrackets e.Expression)` не может содержаться в элементе `e.Pattern`.

Листинг 4. Грамматика абстрактного синтаксического дерева

```
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 ::= ((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)
```

Далее модулем `HighLevelRASL.sref` (или аналогами при оптимизации) по абстрактному синтаксическому дереву строится промежуточное

представление, называемое высокоуровневый RASL. RASL представляет собой последовательность команд различного типа. Именно на этом этапе следует проводить оптимизации. В частности, в этом модуле для каждой функции из дерева вызывается функция `HighLevelRASL-Function`, отвечающая за генерацию команд RASL для функций и её предложений. В полученном промежуточном представлении каждой функции соответствует команда `(#Function s.ScopeClass (e.Name) e.HiCommands)`, а для пустой функции `(#CmdEnum s.ScopeClass e.Name)`, где `e.Name` — имя функции, `e.HiCommands` — команды, соответствующие предложениям функции. Полная структура RASL'а после выполнения прохода генерации высокоуровнего RASL'а представлена в Приложении 1. Список команд сопоставления, являющихся, в некотором смысле, основой любой программы на Простом Рефале представлен в листинге 5.

Листинг 5: Фрагмент грамматики HighLevelRASL'a, отвечающий за команды сопоставления

```
s.MatchCommand e.MatchInfo ::=
    #CmdBrackets s.NewBracketsOffset
  | #CmdADT s.NewBracketsOffset e.Name
  | #CmdNumber s.Number
  | #CmdIdent e.Name
  | #CmdChar s.Char
  | #CmdName e.Name
  | #CmdRepeated s.Mode s.VarOffset s.SampleOffset
  | #CmdEmpty
  | #CmdVar s.Mode s.VarOffset
  | #CmdCharSave s.Offset s.Char
```

Проход модуля `LowLevelRASL.sref` осуществляет развёртывание составных инструкций промежуточного кода в более примитивные команды и формирование интерпретируемого кода (при компиляции в режиме интерпретации). В зависимости от переданных опций, осуществляется прямая кодогенерация или генерация интерпретируемого кода с вызовом интерпретатора. Полученное представление называется низкоуровневый RASL и незначительно отличается от высокоуровнего.

Последний проход — генерация кода на C++. Во время него на основе RASL генерируется целевой код, с помощью функций модуля `Generator.sref`. При генерации каждая команда промежуточного представления преобразуется в блок кода на C++. Функции, используемые в сгенерированном коде, объявляются в файлах „`refalrts.h`“ и „`refalrts.cpp`“ — рантайм-библиотеке компилятора.

Глава 2. Разработка алгоритма.

2.1. Обзор существующих решений.

В диссертации 1978 года С. А. Романенко «Машинно-независимый компилятор с языка рекурсивных функций»[4] был описан один из возможных способов оптимизации.

В работе Романенко рассматривает компилятор языка Рефал с промежуточным языком сборки, схожим по структуре с RASL'ом компилятора Простого Рефала. На определенном этапе компиляции программа переводится на язык сборки. Романенко проводит оптимизацию после этого перевода. Сначала каждое предложение независимо переводится на язык сборки. Получившиеся последовательности команд образуют дерево, где корнем является начало выполнения функции, каждое ребро отмечено одной из команд языка сборки (рисунок 1). При этом самая левая ветка отображает команды a_{1i} первого предложения, а последняя — команды a_{ni} последнего (n-го) предложения.

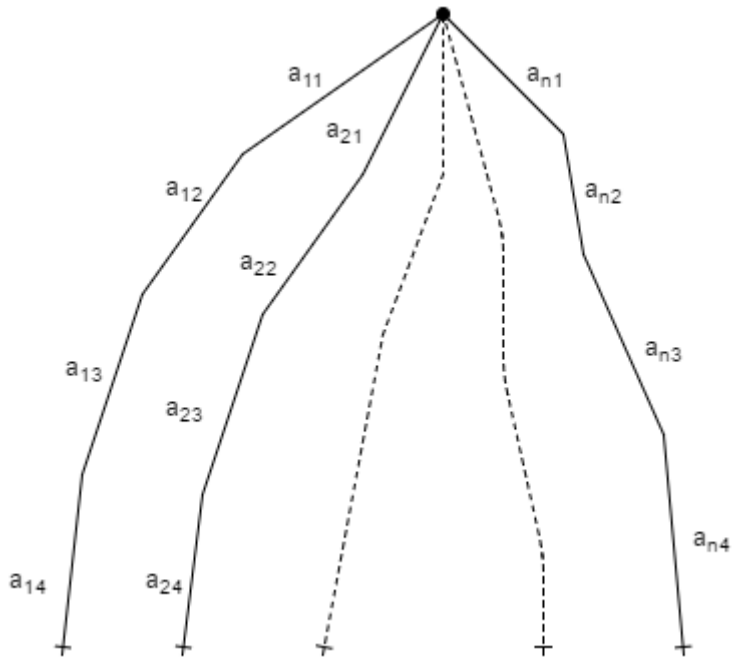

Рисунок 2: Дерево команд языка сборки

Программа начинает выполняться с корня дерева. Если команда успешно

выполнена, то выполнение спускается по направлению соответствующего ребра до ближайшего узла, а затем переходит на самое левое ребро, выходящее из этого узла. Если команда не выполнилась, то выполнение поднимается до ближайшего узла против направления соответствующего ребра, а затем переходит на ребро, выходящее из этого узла и следующего за ребром, по которому выполнение перешло ранее. Если такого ребра нет, то выполнение опять поднимается до ближайшего сверху узла и процесс повторяется.

После составления дерева команд языка сборки проводится исследование и оптимизация дерева. Если 2 соседних (между ними нет ребер) ребра, выходящих из одного узла, отмечены одной и той же командой, то производится их объединение. Процесс изображен на рисунке 2. Здесь a – совпадающая команда, X_1 и X_2 – поддеревья команд языка сборки. При выполнении объединения поддеревьев выполняется небольшое изменение, добавление команды передачи управления, позволяющее программе понимать, где закончилось одно поддерево и началось следующее.

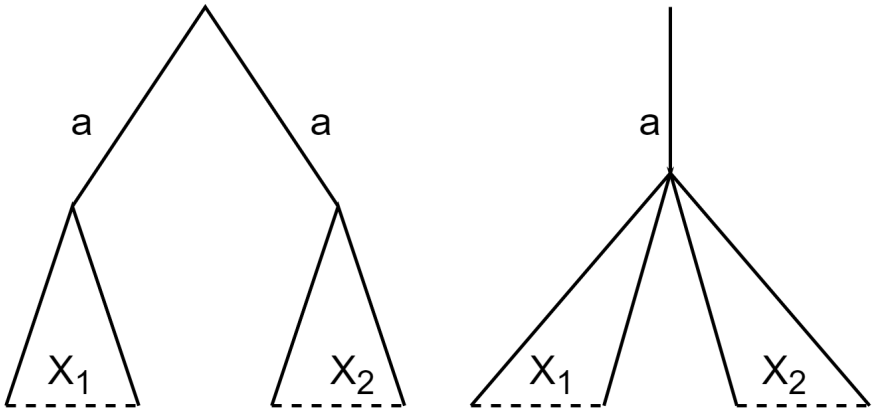

Рисунок 3: Объединение одинаковых ребер дерева

После проведения исследования и изменения дерево команд становится менее «ветвистым» (рисунок 4), что, определенно, ускоряет работу скомпилированной программы. Но так как данная оптимизация не изменяет проводимые компилятором операции, а только дополняет их, то компиляция замедляется. На рисунке a, b, b_1, b_2, c_{11}, c_2, c_{31}, c_{32} – команды языка сборки, X_1,

X_2, X_3, X_4, X_5 – поддеревья.

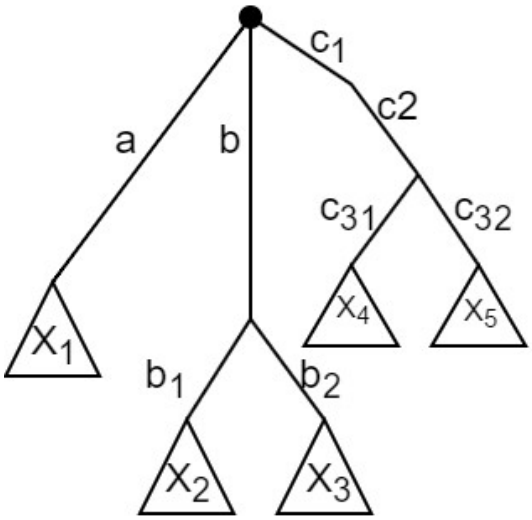

Рисунок 4: Оптимизированное дерево команд языка сборки

2.2. Необходимые определения.

После анализа существующих решений поставленной задачи, был выбран алгоритм оптимизации, для описания которого требуется ввести следующие определения и понятия:

- Жесткий образец — выражение без открытых и повторных переменных. Это значит, что жесткий образец на каждом уровне вложенности скобок содержит не более одной e-переменной, а также попарно различны индексы всех переменных[5].
- Сложность образца — числовая характеристика образца, определяемая по формуле

  $$C(P)=n_t+2n_s+3n_X+3n_{()}-n_e+1 \quad ,$$

  где n_t, n_s, n_e – количество, соответственно, t-, s-, e-переменных, $n_{()}$ – количество пар скобок(как круглых, так и абстрактных), n_X — количество атомов.
- Подстановка — набор замен переменных в образце на некие образцы,

  $$S=\{ v_i \to P_i, i=1,...,N \} \quad .$$

  Запись $P_1 \overset{S}{\to} P_2$ означает, что P_2 получен из P_1 подстановкой S.
- Уточнение образца — операция замены переменных в образце. Образец

P_2 уточняет образец P_1, если существует подстановка S, переводящая P_1 в P_2, $P_1 \Rightarrow^+ P_2$. При этом если $P_1 = P_2$ или $P_1 \Rightarrow^+ P_2$, то имеет место нестрогое уточнение, $P_1 \Rightarrow^* P_2$. Очевидно, для любого образца P верно $e \Rightarrow^* P$.

- Обобщение образца — операция, обратная к уточнению. $P_1 \Rightarrow^+ P_2$ - P_1 обобщает P_2 , $P_1 \Rightarrow^* P_2$ - P_1 нестрого обобщает P_2 . Возможна и другая запись: $P_2 \Leftarrow^* P_1$.
- Пусть $P_1 \Rightarrow^+ P_2$, при этом не существует такого образца P_3 , что $P_1 \Rightarrow^+ P_3 \Rightarrow^+ P_2$. Тогда $P_1 \Rightarrow^+ P_2$ называется минимальным уточнением (или минимальным обобщением) и обозначается $P_1 \Rightarrow_{\min} P_2$.

Теорема 2.2.1.

Минимальное уточнение возможно только ограниченным множеством замен(подстановок):

e → ε (пустота);

e → te;

e → et;

t → (e);

t → s;

s → X (атом).

Доказательство.

Минимальная подстановка — по аналогии с минимальным уточнением, такая подстановка $v \to P$, что не существует образца Q , содержащего переменной w и подстановки $S = \{w \to P'\}$, что $v \to Q$, а $Q \overset{S}{\Rightarrow} P$.

Очевидно, что каждая из последних трех подстановок минимальна, потому что заменяемая переменная по определению является своей заменой. Так же не вызывает сомнений замена e → ε, так как только e-переменная может являться пустой последовательностью. Подстановки e → te и e → et косвенно следуют из определений t- и e-переменных. E-переменная является

последовательностью термов. T-переменная является термом. Если к последовательности термов присоединить терм, получится последовательность термов. Другими словами, e-переменная может быть дополнена t-переменной до e-переменной. Значит, все указанные подстановки минимальны.

Других подстановок для s-переменной не существует, так как это все допустимые для нее значения. Для t-переменной возможна подстановка атома, но ее можно заменить на последовательность подстановок t → s → X. Подстановка s-переменной или атома вместо e-переменной не имеет смысла, так как t-переменная учитывает эти возможности. Помимо этого, для e-переменной возможна подстановка последовательности термов любой длины, но она «покрывается» повторным применением 2 или 3 замен.

Таким образом, перечислены все минимальные подстановки. Стоит отметить, что каждая из них увеличивает сложность образца на 1.

Теорема 2.2.1.

Если $P \Rightarrow^{+} Q$ и существует 2 способа построения цепочки минимальных уточнений

$$P \underset{\min}{\Rightarrow} R_1^1 \underset{\min}{\Rightarrow} \ldots \underset{\min}{\Rightarrow} R_N^1 \underset{\min}{\Rightarrow} Q \quad ,$$

$$P \underset{\min}{\Rightarrow} R_1^2 \underset{\min}{\Rightarrow} \ldots \underset{\min}{\Rightarrow} R_M^2 \underset{\min}{\Rightarrow} Q \quad ,$$

то $M=N$.

Доказательство.

В доказательстве используется определение сложности образца. Каждая из минимальных замен увеличивает сложность образца на 1. При этом Q может быть получен из P цепочкой из M минимальных замен, то есть

$$C(Q)=C(P)+M \quad .$$

Так же Q может быть получен из P цепочкой из N минимальных замен, то есть

$$C(Q)=C(P)+N \quad .$$

Значит, $M=N$.

Замечание.

Из равенства длин цепочек не следует равенство самих цепочек. Например, $set \to sett \to stt \to sts$ и $set \to ses \to stes \to sts$ совпадают по длине, имеют одинаковые начальные и конечные образцы, но цепочки подстановок различны.

2.3. Сложнейшие обобщения образцов.

Пусть имеется набор образцов $P_1, \ldots, P_N$.

Локальное сложнейшее обобщение набора образцов $P_1, \ldots, P_N$. (далее ЛСО) — такой образец P^*, обобщающий каждый P_i из набора, что не существует образца Q, обобщающего каждый P_i из набора и P^*

$$P^* \Rightarrow^* P_i, i=1,\ldots,N \mid \nexists Q \Rightarrow^* P_i, i=1,\ldots,N : Q \Rightarrow^* P^* \ .$$

Глобальное сложнейшее обобщение набора образцов $P_1, \ldots, P_N$. (далее ГСО) — такой образец P^*, обобщающий каждый P_i из набора, что не существует образца Q, обобщающего каждый P_i из набора, сложность которого больше сложности P^*:

$$P^* \Rightarrow^* P_i, i=1,\ldots,N \mid \nexists Q \Rightarrow^* P_i, i=1,\ldots,N : C(Q) > C(P^*) \ .$$

Из определений следует, что $ГСО \in ЛСО$. Действительно, определения совпадают во всем, кроме последнего условия: отсутствие уточнения, обобщающего все образцы набора для ЛСО; максимальная сложность среди всех обобщений образцов набора для ГСО. При этом, из условия $C(Q) \Rightarrow^* C(P)$ следует условие $C(Q) > C(P)$, но не наоборот. Значит, если образец не принадлежит *ЛСО*, то он не принадлежит и *ГСО*, но если образец не принадлежит *ГСО*, то он может принадлежать *ЛСО*. Значит, $ГСО \in ЛСО$, но не $ЛСО \in ГСО$.

Сложнейшее обобщение имеет некоторые свойства, позволяющие упростить его построение.

Лемма 2.3.1.

Пусть P – образец, содержащий переменную v, и существует подстановки $S_Q = \{v \to Q'\}, P \xrightarrow{S_Q} Q, \quad S_R = \{v \to R'\}, P \xrightarrow{S_R} R$. Значит, $Q' \Rightarrow^* R'$ тогда и только тогда, когда $Q \Rightarrow^* R$.

Доказательство.

Очевидно, что P можно представить, как $P=e_1 v e_2$, тогда $Q=e_1 Q' e_2$, $R=e_1 R' e_2$. Естественно, что из $Q' \Rightarrow^* R'$ следует $Q \Rightarrow^* R$, и наоборот.

Теорема 2.3.1.

Пусть P – образец, содержащий переменную v, $S_1,\ldots,S_N$, $S_i=\{v \to P_i'\}$, $P \xrightarrow{S_i} P_i$, $S^*=\{v \to P^{*\prime}\}, P \xrightarrow{S^*} P^*$. В этом случае $P^* \in ЛСО(P_1,\ldots,P_N)$ тогда и только тогда, когда $P^{*\prime} \in ЛСО(P_1',\ldots,P_N')$.

Доказательство.

Доказательство состоит из двух этапов.

1 этап. Если $P^* \in ЛСО(P_1,\ldots,P_N)$, то $P^{*\prime} \in ЛСО(P_1',\ldots,P_N')$. Пусть данное утверждение неверно, то есть $P^{*\prime} \notin ЛСО(P_1',\ldots,P_N')$. Для каждого $i=1, \ldots, N$, существуют подстановки $S_i=\{v \to P_i'\}, P \xrightarrow{S_i} P_i$ и $S^*=\{v \to P^{*\prime}\}, P \xrightarrow{S^*} P^*$, и $P^* \Rightarrow^* P_i$, так как $P^* \in ЛСО(P_1,\ldots,P_N)$. В соответствии с доказанной выше леммой, $P^{*\prime} \Rightarrow^* P_i', i=1,\ldots,N$. Значит, $P^{*\prime}$ является некоторым, но не сложнейшим, обобщением $P_1',\ldots,P_N'$. Пусть существует $\bar{P}' \in ЛСО(P_1,\ldots,P_N)$, , значит, . $P^{*\prime} \Rightarrow^* \bar{P}'$ Но в этом случае существует $\bar{P} \Rightarrow^* P_i, i=1,\ldots,N$, такой, что $P^* \Rightarrow^* \bar{P}$, что противоречит начальному условию $P^* \in ЛСО(P_1,\ldots,P_N)$. Значит, $P^{*\prime} \in ЛСО(P_1',\ldots,P_N')$.

2 этап. Если $P^{*\prime} \in ЛСО(P_1',\ldots,P_N')$, то $P^* \in ЛСО(P_1,\ldots,P_N)$. Пусть утверждение неверно, то есть существует образец $\bar{P}$ такой, что $\bar{P} \Rightarrow^* P_i, i=1,\ldots,N$, $P^* \Rightarrow^* \bar{P}$ и $\bar{S}=\{v \to \bar{P}'\}, P \xrightarrow{\bar{S}} \bar{P}$. Последнее означает, что существует подстановка $\bar{S}=\{w \to \bar{P}^{*\prime}\}$, где w – переменная из $P^{*\prime}$, $P^{*\prime} \xrightarrow{\bar{S}^*} \bar{P}'$. Пусть $\bar{P} \in ЛСО(P_1,\ldots,P_N)$, тогда по первой части теоремы $\bar{P}' \in ЛСО(P_1',\ldots,P_N')$. Но существование подстановки $\bar{S}^*$ значит, что $P^{*\prime} \Rightarrow^* \bar{P}'$, что противоречит начальному условию $P^{*\prime} \in ЛСО(P_1',\ldots,P_N')$. Значит, $P^* \in ЛСО(P_1,\ldots,P_N)$.

Теорема 2.3.2.

Пусть P – образец, содержащий переменную v, существуют подстановки $S_1,\ldots,S_N$, $S_i=\{v \to P_i'\}, P \xrightarrow{S_i} P_i$ и $S^*=\{v \to P^{*\prime}\}, P \xrightarrow{S^*} P^*$. В этом случае

$P^* \in ГСО(P_1, \ldots, P_N)$ тогда и только тогда, когда $P^{*\prime} \in ГСО(P_1', \ldots, P_N')$.

Доказательство.

Доказательство состоит из двух этапов.

1 этап. Если $P^* \in ГСО(P_1, \ldots, P_N)$, то $P^{*\prime} \in ГСО(P_1', \ldots, P_N')$. Пусть данное утверждение неверно, то есть $P^{*\prime} \notin ГСО(P_1', \ldots, P_N')$. Тогда для каждого $i=1, \ldots, N$, существуют подстановки $S_i = \{v \to P_i'\}, P \xrightarrow{S_i} P_i$ и $S^* = \{v \to P^{*\prime}\}, P \xrightarrow{S^*} P^*$, и $P^* \Rightarrow^* P_i$, так как $P^* \in ГСО(P_1, \ldots, P_N)$. Тогда, в соответствии с леммой 2.3.1, $P^{*\prime} \Rightarrow^* P_i', i=1, \ldots, N$. Значит, $P^{*\prime}$ является некоторым, но не сложнейшим обобщением $P_1', \ldots, P_N'$. Пусть существует $\bar{P}' \in ГСО(P_1, \ldots, P_N)$, , значит, $P^{*\prime} \Rightarrow^* \bar{P}'$ и $C(\bar{P}') > C(P^{*\prime})$. Но в этом случае $\bar{P}$, полученный подстановкой $\bar{P}'$ вместо v, обладает большей сложностью, чем P^* , что противоречит начальному условию $P^* \in ГСО(P_1, \ldots, P_N)$. Значит, $P^{*\prime} \in ГСО(P_1', \ldots, P_N')$.

2 этап. Если $P^{*\prime} \in ГСО(P_1', \ldots, P_N')$, то $P^* \in ГСО(P_1, \ldots, P_N)$. Пусть утверждение неверно, то есть существует образец $\bar{P}$ такой, что $\bar{P} \Rightarrow^* P_i, i=1, \ldots, N$, $C(P^*) < C(\bar{P})$ и $\bar{S} = \{v \to \bar{P}'\}, P \xrightarrow{\bar{S}} \bar{P}$. Последнее означает, что существует подстановка $\bar{S} = \{w \to \bar{P}^{*\prime}\}$, где w – переменная из $P^{*\prime}$, $P^{*\prime} \xrightarrow{\bar{S}^*} \bar{P}'$, а значит, $C(P^{*\prime}) < C(\bar{P}')$. Пусть $\bar{P} \in ГСО(P_1, \ldots, P_N)$, тогда по первой части теоремы $\bar{P}' \in ГСО(P_1', \ldots, P_N')$. Тогда существует образец $\bar{P}'$, чья сложность выше сложности $P^{*\prime}$, значит, $P^{*\prime} \notin ГСО(P_1', \ldots, P_N')$, что противоречит начальному условию. Значит, $P^* \in ГСО(P_1, \ldots, P_N)$.

2.4. Классы образцов.

Класс $c(k)$ — множество образцов вида $P = T_1 T_2 \ldots T_k$, где T_i – терм.

Класс $c(m,n)$ — множество образцов вида $P = L_1 \ldots L_m e R_n \ldots R_1$, где L_i , R_i – термы. Следует обратить внимание на то, что любой образец класса $c(k)$ может быть представлен, как образец класса $c(m,n)$, так как е-переменная может являться пустой последовательностью, ε. Фактически, $c(k) = c(1,k-1) = \ldots = c(j,k-j) = \ldots = c(k-1,1)$.

Теорема 2.4.1.

Если $P_i \in c(m_i, n_i), i=1,\dots,N$, то для набора образцов $P_1, \dots, P_N$ верно:

$$ЛСО(P_1,\dots,P_N) \in c(m,n), где\ m=min(m_i), n=min(n_i) \ .$$

Если $P_i \in c(k_i), i=1,\dots,N$, то для набора образцов $P_1, \dots, P_N$ верно

$$ЛСО(P_1,\dots,P_N) \in c(j,k-j), где\ k=min(k_i) \ .$$

Доказательство.

Если $m_1<m_2$, $n_1<n_2$, $P_1 \in c(m_1,n_1), P_2 \in c(m_2,n_2)$, то есть $P_1=L_1^1 \dots L_{m_1}^1 e R_{n_1}^1 \dots R_1^1$ и $P_2=L_1^2 \dots L_{m_2}^2 e R_{n_2}^2 \dots R_1^2$, то путем минимальных уточнений типа e → te и e → et, а именно $e \to L_i^2 e$ и $e \to e R_i^2$ можно получить P_1 из P_2, то есть $P_1 \Rightarrow^+ P_2$. Тогда не вызывает сомнений и то, что для $m=min(m_i), n=min(n_i), P \in c(m,n), P \Rightarrow^+ P_i$. То есть такое P – обобщение $P_1, \dots, P_N$. При этом не существует таких $\bar{P}$, что $P \Rightarrow^* \bar{P} \Rightarrow^* P_i, i=1,\dots,N$, потому что это значило бы неминимальность выбранных m и n. Таким образом, $P=ЛСО(P_1,\dots,P_N) \in c(m,n)$.

Если $k_1<k_2$, $P_1 \in c(k_1), P_2 \in c(k_2)$, то путем минимальных обобщений типа te → e, et → e и ε → e можно получить P_2 из P_1, то есть $P_1 \Rightarrow^+ P_2$. Значит, для $k=min(k_i), P \in c(k), P \Rightarrow^+ P_i$. Тогда , аналогично с первой частью доказательства, P является *ЛСО(P_1, ..., P_N)*. При этом образец класса $c(k)$ является так же образцом всех классов типа $c(j,k-j), j=0,\dots,k$, а значит, для него применима первая часть теоремы. Таким образом, $P=ЛСО(P_1,\dots,P_N) \in c(j,k-j), j=0,\dots,k$

Следствием этой теоремы можно считать такое замечание.

Следствие 2.4.1.1.

Пусть в наборе образцов $P_1, \dots, P_N$ присутствуют образцы обоих классов. Пусть $k=min(k_i), m=min(m_i), n=min(n_i)$. Тогда для ЛСО допустимы следующие классы:

- $k \geq m+n$, ЛСО относится к классу $c(m,n)$;
- $k<m+n, k \geq m, k \geq n$, ЛСО относится к классам $c(j,k-j), j=k-n,\dots,m$;
- $k<m+n, k<m, k \geq n$, ЛСО относится к классам $c(j,k-j), j=k-n,\dots,k$;
- $k<m+n, k \geq m, k<n$, ЛСО относится к классам $c(j,k-j), j=k-m,\dots,k$;

- $k<m+n, k<m, k<n$, ЛСО относится к классам $c(j, k-j), j=0,\ldots,k$.

2.5. Быстрое обобщение.

Быстрое обобщение (далее БО) двух образцов P_1 и P_2 – образец P, построенный по определенным правилам:

1) если P_1 и P_2 являются термами, то быстрое обобщение строится согласно таблице 1;

2) если P_1 и P_2 являются образцами класса $c(k)$, $P_1=T_1^1\ldots T_k^1, P_2=T_1^2\ldots T_k^2$, то

$$БО(P_1, P_2)=T_1^*\ldots T_k^*, где\ T_i^*\ БО(T_i^1, T_i^2) \ ;$$

3) если P_1 и P_2 являются образцами класса $c(m_1, n_1)$ и $c(m_2, n_2)$ соответственно, $P_1=L_1^1\ldots L_{m_1}^1 e R_{n_1}^1\ldots R_1^1, P_2=L_1^2\ldots L_{m_2}^2 e R_{n_2}^2\ldots R_1^2$, то

$$БО(P_1, P_2)=L_1^*\ldots L_m^* e R_n^*\ldots R_1^*, где\ L_i^*\ БО(L_i^1, L_i^2), R_i^*\ БО(R_i^1, R_i^2), m=min(m_1, m_2) n=min(n_1, n_2);$$

4) иначе, $БО(P_1, P_2)=e$.

Быстрое обобщение набора образцов $P_1, \ldots, P_N$ определяется рекурсивно:

$$БО(P_1, \ldots, P_{N-1}, P_N)=БО(БО(P_{1,\ldots}, P_{N-1}), P_N) \ .$$

Таблица 1. Построение быстрого обобщения для двух термов

P_1/P_2	X	Y≠X	s	t	(P_1^*)
X	X	s	s	t	t
s	s	s	s	t	t
t	t	t	t	t	t
(P_2^*)	t	t	t	t	$(БО(P_1, P_2))$

Теорема 2.5.1.

$БО(P_1, \ldots, P_N) \Rightarrow^* ГСО(P_1, \ldots, P_N)$.

Доказательство.

Для доказательства этой теоремы достаточно рассмотреть каждую из 4 возможностей построения быстрого обобщения.

Если $БО(P_1, \ldots, P_N)=e$, то верность утверждения теоремы не вызывает сомнений, так как $e \Rightarrow^* ГСО(P_1, \ldots, P_N)$.

Если все образцы набора принадлежат классу $c(k)$, то их быстрое

обобщение принадлежит этому же классу, а глобальное сложнейшее обобщение — классам $c(j,k-j), j=0,\ldots,k$ по теореме 2.4.1. В таком случае, ГСО обобщается до БО минимальной подстановкой $e \to \varepsilon$, а значит выполнена теорема.

Если все образцы набора принадлежат классам $c(m_i,n_i)$ способ построения быстрого обобщения и теорема 2.4.1 показывают, что и быстрое, и глобальное обобщения принадлежат одному классу $c(m,n)$, где $m=min(m_i), n=min(n_i)$, а значит, $БО(P_1,\ldots,P_N) \Rightarrow^{+} ГСО(P_1,\ldots,P_N)$ или $БО(P_1,\ldots,P_N)=ГСО(P_1,\ldots,P_N)$, то есть $БО(P_1,\ldots,P_N) \Rightarrow^{*} ГСО(P_1,\ldots,P_N)$.

Если $P_1, \ldots, P_N$ являются термами, но не являются скобочными выражениями, то их БО и ГСО совпадают. Если некоторые из них являются скобочными выражениями, то выражения внутри скобок подчиняются всему выше описанному, а значит и для $P_1, \ldots, P_N$ — термов верно $БО(P_1,\ldots,P_N) \Rightarrow^{*} ГСО(P_1,\ldots,P_N)$.

Следствие 2.4.1.1.

Быстрое обобщение может использоваться для упрощения построения глобального сложнейшего обобщения.

Доказательство.

Действительно, по теореме 2.5.1 $БО(P_1,\ldots,P_N) \Rightarrow^{*} ГСО(P_1,\ldots,P_N)$, что означает существование подстановки $S=\{v_j \to P_j' \mid j=1,\ldots,K\}$, где $v_1, \ldots, v_K$ – переменные образца $P=БО(P_1,\ldots,P_N)$, такой, что $БО(P_1,\ldots,P_N) \xrightarrow{S} ГСО(P_1,\ldots,P_N)$. Так как $P=БО(P_1,\ldots,P_N)$, то существуют такие подстановки $S_i=\{v_j \to P_{ij}' \mid j=1,\ldots,K\}, i=1,\ldots,N$, что $P \xrightarrow{S_i} P_i, i=1,\ldots,N$. По теореме 2.3.2, $P_j' \in ГСО(P_{1j}',\ldots,P_{Nj}'), j=1,\ldots,K$ тогда и только тогда, когда $БО(P_1,\ldots,P_N) \Rightarrow^{*} ГСО(P_1,\ldots,P_N)$. Так как второе из условий выполнено, то, очевидно, должно выполняться и первое. Значит, замена переменных в быстром обобщении набора образцов $P_1, \ldots, P_N$ на глобальное сложнейшее обобщение наборов соответствующих переменным подстановок переведет быстрое

обобщение в глобальное сложнейшее.

Именно этот факт используется в алгоритме вычисления глобального сложнейшего обобщения.

2.6. Алгоритм вычисления глобального сложнейшего обобщения.

Алгоритм вычисления глобального сложнейшего обобщения можно условно разбить на несколько этапов.

Предварительным этапом можно назвать вычисление жестких образцов $P_i^H, i=1,\dots,N$ для каждого из образцов начального набора образцов $P_1,\dots,P_N$.

Первым этапом находится быстрое обобщение $P^*=БО(P_1^H,\dots,P_N^H)$. Кроме самого обобщения вычисляются так же подстановки $S_i=\{v_j \to P_{ij} | j=1,\dots,N\}, P^* \overset{S_i}{\to} P_i$.

После вычисления быстрого обобщения и подстановок производится разбор полученного образца. Если в быстром обобщении нет е-переменных с несколькими альтернативами, то оно является и глобальным сложнейшим обобщением, и на этом алгоритм заканчивается. Стоит заметить, что подстановки $S_1=\{e_n \to P, v_j \to P_{1j},\dots\}$ и $S_2=\{e_n \to P, v_j \to P_{2j},\dots\}$ для переменной e_n являются эквивалентными и не дают нескольких альтернатив, несмотря на то, то что другие переменные имеют разные значения в подстановках.

Если же для какой-либо е-переменной найдены несколько различных подстановок, то вычисляются классы дальнейшего поиска ГСО. Для этого сначала находятся классы каждой из замен, то есть для каждой подстановки $S_i=e_j \to P_{ij}$ вычисляется класс образца P_{ij}. По теореме 2.4.1 или следствию 2.4.1.1 из пункта 2.4 данной работы допустимые для ГСО всех образцов-замен переменной e_j имеют определенный вид, зависящий от классов образцов P_{ij}.

Следующие шаги проводятся для каждого из допустимых классов ГСО. Сначала каждый образцов P_{ij} накладывается на текущий класс. Под наложением на класс *c(m, n)* подразумевается обобщение P_{ij} до $L_1^i \dots L_m^i e R_n^i \dots R_1^i$. Наложение на класс *c(k)* возможно только для образца класса *c(k)*, а значит, все образцы

должны принадлежать классу *c(k)*. После наложения на текущий класс для полученных образцов рекурсивно вычисляется потермовое глобальное сложнейшее обобщение — частичное ГСО. Так как все наложенные образцы принадлежат одному классу, это возможно.

Последним этапом является выбор наиболее подходящего частичного ГСО. Полученные для каждого допустимого класса частичные ГСО сравниваются по своей числовой характеристике — сложности образца. Сложнейшее (имеющее максимальную сложность) частичное ГСО $\bar{P}^*$ считается глобальным сложнейшим обобщением для данной подстановки e-переменной.

Таким образом, имеется образец P^*, содержащий переменную e_j, подстановки $S_i=\{ e_j \to P_{ij} \}, P^* \xrightarrow{S_i} P_i^H$, а также $\bar{P}^* \in ГСО(P_{1j},\dots,P_{Nj})$. Тогда по теореме 2.3.2, доказанной в пункте 2.3 данной работы, подстановка $S^*=\{ e_j \to \bar{P}^* \}$ переводит быстрое обобщение P^* в ГСО $\bar{P}$: $, P^* \xrightarrow{S^*} P$.

После выполнения этих действий для каждой из подстановок e-переменных с несколькими альтернативами будет получено глобальное сложнейшее обобщение *P* для начального набора образцов $P_1,\dots,P_N$.

Пример работы алгоритма.

Пусть даны образцы

$$P_1=X((e.1)Ae.2(B))Y \quad \text{и}$$

$$P_2=X(A(C))Z \quad .$$

Их быстрое обобщение имеет вид

$$P^*=X(e.3)s.1$$

с подстановками $S_1=\{e.3 \to (e.1)Ae.2(B)\}$ и $S_2=\{e.3 \to A(C)\}$.

Так как в быстром обобщении присутствует e-переменная с несколькими альтернативами ($e.3 \to (e.1)Ae.2(B)$ или $e.3 \to A(C)$), то вычисляются допустимые классы частичного ГСО. Для первой подстановки образец принадлежит классу *c(2, 1)*, для второй — *c(2)*. По следствию 2.4.1.1 для *m=2*, *n*=1, *k=2* ($k<m+n, k \geq m, k \geq n$) допустимыми классами являются классы вида

$c(j,k-j), j=k-n,\ldots,m$, а точнее $c(j,2-j), j=1,2$, то есть *c(1, 1)* и *c(2, 0)*.

После вычисления допустимых классов для каждого из них находятся наложения на каждый из образцов и вычисление частичного ГСО.

Для допустимого класса *c(1, 1)*:

наложение для P_1 – $P_1^1=(e.1)e.4(B)$;

наложение для P_2 – $P_2^1=Ae.5(B)$;

частичное ГСО — $\bar{P}^1=t.1e.6(B)$, его сложность $C(\bar{P}^1)=1+3+3-1+1=7$.

Для допустимого класса *c(2, 0)*:

наложение для P_1 – $P_1^2=(e.1)Ae.4$;

наложение для P_2 – $P_2^2=A(B)e.5$;

частичное ГСО — $\bar{P}^2=t.1t.2e.6$, его сложность $C(\bar{P}^2)=2-1+1=2$.

Значит, для получения ГСО используется подстановка $S^*=\{e.3\rightarrow t.1e.6(B)\}$, а ГСО имеет вид $\bar{P}=X(t.1e.6(B))s.1$.

Глава 3. Реализация алгоритма.

В модуле `HighLevelRasl.sref` компилятора Простого Рефала производится перевод текста программы из абстрактного синтаксического дерева в последовательность команд промежуточного представления RASL. Так как это последний этап существования программы, как списка предложений вида `((e.Pattern)(e.Result))`, то именно на этом этапе компиляции требуется проведение оптимизации. Для возможности работы компилятора как в режиме оптимизации, так и без нее, исходный код компилятора не изменяется, а дополняется модулем `HighLevelRasl-OptPattern.sref`, который запускается только в случае запуска с соответствующим ключом-опцией. Кроме того, компилятор дополняется модулями `HardSent.sref`, `FastGen.sref` и `GlobalGen.sref`, содержащими функции вычисления жестких образцов, быстрого обобщения и глобального сложнейшего обобщения, соответственно, что позволяет структурировать код.

3.1. Первые попытки оптимизации.

Несмотря на хорошее обоснование алгоритма обобщений, первые попытки внедрения оптимизаций проводились с помощью метода, похожего На метод Романенко. В качестве языка сборки использовалось промежуточное представление HighLevelRASL. Главным отличием от предложенного Романенко алгоритма являлось отсутствие дерева команд. После окончательной генерации промежуточного представления каждое предложение представляло собой список команд, «обернутый» в команду `(#CmdSentence)`. Список команд в каждом предложении начинался с команды инициализации `(#CmdInitB0)`, поэтому даже при отсутствии одинаковых команд сопоставления после оптимизации хотя бы инициализация проводилась единожды, а не повторялась несколько раз. Выявление одинаковых команд проводилось рекурсивно, сначала для первой пары предложений в списке проводилось слияние, после чего для результата слияния и оставшихся предложений вызывалась функция выявления одинаковых команд (листинг 6).

Листинг 6: Первые попытки оптимизации.

```
CommonPattern {
  (e.first) (e.second) e.tail =
    <CommonPattern <Merge (e.first) (e.second)> e.tail>;
  e.any = e.any;
}

Merge {
  (#CmdSentence e.tail1 )(#CmdSentence e.tail2 ) =
    (#CmdSentence <DoMerge ( e.tail1 )( e.tail2 )>);
  (e.any)(e.other) = (e.any)(e.other);
}

DoMerge {
  (t.Common e.tail1 )(t.Common e.tail2 ) =
    t.Common <DoMerge ( e.tail1 )( e.tail2 )>;
  (e.any)(e.other) = ( #CmdSentence e.any)( #CmdSentence e.other);
}
```

Несмотря на простоту решения, попытка оптимизации оказалась неудачной. Выигрыш времени при выполнении оптимизированной программы составлял не более 5 %, даже на самых приспособленных к ней примерах. Это произошло потому что, в отличии от алгоритма Романенко, данный метод не давал реального ветвления дерева команд. Если бы дерево было построено так же, как у Романенко, то данный алгоритм объединял бы не пару соседних ребер, а все ребра слева направо, выходящие из одного узла, если они отмечены одинаковыми командами, до первой несовпадающей команды. Оставшиеся ребра не рассматривались бы. В итоге дерево имело бы странный вид, прекрасно отражающий неэффективность оптимизации (рисунок 5).

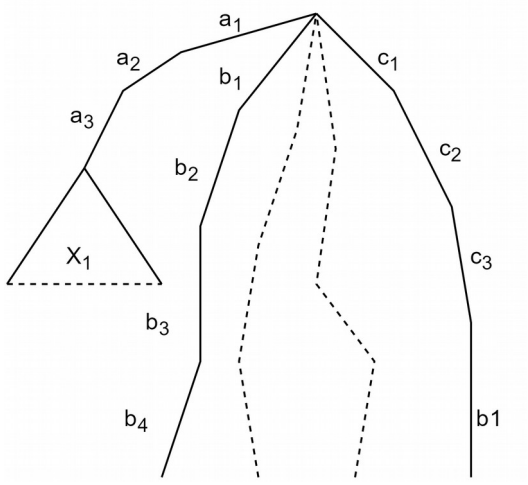

Рисунок 5: Неудачное выделение общих частей предложений.

Здесь a_1, a_2, a_3 – совпавшие команды нескольких первых предложений. По построению a_1 не совпадает с b_1, но при этом b_i может совпадать с любыми из команд следующих ребер.

В связи с очевидной неэффективностью описанного метода от него было решено отказаться.

3.2. Генерация жёстких образцов.

Генерация жестких образцов для списка образцовых частей предложений функции производится в модуле `HardSent.sref`. В качестве входных данных заглавная функция `HardSent` модуля получает образец, жесткий образец которого необходимо построить. В качестве результата функция возвращает жесткий образец и подстановки переменных этого образца, переводящие его в исходный образец.

Пусть Ƥ — множество всех жестких образцов образца *P*. Сложнейшим жестким образцом образца P называется образец $\bar{P}\in\mathcal{P}$, такой, что не существует $Q\in\mathcal{P}$, что $C(Q)>C(\bar{P})$. В листинге 7 представлен псевдокод алгоритма вычисления сложнейшего жесткого образца. На вход такой функции подается образец P, а результатом выполнения являются жесткий образец и набор подстановок. В приведенном ниже алгоритме проводятся только отделения терма слева. Для термов справа действуют те же правила, но похожие этапы алгоритма были опущены.

Листинг 7. Псевдокод алгоритма, генерирующего сложнейший жесткий образец.

```
P – входной образец
S – набор подстановок
S = []
HardSent {
  if P = (P') then ret (HardSent(P'))
  if P = (P1)P2 then ret (HardSent(P1))HardSent(P2)
  if P = T P' then ret HardSent(T) HardSent(P') // T - терм
  if (P = v) and (v = s- или t-переменная) then
    S.append({X → v}) // X – уникальное в списке подстановок имя
    ret X             // переменной того же типа, что v.
  else
    S.append({eX → P}) // eX – уникальное в списке подстановок
    ret e              // имя e-переменной
```

```
}
```

Вид результата выполнения в реальной программе более компактный. Жесткий образец и подстановки объединены в единую последовательность скобок. Каждая скобка содержит тэг s-, t- или e-переменной, а так же соответствующую ей подстановку. Таким образом, если собрать тэги из всех скобок, будет получен жесткий образец (индексы не имеют значения, так как жесткий образец подразумевает отсутствие повторных переменных, а значит, все переменные уникальны), а если собрать все замены, то получится исходный образец. Таким образом, при составлении жесткого образца получается образец такого вида:

$$P_H = (v_1('idx')((P_1')))\ldots(v_n('idx')((P_n'))) \ ,$$

где v_i – тэг типа переменной, P_i' – подстановка переменной v_i в образец P_H. Имена(индексы) переменных начального образца P сохранены в подстановках P_i', именно поэтому возможно восстановление начального образца.

Подстановки переменных не используются до этапа вычисления ГСО, поэтому их сохранение может показаться неэффективным. Наложение быстрого обобщения на каждый из образцов перед генерацией ГСО позволяет не «протаскивать» их через весь алгоритм. Несмотря на это, наложение сопряжено со многими проблемами, например, необходимостью организации цикла по открытой e-переменной. Для упрощения алгоритма подстановки переменных сохраняются.

3.3. Быстрое обобщение.

Все функции, участвующие в вычислении быстрого обобщения списка образцовых выражений, реализованы в модуле `FastGen.sref`.

В качестве входных данных основная функция `FastGen` модуля `FastGen.sref` получает список жестких образцов и замен переменных для них. По правилам, описанным в пункте 2.5, строится быстрое обобщение. Кроме 4 вариантов, рассмотренных в алгоритме построения, отдельно учитываются еще

2 варианта: один образец или стоящие подряд пустые образцовые выражения. Листинг 8 показывает устройство функции `FastGen`.

Листинг 8. Фрагмент функции вычисления быстрого обобщения

```
FastGen {
  s.Num (e.1) = s.Num (e.1);
  s.Num () () e.Tail = <FastGen <Inc s.Num> () e.Tail>;

  /* Pair of terms */
  s.Num
  (t.1) (t.2) e.Tail =
    <FastGen
      <Inc s.Num> (<FastGen-Terms s.Num (t.1) (t.2)>) e.Tail
    >;

  /* c(m, n) */
  s.Num
  ( e.1 (#E '$' (e.Pattern1)) e.2 )
  ( e.3 (#E '$' (e.Pattern2)) e.4 )
  e.Tail =
    <FastGen
      <Inc s.Num>
      (<FastGen-MeN
        s.Num
        ( e.1 (#E '$' (e.Pattern1)) e.2 )
        ( e.3 (#E '$' (e.Pattern2)) e.4 )
      >)
      e.Tail
    >;

  /* c(k) */
  s.Num
  ( e.1 ) ( e.2 ) e.Tail =
    <FastGen
      <Inc s.Num>
      (<FastGen-K
        s.Num ( e.1 ) ( e.2 )
      >)
      e.Tail
    >;
}
```

Как и многие другие функции, `FastGen` является рекурсивной. Функция находит быстрое двух первых элементов списка, после чего помещает результат в начало этого же списка вместо обработанных образцов и вызывается рекурсивно.

Отличием программы от алгоритма из пункта 2.5 является число `s.Num` обработанных образцов. Это число отражает количество уже отработанных образцов. При составлении быстрого обобщения двух элементов списка число `s.Num` увеличивается на единицу, после чего проводится работа с их подстановками.

Если элементы оказались атомами, то, в соответствии с таблицей 1, они заменяются либо на атом, если совпадают (подстановок нет, так как атом не является переменной), либо на s-переменную с подстановками в виде первого и второго атомов: $\#S\ '\$'((\#Atom.1))((\#Atom.2))$. Аналогично обрабатываются и пары атом-s-переменная и s-переменная- s-переменная.

Если типы переменных при сопоставлении совпали, то подстановки для этих переменных просто переносятся в подстановки для результата: $\#Tag\ '\$'((Subst_1))...((Subst_N))$. Если тип не совпал, то подстановка обобщается до большего типа, а уже совершенные совмещения подстановок для меньшего отменяются. Например, при совмещении s- и t-переменных результатом будет являться t-переменная:

$\#S\ '\$'((Subst_{S1}))...((Subst_{SK}))$ и $\#T\ '\$'((Subst_{T1}))...((Subst_{TN}))$ дадут результат $\#T\ '\$'((\#S'\$'((Subst_{S1})))) ...((\#S'\$'((Subst_{SK})))) ((Subst_{T1}))...((Subst_{TN}))$. На этом этапе появляется необходимость в понимании числа совершенных совмещений. Например, совместив пару одинаковых атомов в один, а после разделив подстановку, можно потерять некоторые из замен. Если же известно их количество, то можно восстановить недостающие элементы подстановок по окружающим заменам. Поэтому сохранение числа обработанных образцов `s.Num` является полезной модификацией алгоритма.

Если же результатом обобщения является e-переменная, то снова возникает необходимость деления совмещенных подстановок.

3.4. Глобальное сложнейшее обобщение.

Все функции, реализующие построение глобального сложейшего обобщения, кроме вычисляющих быстрое обобщение, реализованы в модуле

компилятора `GlobalGen.sref`. Синтаксис Простого Рефала поддерживает использование безымянных функций, что упрощает разработку кода. Код функций `CreateGlobalGen`, `GlobalGen` и `GlobalGen-Aux` представлен в листинге 9.

Листинг 9. Функции модуля GlobalGen.sref

```
$ENTRY CreateGlobalGen {
  e.HardSentences =
    <Fetch
      <CreateFastGen e.HardSentences>
      <Seq
        {s.Num (e.FastGen) =
          (e.FastGen)(<InspectFastGen e.FastGen>);}
        {(e.FastGen)(e.Inspected) =
            (e.FastGen)(<GlobalGen e.Inspected>);}
      >
    >;
}
GlobalGen {
  e.Inspected =
    <Map
      {
        (#E 0 '$' e.Body) = <GlobalGen-Aux e.Body>;
        (#E 1 '$' e.Body) = (#E '$' e.Body);
        (#Brackets s.Num e.inBrackets) =
          (#Brackets <GlobalGen e.inBrackets>);
        (e.Any) = (e.Any);
      }
      e.Inspected
    >;
}
GlobalGen-Aux {
  e.Replacements =
    <GetComplex
      <Map
        { ((e.Class) e.Body) =
            <CalcComplexity <Generalization 1 e.Body>>;}
        <CreateSuperposition
          (e.Replacements )
          (<ComputePossibleClasses e.All>)
        >
      >
    >;
}
```

Функция `CreateGlobalGen` отвечает за вызов функций вычисления

быстрого обобщения и проверки наличия многоальтернативных e-переменных. Функция `InspectFastGen` просматривает все подстановки переменных в БО и вычисляет их количество. Она возвращает размеченное быстрое обобщение, где у e-переменых с несколькими подстановками после тэга стоит отметка «0», а у однозначно-определенных — «1». Функция `GlobalGen` обрабатывает исследованный образец, рекурсивно вызывая себя для внутрискобочных выражений. Функция `GlobalGen-Aux` реализует алгоритм, описанный в пункте 2.5. Для каждой замены проводится вычисление класса, для всех замен — вычисление допустимых классов частичного ГСО, после ищутся потермовые обобщения, сложнейшее из которых и возвращает функция.

3.5. Высокоуровневый RASL.

Оптимизация с составлением ГСО применяется на этапе создания высокоуровнего RASL'a — промежуточного представления. До генерации промежуточного представления в функции `HighLevelRASL-Function` модуля `HighLevelRASL-OptPattern.sref` генерируется список команд и подстановок для предложений функций компилируемой программы. На момент начала работы каждое предложение обрабатывалось отдельно, что не давало возможности оптимизации. Но поскольку поиск ГСО является сложной операцией, затрачивающей время и ресурсы, разумно ввести проверку необходимости этого поиска. В частности, если функция не содержит предложений или содержит одно предложение, вычисление ГСО будет вырожденным. В связи с этим функция `HighLevelRASL-Function` претерпела значительные изменения (листинг 10). Несмотря на это, некоторая часть прошлой версии этой функции осталась в программе в качестве функции `HighLevelRASL-OneFunction`, выполняющей обработку одного предложения без вычисления ГСО. В свою очередь, функция `HighLevelRASL-MulFunction` реализовывает выделение общих частей сопоставлений на основе ГСО, после чего проверяет уже уникальные для каждого предложения подстановки для каждого предложения. Для пустой функции, то есть для функции, не

содержащей ни одного предложения, Даже выполнение функции `HighLevelRASL-OneFunction` будет вырожденным. Поэтому производимая проверка учитывает отдельно случай пустой функции. Для него без каких-либо вычислений сразу возвращается команда генерации пустой функции `(#CmdEnum s.ScopeClass e.Name)`.

Листинг 10. Функция HighLevelRASL-Function

```
HighLevelRASL-Function {
  s.ScopeClass (e.Name) e.Sentences =
    <Fetch
      <ListLen (0) e.Sentences>
      {
        0 = (#CmdEnum s.ScopeClass e.Name);
        1 =
          <HighLevelRASL-OneFunction s.ScopeClass (e.Name)
e.Sentences>;
        s.Else =
          <HighLevelRASL-MulFunction s.ScopeClass (e.Name)
e.Sentences>;
      }
    >;
}
```

3.6. Генерация кода.

Генерация кода косвенно начинается еще в модуле `HighLevelRASL-OptPattern.sref` в функциях `GenPatern` и `GenResult`. Эти функции на основе команд, полученных из `HighLevelRASL-Function`, генерируют промежуточное представление RASL. Так как в прошлых версиях компилятора генерация RASL-представления происходила для каждого предложения, то эти функции нуждались в модификации. В частности, `GenPatern` не мог обработать подстановку s- или t-переменной, так как ранее в этом не было необходимости. В отличии от e-переменной, занимающей 2 позиции в контексте, s- и t-переменные занимают одно место.

В новой версии функция `GenPatern` вызывается не для каждого предложения функции, а отдельно для общих частей и различающихся частей предложений (фактически, существует 2 функции, `GenPatern` и `GenPatern-`

`New`, но различия в них не значительны). При вызове для общих частей предложений в качестве аргумента передается составленное глобальное сложнейшее обобщение. На его основе формируется список команд сопоставления выражения с обобщением. После составления промежуточного представления для общей части предложений, `GenPatern` вызывается для генерации оставшихся команд для каждого предложения. При этом она получает в качестве аргумента список подстановок и генерирует на их основе команда сохранения и сопоставления. Команды сохранения необходимы для правильной обработки открытых e-переменных. При организации цикла по значениям такой переменной необходимо сохранение промежуточных значений смещений ее границ. Так как для каждого предложения подобное сопоставление может иметь разные результаты, то разумно скопировать необходимые значения в свободные участки памяти, после чего проводить работу с ними. Таким образом, данные, общие для всех предложений останутся неприкосновенными. Генерация команд сопоставления совпадает с аналогичной для общих частей предложений.

Введение изменений в `GenPatern` повлекло за собой необходимость дополнения модуля `Generator.sref` функциями обработки переменных, занимающих одно место в контексте. Для этого, кроме стандартных направлений движения #AlgLeft и #AlgRight, было введено условное напрвление #AlgTerm. При обнаружении подобного тэга направления модуль `Generator.sref` генерирует команду вызова функций, отвечающих за сопоставление термов. В свою очередь, файлы `refalrts.h` и `refalrts.cpp` были дополнены соответствующими функциями, названия которых оканчиваются на суффикс "`_term`".

После внесения необходимых изменений в необходимые функции и модули, генерация кода на основе оптимизированного RASL'а стала возможна. Результат выполнения оптимизации хорошо виден на примере функции, приведенной в листинге 11.

Листинг 11. Пример. Тестовая функция

```
$ENTRY Go {
  'a' 'b' e.Any ('c') = ;
  'a' 'd' e.Any ('e') = ;
}
```

Код на C++, сгенерированный компилятором в обычном режиме и в режиме оптимизации приведен для сравнения в Приложениях 2 и 3. Фрагменты этого кода можно увидеть в таблице 2.

Таблица 2: Сравнение сгенерированного кода на C++

Без оптимизации	С оптимизацией

`do {`
`  if( !char_left(`
`  'a',`
`  context[0], context[1])`
`  )`
`  continue;`
`  if( !char_left(`
`  'b',`
`  context[0], context[1])`
`  )`
`  continue;`
`...`
`  if( !brackets_right(`
`  context[2],`
`  context[3],`
`  context[0],`
`  context[1])`
`  )`
`  continue;`
`...`
`} while ( 0 );`
`if( !char_left(`
`  'a',`
`  context[0], context[1] )`
`  )`
`  return cRecognitionImpossible;`
`if( !char_left(`
`  'd',`
`  context[0], context[1] )`
`  )`
`  return cRecognitionImpossible;`
`…`
`if( !brackets_right(`
`  context[2],`
`  context[3],`
`  context[0],`
`  context[1])`
`  )`
`  return cRecognitionImpossible;`
`...`
	`if( !char_left(`
	`  'a',`
	`  context[0], context[1] )`
	`  )`
	`  return cRecognitionImpossible;`
	`...`
	`if( !brackets_right(`
	`  context[2],`
	`  context[3],`
	`  context[0],`
	`  context[1])`
	`  )`
	`  return cRecognitionImpossible;`
	`do {`
	`  if( !char_left(`
	`  'b',`
	`  context[0], context[1])`
	`  )`
	`  continue;`
	`...`
	`} while ( 0 );`
	
	`if( !char_left(`
	`  'd',`
	`  context[0], context[1] )`
	`  )`
	`  return cRecognitionImpossible;`
	`...`

При генерации кода на C++ для данного примера без оптимизации блок кода сопоставлений переменных будет составлен для каждого из предложений. Из-за этого проверка совпадения первого атома объектного выражения с символом 'a' проводится дважды, так же как сопоставление крайнего правого терма со скобочной последовательностью. При генерации кода на C++ с оптимизацией команды сопоставления одинаковых элементов будут вынесены «за скобки», поэтому в сгенерированном коде сначала будет находится блок проверки на наличие у объектного выражения скобочного терма на крайней правой позиции, атома 'a' на крайней левой позиции и s-переменной на второй позиции слева (атомы ' b' и 'd' обобщаться до s-переменной). Если любая из этих проверок провалится, то функция не сможет распознать объектное выражение, поэтому выполнение остановится с ошибкой. После блока общих проверок находятся блоки сопоставлений для каждого предложения за исключением общих частей.

Даже на фрагменте видно, что оптимизированный код структурированнее, занимает меньше строк. Так же очевидно, что выполнение программы без повторных проверок происходит быстрее.

Глава 4. Тестирование.

Тестирование программы проводится в 2 этапа. Первый проверяет зависимость эффективности оптимизации от машины, выполняющей компиляцию, а второй — зависимость эффективности оптимизации от формата файла, устройства функций.

4.1. Тестирование разных машин.

Так как компилятор Простого Рефала является переносимым, то он должен работать на разных машинах и операционных системах, естественно, с использованием различных компиляторов C++. Проверка эффективности внедренной оптимизации проводится на 2 компьютерах с разными характеристиками и операционными системами.

Машина 1. Процессор Intel® Core™ i5-2520M CPU @ 2.50GHz × 4. ОС Ubuntu 14.04 LTS. Компилятор g++ 4.8.5.

Машина 2. Процессор Intel® Core™ i7-5500U CPU @ 2.40GHz × 2. ОС Windows 10 x64. Компилятор Microsoft (R) C/C++ версии 19.00.23506 для x64.

Для оценки результата оптимизации использовался сам компилятор: версия, скомпилированная в обычном режиме и оптимизированная версия. Время компиляции одних и тех же исходных разными версиями компилятора является наглядным результатом изменения скорости работы программы.

Тестирование проводилось на 2 файлах разной сложности, а так же на исходных файлах самого компилятора. Результат является средним для нескольких запусков. Под временем компиляции программы имеется ввиду чистое время компиляции Рефала (Таблица 3), так как время работы машинозависимого компилятора C++ может сильно отличаться. Одна из встроенных опций компилятора — замер времени, потраченного на работы с образцовыми выражениями. Это время так же изменяется при оптимизации (Таблица 4).

Результаты тестирования показывают, что эффективность оптимизации может изменяться в зависимости от компилятора. Несмотря на то, что важным фактором является формат исполняемой программы, основные различия в

результатах видны именно для разных машин.

Таблица 3. Сравнение времени компиляции программы

	Lexer.sref	Genertor.sref	Самоприменение
Машина 1, без оптимизаций	3.505s	0.831s	6.373s
Машина 1, с оптимизацией	3.460s	0.828s	6.326s
Относительная разность	1.5%	0.3%	0.3%
Машина 2, без оптимизаций	4.449s	1.044s	9.496s
Машина 2, с оптимизацией	2.900s	0.790s	5.951s
Относительная разность	35%	24%	37%

Таблица 4. Сравнение времени компиляции образцов

	Lexer.sref	Genertor.sref	Самоприменение
Машина 1, без оптимизаций	1.601s	0.393s	2.882s
Машина 1, с оптимизацией	1.582s	0.366s	2.860s
Относительная разность	1.2%	7%	0.8%
Машина 2, без оптимизаций	2.334s	0.544s	4.823s
Машина 2, с оптимизацией	1.246s	0.264s	2.104s
Относительная разность	47%	51%	56%

4.2. Тестирование файлов разной структуры.

Другой этап тестирования проводился только на Машине 2. При этом тестовые программы были скомпилированы в 2 режимах: стандартном и оптимизирующем. Скорость выполнения программ в каждом из режимов, а так же абсолютная и относительная разность результатов приведены в таблице 5. Первый столбец (NonOpt) показывает время выполнения программы без оптимизации, второй (Opt) – с оптимизацией. Третий и четвертый отражают разность во времени работы (абсолютная разность, с) и отношение разности во времени работы ко времени выполнения без оптимизации (относительная разность, %) соответственно.

В качестве тестовых программ использовались специально написанные программы разного формата. Программа подсчета количества скобок в заданном файле состоит из 2 функций, одна из которых состоит из одного предложения, а вторая — с предложениям с «неудачными» обобщениями.

Программа подсчета каждой из букв латинского алфавита в заданном файле содержит функцию `LetterCounter` (листинг 12), состоящую только из образцов схожего формата. 3-я программа — лексический анализатор заданного файла [6]. Он состоит из множества функций, обобщение образцов которых должно значительно ускорить выполнение программы.

Листинг 12: Функция LetterCounter, состоящая из предложений похожего формата

```
LetterCounter {
  s.Amount (s.Letter) s.Letter e.Tail =
    <LetterCounter <Inc s.Amount> (s.Letter) e.Tail>;
  s.Amount (s.Letter) (e.inBrackets) e.Tail =
    <LetterCounter s.Amount (s.Letter) e.inBrackets e.Tail>;
  s.Amount (s.Letter) t.Term e.Tail =
    <LetterCounter s.Amount (s.Letter) e.Tail>;
  s.Amount (s.Letter) e.Any = s.Amount;
}
```

Программы специально устроены так, что проводят большое количество вычислений, за счет чего выполняются довольно долго. Благодаря этому яснее виден эффект оптимизации.

Таблица 5: Время выполнения программ, скомпилированных в разных режимах

	NonOpt	Opt	$\Delta_{абс}$	$\Delta_{отн}$
BracketCounter.exe	4,081c	3,802c	0,279c	7%
LetterCounter.exe	43,412c	35,582c	7,830c	18%
Lexer.exe	6,366c	4,207c	2,159c	34%

Результаты этого этапа тестирования, как и ожидалось, показывают, что оптимизация играет бóльшую роль при наличии функций с большим количеством схожих по структуре образцов. За счет организации оптимизации компилятора функции с одним предложением не оптимизируются, а функции со слишком разными образцовыми частями предложений могут даже замедлить выполнение.

Заключение.

В рамках данной выпускной квалификационной работы бакалавра было проведено исследование предметной области, изучен язык Простой Рефал, найдены способы оптимизации. Выбранный алгоритм был подробно описан, его применимость обоснована. Был разработан и реализован алгоритм обобщения для оптимизации сопоставления с образцом в компиляторе Простого Рефала.

Алгоритм заменяет генерацию команд промежуточного представления для каждого предложения в функции на генерацию команд сопоставления для общих частей предложений как одного блока и генерацию команд для оставшихся частей предложений. Благодаря этому, удается избежать дублирования кода.

Проведен ряд экспериментов на разных компьютерах, операционных системах и компиляторах языка C++. Результаты были проанализированы. Они показали, что оптимизация применима и эффективна в разной среде. Проведен ряд экспериментов с разными компилируемыми программами, анализ результатов тестирования показал наличие зависимости между структурой компилируемого файла и эффективностью оптимизации.

Разработанный модуль `HighLevelRASL-OptPattern.sref` полностью выполняет поставленную задачу, оптимизируя сопоставление с образцом и ускоряя работу скомпилированной программы. Описанный теоретический комплекс может быть использован при создании других оптимизаторов и верификаторов программ на языке Простой Рефал.

Литература.

1. А. В. Коновалов. Пользовательская документация для языка Простой Рефал. [Электронный ресурс]. URL: https://github.com/Mazdaywik/simple-refal/blob/master/doc/manul.docx (дата обращения: 20.06.2016).

2. В. Ф. Турчин, Пользовательская документация для языка РЕФАЛ-5. [Электронный ресурс]. URL: http://www.refal.net/rf5_frm.html (дата обращения: 17.05.2016).

3. А. В. Коновалов, Компилятор Простого Рефала. [Электронный ресурс]. URL: https://github.com/Mazdaywik/simple-refal (дата обращения: 24.06.2016).

4. Романенко С. А. Машинно-независимый компилятор с языка рекурсивных функций. 1978.

5. Немытых А. П. Суперкомпилятор SCP4: Общая структура. М.: Издательство ЛКИ, 2007. — 152 с.

6. Альфред В. Ахо, Моника С. Лам, Рави Сети, Джеффри Д. Ульман. Компиляторы: принципы, технологии и инструментарий. — 2 изд. — М.: Вильямс, 2008. — 1184 с.

Приложения.

Приложение 1. Высокоуровневый RASL

```
<HighLevelRASL e.AST>
<HighLevelRASL-OptPattern e.AST>
<HighLevelRASL-OptResult e.AST>
  == e.RASLAST

e.RASLAST ::= t.RASLAST-Item*
t.RASLAST-Item ::=
    (#Function s.ScopeClass (e.Name) e.HiCommands)
  | t.CmdTopLevelItem

t.CmdTopLevelItem ::=
    (#CmdEnum s.ScopeClass e.Name)
  | (#CmdSwap s.ScopeClass e.Name)
  | (#CmdDeclaration s.ScopeClass e.Name)
  | (#CmdDefineIdent e.Name)
  | (#CmdSeparator)

e.HiCommands ::= t.HiCommand*
t.HiCommand ::=
    t.StatementCommand
  | t.SingleCommand

t.StatementCommand ::=
    (#CmdSentence e.HiCommands)
  | (#CmdOpenELoop s.Direction s.RangeOffset s.VarOffset e.HiCommands)

t.SingleCommand ::=
    (#CmdIssueMem s.Offset)
  | (#CmdInitB0)
  | (s.MatchCommand s.Direction s.Offset e.MatchInfo)
  | (#CmdSave s.OldOffset s.NewOffset)
  | (#CmdComment e.Text)
  | (#CmdEmptyResult)
  | (#CmdAllocateElem s.Offset s.AllocType e.AllocInfo)
  | (#CmdCopyVar s.Mode s.VarOffset s.SampleOffset
  | (#CmdInsertElem s.Offset)
  | (#CmdInsertRange s.Offset)
  | (#CmdInsertVar s.Mode s.Offset)
  | (#CmdLinkBrackets s.LeftOffset s.RightOffset)
  | (#CmdPushStack s.Offset)
  | (#CmdFail)
  | (#CmdReturnResult)

s.MatchCommand e.MatchInfo ::=
    #CmdBrackets s.NewBracketsOffset
  | #CmdADT s.NewBracketsOffset e.Name
  | #CmdNumber s.Number
  | #CmdIdent e.Name
  | #CmdChar s.Char
  | #CmdName e.Name
  | #CmdRepeated s.Mode s.VarOffset s.SampleOffset
  | #CmdEmpty
  | #CmdVar s.Mode s.VarOffset
  | #CmdCharSave s.Offset s.Char

s.Direction ::= #AlgLeft | #AlgRight

s.AllocType e.AllocInfo ::=
    #ElChar s.Char
  | #ElName e.Name
  | #ElIdent e.Name
  | #ElNumber s.Number
  | #ElString e.String
  | #ElOpenADT | #ElCloseADT
  | #ElOpenBracket | #ElCloseBracket
  | #ElOpenCall | #ElCloseCall

s.VarOffset, s.SampleOffset ::= s.Offset
s.OldOffset, s.NewOffset ::= s.Offset
s.LeftOffset, s.RightOffset ::= s.Offset
```

Приложение 2: Сгенерированный без оптимизации код на C++

```
refalrts::FnResult Go(refalrts::Iter arg_begin, refalrts::Iter arg_end) {
  refalrts::this_is_generated_function();
  refalrts::Iter context[4];
  refalrts::zeros( context, 4 );
  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] );
    // 'ab' e.Any#1/0 ( 'c' )
    if( ! refalrts::char_left( 'a', context[0], context[1] ) )
      continue;
    if( ! refalrts::char_left( 'b', context[0], context[1] ) )
      continue;
    context[2] = 0;
    context[3] = 0;
    if( ! refalrts::brackets_right( context[2], context[3], context[0],
context[1] ) )
      continue;
    if( ! refalrts::char_left( 'c', context[2], context[3] ) )
      continue;
    if( ! refalrts::empty_seq( context[2], context[3] ) )
      continue;
    // Генерация результата
  } while ( 0 );
  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] ) )
    return refalrts::cRecognitionImpossible;
  if( ! refalrts::char_left( 'd', context[0], context[1] ) )
    return refalrts::cRecognitionImpossible;
  context[2] = 0;
  context[3] = 0;
  if( ! refalrts::brackets_right( context[2], context[3], context[0],
context[1] ) )
    return refalrts::cRecognitionImpossible;
  if( ! refalrts::char_left( 'e', context[2], context[3] ) )
    return refalrts::cRecognitionImpossible;
  if( ! refalrts::empty_seq( context[2], context[3] ) )
    return refalrts::cRecognitionImpossible;
  // Генерация результата
}
```

Приложение 3: Сгенерированный с оптимизацией код на C++

```
// Automatically generated file. Don't edit!
#include "refalrts.h"


extern refalrts::FnResult Go(refalrts::Iter arg_begin, refalrts::Iter arg_end);

refalrts::FnResult Go(refalrts::Iter arg_begin, refalrts::Iter arg_end) {
  refalrts::this_is_generated_function();
  // issue here memory for vars with 9 elems
  refalrts::Iter context[9];
  refalrts::zeros( context, 9 );
  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] );
  //FAST GEN:S S E (S )
  //GLOBAL GEN:S S E (S )
  context[2] = 0;
  context[3] = 0;
  if( ! refalrts::char_term( 'a', context[4] ) )
    return refalrts::cRecognitionImpossible;
  if( ! refalrts::brackets_right( context[2], context[3], context[0],
context[1] ) )
    return refalrts::cRecognitionImpossible;
  if( ! refalrts::svar_left( context[4], context[0], context[1] ) )
    return refalrts::cRecognitionImpossible;
  if( ! refalrts::svar_left( context[5], context[0], context[1] ) )
    return refalrts::cRecognitionImpossible;
  if( ! refalrts::svar_left( context[6], context[2], context[3] ) )
    return refalrts::cRecognitionImpossible;
  if( ! refalrts::empty_seq( context[2], context[3] ) )
    return refalrts::cRecognitionImpossible;
  do {
    refalrts::start_sentence();
    //'a''b'E ('c')
    context[7] = context[0];
    context[8] = context[1];
    if( ! refalrts::char_term( 'b', context[5] ) )
      continue;
    if( ! refalrts::char_term( 'c', context[6] ) )
      continue;
  // Генерация результата
  } while ( 0 );

  //'a''d'E ('e')
  context[7] = context[0];
  context[8] = context[1];
    return refalrts::cRecognitionImpossible;
  if( ! refalrts::char_term( 'd', context[5] ) )
    return refalrts::cRecognitionImpossible;
  if( ! refalrts::char_term( 'e', context[6] ) )
    return refalrts::cRecognitionImpossible;
  // Генерация результата
}
```