Министерство науки и высшего образования 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:
«Встроенные функции компилятора Рефала-5λ»

Студент ИУ9-81	_____28.06.2020 (Подпись, дата)	Х.Р. Агазаде
Руководитель дипломной работы	_____ (Подпись, дата)	А. В. Коновалов
Консультант	_____ (Подпись, дата)	_____ (ФИО)
Консультант	_____ (Подпись, дата)	_____ (ФИО)
Заведующий кафедрой	_____ (Подпись, дата)	И. П. Иванов

2020 г.

АННОТАЦИЯ

Темой данной работы является «Встроенные функции компилятора Рефал-5λ». Объём данной дипломной работы составляет 58 страниц. Для её написания было использовано 11 источников. В работе содержатся 36 листингов и 5 таблиц.

В дипломную работу входят пять глав. В первой главе описывается язык Рефал-5λ, устройство его компилятора, встроенные функции. Во второй главе постановка задачи данной работы. В третьей главе описана разработка алгоритма оптимизации внутренних функций в компиляторе. В четвёртой главе рассматриваются результаты применения оптимизации встроенных функций к компилятору и оценивается прирост производительности при выполнении компиляции исходников Рефала-5λ. В пятой главе собрано руководство пользователя.

Объектом исследования данного ВКР являются эквивалентные преобразования вызовов встроенных функций компилятора Рефала-5λ. Эти преобразования изменяют исходный код программы на Рефале, с целью ускорения работы итоговой программы, также они открывают возможности для оптимизации функций внутренней библиотеки компилятора.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В библиотеке Рефала-5λ есть встроенные функции с известной для компилятора семантикой. Вызовы этих функций можно вычислять на этапе компиляции, если их аргументы статически известны. Программисты пишут такие вызовы редко, но они могут появиться при выполнении других оптимизаций, таких как прогонка [1] и специализация [2]. Поэтому оптимизация внутренних функций является важной задачей.

Во многих компиляторах C++ есть поддержка оптимизации внутренних функций [3]. Для того, чтобы включить оптимизацию встроенных функций, в C++ используется директива pragma. Вызовы таких функций, как memcpy, sin, cos [4] заменяются на вставку кода, который реализует данные функции. Отличие от встраивания в том, что код встраиваемой функции, должен быть определён в текущем файле, либо подключён из других файлов, а код встроенной функции находится в компиляторе. В данной работе рассматривается реализация подобной функциональности в компиляторе Рефал-5λ.

Цель работы – реализация оптимизации внутренних функций Рефала-5λ. В рамках работы нужно решить задачи:

- Изменение лексики и грамматики языка Рефал-5λ, для поддержки оптимизации внутренних функций.
- Разработка алгоритма оптимизации встроенных функций.
- Внедрение оптимизации встроенных функций в компилятор и оценка времени его работы после оптимизаций.

1 Обзор предметной области

1.1 Введение

РЕФАЛ – Рекурсивный функциональный алгоритмический язык [5], язык функционального программирования. Он ориентирован на символьные вычисления, обработку и преобразования текстов. У данного языка есть много диалектов, язык Рефал-5λ [6] – один из них. Он является точным надмножеством языка Рефал-5 [7]. Для простоты далее будем называть Рефал-5λ Рефалом.

Основные расширения этого языка – функции высшего порядка, присваивания, расширенные результатные выражения, абстрактные типы данных, а также нативные вставки кода на языке C++.

1.2 Обзор грамматики языка

1.2.1 Программа

Корневой объект в грамматике – это программа. *Программа* — это последовательность объявлений, определений, директив включения заголовка, вставок нативного кода и точек с запятой (см. Листинг 1).

```
Program ::= Unit
Unit ::=
    Declaration
  | Definition
  | "$INCLUDE" COMPOUND ";"
  | NATIVE-CODE
  | ";"
```

Листинг 1 Синтаксис программы

Ключевые слова со знаком доллара называются *директивами*.

Директива `$INCLUDE` говорит компилятору, что нужно включить текст из заголовочного файла, указанного в директиве. Заголовочные файлы имеют расширение `.refi`, при этом в директиве расширение можно не писать, оно

будет добавлено автоматически, в приоритете расширение `.refi`. Возможна рекурсивная зависимость при включении файлов. Исходный файл после вставки всех заголовочных файлов становится *единицей трансляции*. В каждый исходный файл компилятор неявно добавляет директиву `$INCLUDE`, она подключает заголовочный файл с объявлениями и определениями внутренних функций Рефала.

Имя функции – это *идентификатор*. Идентификатор начинается на латинскую букву или прочерк, состоит из латинских букв, цифр, прочерков или дефисов. Примеры: `__Put-Line, up_Down`.

```
Declaration ::=
    "$EXTERN" NameList
  | "$ENTRY" NameList
  | "$DRIVE" NameList
  | "$INLINE" NameList
  | "$SPEC" Name Pattern ";"
  | "$LABEL" NameList

Name ::= IDENT
NameList ::= Name { "," Name } ";"
```

Листинг 2 Синтаксис объявлений

Объявление состоит из некоторой директивы и списка имён (см. Листинг 2).

Объявление не создаёт нового объекта, а только задаёт свойства имеющимся, либо пополняет область видимости.

Директива `$DRIVE` помечает функции, для которых будет применяться оптимизация прогонки.

Директива `$INLINE` помечает функции, для которых будет применяться оптимизация встраивания.

Директива `$SPEC` помечает функции, для которых будет применяться оптимизация специализации.

Директива $EXTERN добавляет в область видимости текущей единицы трансляции перечисленные имена. Обычно это имена entry-функций, которые описаны в других единицах трансляции. Также можно использовать имена из текущей единицы трансляции. Другие формы записи $EXTERN: $EXTERNAL, $EXTRN.

Директива $ENTRY говорит о том, что перечисленные в ней имена функций являются entry-функциями.

Определение определяет функцию. Синтаксис приведён на листинге 3.

```
Definition ::=
    ["$ENTRY"] Name Body
  | "$ENUM" NameList
  | "$EENUM" NameList
  | "$META" NameList
  | "$SWAP" NameList
  | "$ESWAP" NameList
```

Листинг 3 Синтаксис определения

Конструкция ["$ENTRY"] Name Body – основной способ определения функции. Директива $ENTRY делает функцию entry-функцией. Это делает её доступной для вызова из других единиц трансляции. Если директивы $ENTRY перед определением нет и имя функции не указано в списках $ENTRY, то функция является локальной**.**

Директива $META определяет метафункцию — функцию, которая работает с текущей областью видимости. Например, метафункция Mu вызывает функцию из текущей области видимости по заданному имени.

Метафункция определяется как локальная. В теле метафункции вызывается внешняя функция с именем вида __Meta_Name (где Name — имя метафункции). Этой функции передаётся исходный аргумент и таблица, которая описывает область видимости.

Директива $ENUM определяет пустую локальную функцию (в ней нет предложений). Директива $EENUM определяет пустые entry-функции.

1.2.2 Тело функции

Тело функции отделяется фигурными скобками (см. Листинг 4). Тело содержит либо набор предложений, либо нативную вставку. Если функция содержит набор предложений при вызове будет выполняться первое предложение, которое было сопоставлено с аргументом вызова. Если таких предложений не было найдено, то программа аварийно завершается.

```
Body ::=
    "{" [Sentences] "}"
  | "{" NATIVE-BODY "}"

Sentences ::= Sentence { ";" Sentence } [";"]
```

Листинг 4 Синтаксис тела функции

Предложения разделяются точкой с запятой. В конце последнего предложения можно ставить необязательную точку с запятой.

Предложение – это правило, которое описывает вычисление функции при определённых ограничениях на аргумент (см. Листинг 5).

Предложение состоит из левой и правой части. Правая часть начинается либо со знака равенства, либо с запятой.

Левая часть описывает ограничения, налагаемые на некоторое выражение. Она состоит из образца и из нуля или более условий, перечисленных через запятую.

Образец описывает шаблон – выражение с переменными. Аргумент функции сопоставляется с образцом. Сопоставление выражения с образцом считается успешным, если можно найти такую подстановку переменных образца, которая будет переводить образец в данное выражение.

Условие – описывает дополнительное ограничение. Оно начинается с запятой, и состоит из пары – расширенное результатное выражение в левой части условия и образец в правой части условия, разделяются они двоеточием.

```
Sentence ::= LeftPart RightPart

LeftPart ::= Pattern { Condition }
Condition ::= "," ResultEx ":" Pattern

RightPart ::=
    { Assignment } "=" ResultEx
  | { Assignment } "," ClassicBlock
Assignment ::= "=" ResultEx ":" LeftPart

ResultEx ::= Result { ":" Body }
ClassicBlock ::= Result ":" Body
```

Листинг 5 Синтаксис предложений.

Для проверки условия вычисляется значение её левой части, а затем это значение сопоставляется с правой частью. Если сопоставление успешное, то условие выполнено.

Выражение, переданное в функцию успешно сопоставлено с левой частью, если оно сопоставилось с образцом и со всеми условиями. Условия проверяются последовательно слева направо.

Если левая часть предложения успешно сопоставилась с выражением управление передаётся в правую часть, если нет, то идёт проверка следующего предложения. Если предложение было последним и сопоставление с левой частью было неуспешным, программа аварийно завершается.

Правая часть состоит из нуля или более присваиваний, за которыми следует знак равно и расширенное результатное выражение, либо запятая и классический блок.

При выполнении правой части сначала выполняются присваивания, а затем вычисляется значение расширенного результатного выражения либо классического блока.

Присваивание начинается на знак равенства, за которым следует расширенное результатное выражение и левая часть, разделённые двоеточием.

Вычисляется расширенное результатное выражение и сопоставляется с левой частью. Если сопоставление неуспешное программа аварийно завершается.

Присваивания используют для вычисления некоторого значения и его связи с переменными. Иногда используются для побочных эффектов.

Расширенное результатное выражение состоит из результатного выражения и последовательности тел функций, разделённых двоеточием. Значение результатного выражения последовательно преобразуется вызовами вложенных функций, которые описаны их телами.

Классический блок состоит из результатного выражения и тела функции. Его семантика эквивалентна расширенному результатному выражению с одним телом функции.

Образцовые и результатные выражения могут содержать переменные. Пополняют область видимости переменных только образцы, т.е. левые части, условия и присваивания. Область видимости переменной начинается с точки её появления в предложении и заканчивается в конце предложения. В блоках и вложенных функциях можно использовать все переменные, которые доступны в месте расположения блока.

1.2.3 Объектные выражения

Объектные выражения – это данные языка Рефал. Они состоят из объектных термов (синтаксис в Листинге 6).

```
ObjectExpr ::= { ObjectTerm }
ObjectTerm ::=
    RuntimeSymbol
  | "(" ObjectExpr ")"
  | "[" Name ObjectExpr "]"

RuntimeSymbol ::= CHAR | WORD | NUMBER | FUN-PTR | CLOSURE
```

Листинг 6 Синтаксис объектных выражений.

Объектный терм – это либо символ, либо составной терм. *Символ* – неделимый элемент данных. Неделимость означает, что его нельзя разделить на переменные при сопоставлении с образцом. Во время исполнения программы символ может быть числом, словом, литерой, указателем на функцию или замыканием. *Скобочный терм* составляется из скобок двух видов: круглые структурные скобки и квадратные АТД-скобки (АТД- абстрактный тип данных).

После открывающейся квадратной скобки должно стоять имя функции из текущей области определения. Если функция локальная, то такой терм можно использовать в результатном выражении или сопоставить его с образцом, только в той единице трансляции, в которой этот терм был объявлен. В других единицах трансляции этот терм может быть сопоставлен только с t-переменной. Таким образом реализуется инкапсуляция данных в Рефале.

Для того, чтобы создать терм с АТД-скобками можно использовать имя любой локальной функции. Обычно для этого используется директива `$ENUM`.

1.2.4 Образцовые выражения

Образец или *образцовое выражение* определяет некоторое множество объектных выражений (синтаксис на Листинге 7).

```
Pattern ::= { PatternTerm }
PatternTerm ::=
    Symbol
  | VARIABLE ["^"]
  | "(" Pattern ")"
  | "[" Name Pattern "]"

Symbol ::= COMPOUND | IDENT | "&" Name | CHAR | NUMBER
```

Листинг 7 Синтаксис образцовых выражений.

Образец определяется как последовательность символов, переменных, скобочных термов или АТД-скобок.

Символ определяется как составной символ, идентификатор, указатель на функцию, литера или число.

С помощью идентификатора и *составного символа* можно записать символ-слово – последовательность знаков, которую можно записать как единое целое. Составной символ записывается как строка в двойных кавычках. Она может содержать escape-последовательности (`\'`, `\"`, `\\`, `\n` и т.д.).

Если символ-слово удовлетворяет требованиям для идентификатора, то он может быть записан как в двойных кавычках, так и без.

Литера – печатный знак из некоторого символьного множества. В данной реализации Рефала поддерживаются символы ASCII. Литера записывается в одинарных скобках. Несколько литер могут быть записаны, как составной символ, только с одинарными кавычками вместо двойных. Они будут обозначать последовательность одиночных литер.

Число или *макроцифра* – это целое число в интервале $[0, 2^{32} - 1]$.

Переменная записывается как `type.index`, где `type` – один из знаков `s`, `t` или `e`, а `index` - последовательность латинских букв, цифр, знаков прочерка и дефиса и не может начинаться на дефис. Переменная описывает некоторую часть объектного выражения. Существует три типа переменных: s-переменные соответствуют символам, t-переменные термам и e-переменные произвольным выражениям (пустые выражения также допустимы).

В Рефале есть возможность переопределить переменную, которая была связана с некоторым значением ранее в предложении. Для того, чтобы переопределить переменную нужно поставить символ “^” после него. Переопределение меняет значение переменной на новое.

Это бывает нужно в случаях, когда переменная с некоторым индексом была использована ранее, но её значение уже не нужно, и её нужно использовать в присваивании, условии или блоке. Такие переменные называются *переопределёнными*. Если в образце переопределённая переменная встречается несколько раз, то знак “^” нужно ставить только после первого вхождения.

Если индекс переменной начинается со знака прочерка, то переменная является *безымянной*. Значения безымянных переменных не должны быть равны, хотя их индексы равны. Знак “^” после безымянных переменных запрещён.

Если подставить в образец вместо переменных значения соответствующих типов, причём переменные с одинаковым индексом заменяются на одинаковые значения, то получится некоторое объектное выражение. Таким образом, образец описывает множество объектных выражений, получаемых подстановкой значений в переменные.

Подстановка значений в образец должна удовлетворять определённым требованиям:

1. Подстановка переменных в левую часть предложения должна превращать образец в аргумент функции.
2. Значения, которые подставляются в переменные, должны совпадать с типами переменных. S, t и e можно заменять только на символы, термы и произвольные выражения.
3. Если в образце несколько раз встречается переменная с одним и тем же именем, то все они должны заменяться на одно и то же выражение.
4. Если существует несколько переменных, которые удовлетворяют требованиям 1-3, то выбрать нужно ту, у которой самое левое вхождение e-переменной – кратчайшее. Если неопределённость не разрешилась, то надо перейти к следующей e-переменной и так до тех пор, пока неопределённость не разрешится.

1.2.5 Результатные выражения

Результатное выражение определяет объектное выражение. Оно представляет собой последовательность результатных термов (см. Листинг 8).

```
Result ::= { ResultTerm }
ResultTerm ::=
    Symbol
  | VARIABLE
  | "(" Result ")"
  | OpenCall Result ">"
  | "[" Name Result "]"
  | "{" Body "}"
OpenCall ::= "<" [Name] | "<+" | "<-" | "<*" | "</" | "<%"
| "<?"

Symbol ::= IDENT | "&" Name | COMPOUND | CHAR | NUMBER
```

Листинг 8 Синтаксис результатных выражений.

Каждый из термов может быть символом, переменной, скобочным термом, термом в АТД-скобках, вызовом функции и вложенной функцией. В результатных выражениях не могут быть безымянные переменные.

Вызовы функций записываются в угловых скобках. Если после открывающейся угловой скобки находится имя, то оно должно быть именем функции из текущей области видимости. Если имя не указано, то функция определяется во время выполнения программы. В таком случае первый терм после открывающейся угловой скобки – это либо указатель на функцию, либо объект замыкания.

Вложенная функция записывается как тело функции в фигурных скобках. При выполнении программы она превращается в объект-замыкание. Такой объект может быть сопоставлен только с s-переменной. Вызвать замыкание можно, записав его после открывающейся угловой скобки.

1.3 Абстрактная рефал-машина

Программу на Рефале выполняет абстрактная *рефал-машина* – воображаемая вычислительная машина, которая понимает синтаксис Рефала. У этой машины есть два раздела памяти: поле программ, которое хранит определения функций программы, и поле зрения, которое хранит текущее

состояние вычислений. Состояние вычислений определяется как активное выражение – выражение языка Рефал, содержащее угловые скобки, но при этом оно не содержит переменных.

Рефал-машина выполняет программу пошагово. Каждый шаг – выполнение следующей последовательности действий.

1. Рефал-машина ищет в поле зрения самую левую пару угловых скобок, такую что внутри неё нет других угловых скобок. Такой участок поля зрения называется первичным активным подвыражением.
2. Рефал-машина смотрит, что находится справа от левой скобки активации: там должно располагаться имя функции. Если его там нет, то рефал-машина останавливается с ошибкой «отождествление невозможно».
3. Рефал-машина находит имя функции в поле программ. Функция может быть написана на Рефале, или быть встроенной. Если функция встроенная, то рефал-машина передаёт управление на процедуру на машинном языке, реализующую данную функцию. Если эта функция написана на Рефале, то машина передаёт управление на первое предложение функции.
4. Если можно сопоставить аргумент функции и образец из левой части текущего предложения, то выполняется пункт 5. Иначе управление передаётся на следующее предложение, пункт 4 повторяется. Если предложения закончились, то рефал-машина останавливается с ошибкой «отождествление невозможно».
5. Значения, которые были подставлены после сопоставления подставляются в правую часть предложения. Рефал-машина заменяет первичное активное подвыражение в поле зрения на полученное после подстановки выражение.
6. Если в поле зрения остались угловые скобки вызова, рефал-машина переходит к шагу 1. Иначе рефал-машина успешно завершается.

Из этого алгоритма следует, что Рефал – аппликативный язык, порядок вычислений в нём строго определён: слева-направо. Также отсюда следует, что Рефал реализует оптимизацию хвостовой рекурсии.

1.4 Архитектура компилятора Рефала

Компилятор Рефала осуществляет компиляцию в несколько проходов.

1. *Лексический анализ*. На этом проходе компилятор получает на вход файл с текстом программы. На выходе после анализа получается последовательность токенов. Токены описываются типом, позицией в тексте программы и значением.
2. *Синтаксический анализ*. На этом проходе происходит разбор последовательности токенов с целью построить абстрактное синтаксическое дерево [8].
3. *Разрешение подключаемых файлов*. Данный проход ищет в исходных текстах директивы `$INCLUDE` и загружает содержимое указанных файлов. Для загруженных файлов исполняются проходы 1-3. После создания всех синтаксических деревьев строится результирующее дерево, которое равно их конкатенации. В этом дереве изменяются позиции токенов и удаляются узлы, связанные с директивой `$INCLUDE`. В этом проходе также создаётся функция-мета-таблица. Она используется в реализации метафункций.
4. *Проверка контекстных зависимостей*. Этот проход проверяет контекстно-зависимые ошибки после всех предыдущих проходов.
5. *Обессахаривание*. Многие конструкции Рефала являются синтаксическим сахаром. На этом этапе эти конструкции преобразуются в комбинации более простых конструкций. Например, переопределённые и безымянные переменные переименовываются, присваивания превращаются в блоки, блоки превращаются в вызовы

вложенных функций, вложенные функции превращаются в глобальные функции и операции создания замыканий.

6. *Высокоуровневая оптимизация*. На этом проходе синтаксическое дерево преобразуется с целью оптимизировать программу. Например, на этом этапе реализованы оптимизации прогонки и специализации. Выходное дерево отличается от входного отсутствием пометок об оптимизации.
7. *Генерация промежуточного кода на RASL*. На данном этапе на основе абстрактного синтаксического дерева генерируется код на промежуточном языке RASL (Refal Assembly Language).
8. *Генерация целевого кода.* На данном этапе генератор кода превращает все команды промежуточного кода в соответствующий код на C++.

1.5 Встроенные функции Рефала

Рассмотрим встроенные функции Рефала, которые можно вычислить во время компиляции.

Функция Mu. Формат этой функции приведён на листинге 9. Эта функция принимает на вход функцию и аргумент и возвращает значение, которое было получено при вызове функции с данным аргументом. Функцию можно определить пятью способами: указать имя функции как идентификатор или строку литер в круглых скобках, использовать арифметический символ в одинарных или двойных кавычках, записать функцию в объекте замыкания или использовать указатель на функцию. Если функция указана через её имя и в текущем файле не нашлось функции с таким именем, возникает ошибка во время выполнения программы.

Функция Mu является метафункцией. Она определяется через директиву `$META`. При компиляции директива `$META Mu;` создаёт функцию __Meta_Mu (см. Листинг 10). Этой функции дополнительно передаётся мета-таблица –

таблица пар «имя-указатель на функцию». Вся логика функции Mu определена в функции __Meta_Mu.

```
<Mu t.MuFunc e.Arg> == e.Res

t.MuFunc ::= s.WORD | (s.CHAR+) | s.FUNCTION | s.CLOSURE | s
.Op

s.Op ::=                    '+' | '/' | '%' | '*' | '?' | '-'
| "+" | "/" | "%" | "*" | "?" | "-"
```

Листинг 9 Формат функции Mu.

```
$EXTERN __Meta_Mu, __Step-Drop;
$INLINE Mu;

Mu {
  e.Arg = <__Step-Drop> <__Meta_Mu e.Arg $table>
}
```

Листинг 10 Определение функции Mu.

Функции `Add, Sub, Mul, Div, Mod`. Эти функции возвращают сумму, разность, произведение, неполное частное от деления и остаток от деления своих аргументов соответственно. Формат функции Add приведён на листинге 11. Форматы остальных функций аналогичны.

```
<Add t.FirstNumber e.SecondNumber> == e.NormedNumber

t.FirstNumber ::= s.NUMBER | ({'+'|'-'}? s.NUMBER+)

e.SecondNumber ::= {'+'|'-'}? s.NUMBER+

e.NormedNumber ::= '-'? s.NUMBER+
```

Листинг 11 Формат функции Add

Функция Divmod. Данная функция возвращает неполное частное от деления и остаток от деления своих аргументов. Формат функции приведён на листинге 12.

Функция Compare. Данная функция принимает на вход два числа и возвращает ‘+’, если первое число больше второго, ‘-‘ если первое число меньше

второго, и ‘0’ если они равны. Формат этой функции: `<Compare t.FirstNumber e.SecondNumber> == '-' | '0' | '+'`

```
<Divmod (e.FirstNumber) e.SecondNumber> == (e.Quotient) e.Remainder

e.Quotient, e.Remainder ::= e.NormedNumber
```

Листинг 12 Формат функции Divmod

Функция Chr. Данная функция принимает на вход последовательность макроцифрам и возвращает последовательность литер, которые соответствуют этим макроцифрам по таблице ASCII. Формат этой функции: `<Chr e.Expr> == e.Expr'`

Функция Ord. Данная функция аналогична Chr, выполняет обратное действие. Формат функции: `<Ord e.Expr> == e.Expr'`.

Функция Upper. Данная функция принимает на вход последовательность литер, а возвращает те же литеры, но переводит их в верхний регистр. Эта функция обратная к Lower. Формат функции: `<Upper e.Expr> == e.Expr'`.

Функция Lower. Данная функция принимает на вход последовательность литер, а возвращает те же литеры, но переводит их в нижний регистр. Эта функция обратная к Upper. Формат функции: `<Lower e.Expr> == e.Expr'`.

Функция Numb. Данная функция берёт на вход строку, представляющую некоторую макроцифру, и возвращает макроцифру. Формат данной функции:

```
<Numb {'+'|'-'}? s.DIGIT-CHAR* e.NotDigitChars> == '-
'? s.MACRODIGIT+
```

Функция Symb. Эта функция обратная к Numb. Она берёт на вход макроцифру и возвращает строку, представляющую эту макроцифру. Формат функции на листинге 13.

```
<Symb e.Sign s.NUMBER+> == e.Sign s.CHAR-DIGIT+

e.Sign ::= {'+'|'-'}?
```

Листинг 13 Формат функции Symb

Функция Implode. Эта функция принимает на вход последовательность литер и последовательность других символов, которая игнорируется. Функция создаёт идентификатор из входных литер и возвращает его. Строка из литер должна удовлетворять требованиям идентификатора. Если первый символ не является литерой, то функция возвращает 0 и аргумент. Формат функции на листинге 14.

```
<Implode e.NameChars e.NotNameTerms> == s.COMPOUND e.NotName
Terms

<Implode e.NotNameTerms> == 0 e.NotNameTerms
```

Листинг 14 Формат функции Implode

Функция Explode. Эта функция принимает на вход составной символ и возвращает строку литер, из которых он состоит. Формат функции: `<Explode s.COMPOUND> == s.CHAR*`

Функция First. Эта функция принимает макроцифру N и некоторое выражение. Функция разделяет выражение на две части: префикс и суффикс. В префиксе первые N термов, в суффиксе остальные символы. Если N больше длины входного выражения, то суффикс пустой. Формат функции приведён на листинге 15.

```
<First s.N e.Prefix e.Suffix> == (e.Prefix) e.Suffix, |e.Pre
fix| == s.N

<First s.N e.Expr> == (e.Expr), |e.Expr| < s.N

s.N ::= s.NUMBER
```

Листинг 15 Формат функции First

Функция Last. Эта функция аналогична функции First, только здесь N термов в суффиксе, а не в префиксе. Формат функции приведён на листинге 16.

```
<Last s.N e.Prefix e.Suffix> == (e.Prefix) e.Suffix, |e.Suff
ix| == s.N

<Last s.N e.Expr> == () e.Expr, |e.Expr| < s.N

s.N ::= s.NUMBER
```

Листинг 16 Формат функции Last

Функция Lenw. Эта функция принимает на вход выражение, возвращает количество термов в нём и само выражение. Формат функции: `<Lenw e.Expr> == s.N e.Expr, where s.N == |e.Expr|`

Функция Type. Эта функция принимает на вход выражение, возвращает тип и подтип его первого символа, а также само выражение. Формат функции приведён на листинге 17.

```
<Type e.Expr> == s.Type s.SubType e.Expr

s.Type и s.SubType описывают начало e.Expr

s.Type s.SubType ::=

    'Lu' – латинский символ в верхнем регистре

    | 'Ll' – латинский символ в нижнем регистре

    | 'D0' – десятичная цифра

    | 'Wi' -- идентификатор (составной символ, который
записан без двойных кавычек)

    | 'Wq' – другой составной символ (должен быть в двойных
кавычках)

    | 'N0' -- макроцифра

    | 'Pu' – печатный символ в верхнем
регистре (isupper(c) возвращает true)
```

```
    | 'Pl' -- печатный символ в нижнем регистре
    | 'Ou' -- другой символ в верхнем регистре (isupper(c)
возвращает true)
    | 'Ol' -- другой символ в нижнем регистре
    | 'B0' – структурные скобки (круглые)
    | '*0' -- e.Expr пустое выражение
    | 'H0' – файловый дескриптор (указатель, тип
refalrts::Node::file_info)
    | 'Fg' – глобальная функция
    | 'Fc' – замыкание захватывающее контекст
    | 'Ba' – АТД-скобки (квадратные скобки)
```

Листинг 17 Формат функции Type

2 Постановка задачи

2.1 Создание директивы

Нужно добавить в язык Рефал директиву `$INTRINSIC`, которая будет разрешать оптимизацию некоторых встроенных функций. Пример использования новой возможности приведён на листинге 18. Функции, которые были объявлены данной директивой будем называть интринсиками. Этой директивой нужно будет объявить в компиляторе все функции, для которых были реализованы оптимизации.

```
$INTRINSIC Add, __Meta_Mu;
```

Листинг 18. Пример нового синтаксиса

2.2 Встроенные функции

Оптимизация для функции `__Meta_Mu`:

Вызов этой функции должен заменяться на непосредственный вызов функции переданной первым аргументом. Если именем функции является «настоящая функция» — указатель на функцию (`&Func`) или замыкание (вложенная функция), то её вызов заменяется на вызов функции или замыкания. Если именем функции является идентификатор (символ-слово) или скобочный терм с цепочкой литер внутри, то функция вызывается по указателю, соответствующему данному имени в метатаблице. Имя — соответственно идентификатор (`<__Meta_Mu Func e.Arg $table>`) или строка в скобках (`<__Meta_Mu ('Func') e.Arg $table>`). Также должны оптимизироваться случаи вызовов функций “+”, ‘+’ и т.п. синонимов для соответствующих арифметических функций. Примеры: `<Mu Add 1 2>`, `<Mu &Add 1 2>`, `<Mu (‘Add’) 1 2>`, `<Mu {s.1 s.2 = <Add s.1 s.2>} 1 2>` заменяются на `<Add 1 2>`.

Оптимизация для арифметических функций `Add, Div, Divmod, Mod, Mul, Sub, Compare`:

Оптимизации подлежат вызовы, в которых оба аргумента константные, т.е. их вызов заменяется на результат вычисления. Также оптимизации подлежат вызовы, результат которых заранее известен, например, в случаях `<Add e.X 0>, <Sub e.X 0>, <Div e.X 1>, <Mul e.X 1>, <Divmod e.X 1>`. Но оптимизации подлежат такие вызовы, где e.X является вызовом другой арифметической функции, иначе оптимизация может скрыть ошибку, которая могла возникнуть без оптимизации. Примеры: `<Add 0 <Mul 3 e.z>>` заменяется на `<Mul 3 e.z>`, `<Divmod 0 255>` заменяется на `(0) 0`.

Оптимизация для функций `Chr, Ord, Upper, Lower`:

При оптимизации входные символы нужно заменять на результат функции, переменные заменять на вызовы этих функций, скобки не изменяются. Пример: `<Lower 'L' e.y 'R'>` заменяется на `'r' <Lower e.y> 'r'`.

Оптимизация для функций `Explode, Implode, Numb, Symb, Implode_Ext, Explode_Ext`:

Эти функции нужно оптимизировать, если они принимают константный аргумент. Пример: `<Implode 'Ident'>` заменяется на `Ident`.

Оптимизация для функции `First, Last, Lenw`:

`First, Last` принимают аргументы `s.N` и `e.Expr`, они оптимизируются, когда `s.N` константа, а `e.Expr` не содержит e-переменных на верхнем уровне. Для `Lenw` оптимизация аналогична, только она не принимает численного аргумента. Пример: `<First 2 ('f') 'c' L>` заменяется на `(('f') 'c') L`.

Оптимизация для функции `Type`:

Результат этой функции зависит от первого элемента, если он константен, то вызов данной функции при оптимизации заменяется на результат вычисления. Пример: `<Type 'F' e 3>` заменяется на `'LuF' e 3`.

2.3 Модификация библиотеки

В Рефале-5 есть библиотека LibraryEx [9], в которой используются функции высшего порядка. Функция Apply из этой библиотеки использует функцию Mu. А сама функция Apply используется такими функциями, как Map, MapAccum, Reduce. Эти функции используются в самом компиляторе повсеместно, поэтому они оптимизируются методами прогонки и специализации. Эти методы оптимизации работают, когда в Apply происходит непосредственный вызов функции, а не через функцию Mu. Но функцию Mu нужно оставить в LibraryEx для Рефала-5λ для совместимости с набором библиотек для Рефала-5, а также для того, чтобы этой библиотекой можно было пользоваться, не зная функций высшего порядка, т.е. зная только Рефал-5. Поэтому необходимо добавить функцию Mu в Apply и объявить встроенной функцию __Meta_Mu, чтобы компилятор оптимизировал вызовы Mu.

3 Разработка

3.1 Лексический анализ

В компилятор нужно внести новую директиву. На стадии лексического анализа нужно распознать токен `$INTRINSIC`. Изменению подлежит функция DoScan-BuildKeyword (см. Листинг 19). Таким образом был добавлен новый тип токена TkIntrinsic.

```
DoScan-BuildKeyword {
      t.Pos (e.KeywordChars) (e.Line) e.Lines
        , (TkExtern '$EXTERN') /* другие директивы */
        (TkIntrinsic '$INTRINSIC')
        : e.Keywords-B (s.Tag e.KeywordChars) e.Keywords-E
        = (s.Tag t.Pos)
          <DoScan <IncCol t.Pos Len e.KeywordChars>
          (e.Line) e.Lines>;
}
```

Листинг 19. Обнаружение директивы $INTRINSIC

3.2 Синтаксический анализ

Синтаксически новая директива не отличается от таких директив, как `$EXTERN, $ENUM`. Нужно создать узел в абстрактном синтаксическом дереве, соответствующий объявлению директивы. Для этого достаточно написать функцию, которая по токену типа TkIntrinsic создаёт соответствующий узел, добавить токен TkIntrinsic в функцию, которая описывает директивы со списком имён (например, `$EXTERN`), а также добавить TkIntrinsic в функцию, определяющую строчное представление токена (см. Листинг 20).

```
NameListTags {
       = TkExtern TkEnum /* другие токены */ TkIntrinsic
}
TkIntrinsic {
       t.Pos e.Name = (Intrinsic t.Pos GN-Local e.Name)
}
TokName {
     /* остальные токены */
     TkIntrinsic = '$INTRINSIC';
}
```

Листинг 20 Синтаксический анализ объявления директивы

3.3 Обессахаривание

Директивы `$DRIVE, $INLINE, $INTRINSIC` имеют иерархию. Если одно и то же имя было объявлено, как `$INLINE` и `$INTRINSIC`, то в синтаксическом дереве остаётся только объявление `$INLINE`. Если имя было объявлено, как `$DRIVE` и `$INTRINSIC`, остаётся `$DRIVE`. Если имя было объявлено, как `$DRIVE` и `$INLINE`, остаётся `$DRIVE`. Эта логика реализована во время прохода обессахаривания в функции Pass-RemoveRedundantDriveInlineIntrinsic (см. листинг 21).

```
Pass-RemoveRedundantDriveInlineIntrinsic {
  e.AST
    = <MapAccum
        {
```

```
          (e.Labels-B (e.Name s.KnownLabel) e.Labels-
E) (Drive e.Name)
            = (e.Labels-B (e.Name Drive) e.Labels-E)
              (Drive e.Name);
          (e.Labels-B (e.Name Drive) e.Labels-
E) (Inline e.Name)
            = (e.Labels-B (e.Name Drive) e.Labels-E)
              (Inline e.Name);
         (e.Labels-B (e.Name Intrinsic) e.Labels-
E) (Inline e.Name)
            = (e.Labels-B (e.Name Inline) e.Labels-E)
              (Inline e.Name);
          (e.Labels-B (e.Name s.KnownLabel) e.Labels-
E) (Intrinsic e.Name)
            = (e.Labels-B (e.Name s.KnownLabel) e.Labels-E)
              (Intrinsic e.Name);

          (e.Labels) (Drive e.Name)
            = (e.Labels (e.Name Drive)) (Drive e.Name);

          (e.Labels) (Inline e.Name)
            = (e.Labels (e.Name Inline)) (Inline e.Name);

          (e.Labels) (Intrinsic e.Name)
            = (e.Labels (e.Name Intrinsic)) (Intrinsic e.Name);
          (e.Labels) t.Other = (e.Labels) t.Other;
```

```
        }
        (/* labels */) e.AST
      >
    : (e.Labels) e.AST^
    = /* здесь пропущен код для краткости. Логика: удаляем метки,
которых нет в e.Labels */
    = e.AST;
}
```

Листинг 21 Проход обессахаривания

3.4 Оптимизация

Оптимизация интринсиков похожа на оптимизацию встраивания — нужно будет обнаруживать вызовы интринсиков, проверять их аргументы, и если аргумент допустимый, то вызов будет заменяться на результат выполнения. Оптимизация встраивания является разновидностью оптимизации прогонки. Обработку интринсиков было решено реализовать в одном проходе с прогонкой.

3.4.1 Вспомогательная информация для оптимизации

В функцию осуществляющую прогонку DriveInlineOptimizerTrick (см. Листинг 22) нужно добавить переменную s.IntrinsicMode, в которую передаётся значение OptIntrinsic, если флаг –Oi включён и NoOpt в ином случае. Вся логика прогонки определена в OptSentence-MakeSubstitutions и до этой функции нужно передать переменную s.IntrinsicMode, чтобы в ней можно было определить, оптимизировать внутренние функции или нет.

```
DriveInlineOptimizerTick {
  t.OptInfo e.AST s.OptDrive s.OptIntrinsic
```

```
    = (<OptSwitch s.OptDrive> <OptSwitch s.OptIntrinsic>)
    : t.Mode
    = <MapAccum
        {
          t.OptInfo^ (Function s.ScopeClass (e.Name)
          Sentences e.Sentences)
            = <OptFunction
                 (Function s.ScopeClass (e.Name)
                 Sentences e.Sentences)
                 t.OptInfo
                 t.Mode
               >;
          t.OptInfo^ t.Other = t.OptInfo t.Other
        }
        t.OptInfo
        e.AST
      >
    : (e.OptInfo) e.AST^ /* остальной код пропущен */
```

Листинг 22. Функция, осуществляющая прогонку, встраивание и оптимизацию интринсиков

Перед запуском оптимизации в синтаксическое дерево добавляется узел DriveInfo, в котором содержатся данные об оптимизации (см. листинг 23). Функцию UpdateDriveInfo, осуществляющую добавление этой информации, нужно модифицировать, чтобы она могла хранить метатаблицы и встроенные функции (описание модификации на листинге 24).

```
e.DriveInfo ::= (e.OptFuncNames (e.OptFunctions))
e.OptFuncNames ::= (e.Name)*
e.OptFunctions ::= t.OptFunction
t.OptFunction ::= (s.Label s.ScopeClass (e.Name) e.OptBody)
e.OptBody ::= Sentences e.Sentences
```

Листинг 23. Описание DruveInfo

```
t.OptFunction ::= (s.Label s.ScopeClass (e.Name) e.OptBody)
s.Label ::= Drive | Inline | Intrinsic | Metatable
  e.OptBody ::=
      Sentences e.Sentences
    | Intrinsic e.Name
    | Metatable e.Metatable
```

Листинг 24. Обновление DriveInfo

3.4.2 Логика оптимизации

В каждом проходе прогонки из каждого результатного выражения выбирается самый левый терм вызова оптимизируемой функции. И он либо оптимизируется, либо заменяется на «холодный вызов», благодаря чему будет пропускаться на следующих проходах. Так сделано для того, чтобы компилятор работал конечное время.

Вызов оптимизируемой функции обрабатывается в функции OptSentence-MakeSubstitutions. В неё нужно добавить предложение, которое будет соответствовать оптимизации интринсиков (см. листинг 25). Также в эту функцию был добавлен аргумент t.Metatables, который содержит список всех метатаблиц. Эта переменная нужна при оптимизации __Meta_Mu. Метатаблицы извлекаются из DriveInfo до вызова OptSentence-MakeSubstitution.

Опишем аргументы функции OptSentence-MakeSubstitution:

- t.Mode — это структура (s.DriveMode s.IntrinsicMode), по этому аргументу функция определяет, нужно ли оптимизировать внутренние функции или осуществлять прогонку.
- e.OriginSentence — предложение, в котором находится вызов оптимизируемой функции.
- e.CallArgs — аргументы оптимизируемой функции.
- t.OptFunction — оптимизируемая функция, её тип соответствует тому, что описано в DriveInfo

Эта функция возвращает набор решений:

- e.SolutionsPack — решение для одного из предложений прогоняемой функции.
- e.ReplacedExpr — то, на что нужно заменить вызов оптимизируемой функции.
- e.Contractions — сужения, применяемые к образцу.
- e.Assignments — присваивания, применяемые к e.ReplacedExpr.
- e.NewFunctions — новая функция, которая могла появиться во время прогонки/встраивания.

В случае оптимизации встроенных функций сужений быть не должно, присваивания не нужны, пакет решений должен быть один. Т.е. результат оптимизации должен выглядеть как: `((новое выражение)((/* нет сужений */) (/* нет присваиваний */)) (/* нет новых функций */))`

В случае, когда оптимизация невозможна, вызов оптимизируемой функции заменяется на холодный вызов с тем же аргументом — он генерируется при помощи MakeColdSolution.

```
/* Формат функции

  <OptSentence-MakeSubstitutions
```

```
    t.Mode (e.OriginSentence) (e.CallArgs) t.OptFunction
  t.Metatables> == (e.SolutionsPack)*
  e.SolutionsPack ::= (e.ReplacedExpr)
                      t.Solution* (e.NewFunctions)
  t.Solution ::= (e.Contractions) (e.Assignments)
*/
OptSentence-MakeSubstitutions {
  (s.DriveMode Intrinsic)
  ((e.Left) (e.Expr)) (e.Args)
  (Intrinsic s.ScopeClass (e.IntrinsicName)
  Intrinsic e.BehaviorName) t.Metatables
    = (Intrinsic s.ScopeClass (e.IntrinsicName)
      Intrinsic e.BehaviorName) : t.IntrinsicFunction
    = <DoOptSentence-MakeSubstitutions-Intrinsic
    t.IntrinsicFunction t.Metatables (e.Args)
e.BehaviorName>
  t.Mode
  ((e.Left) (e.Expr)) (e.Args)
  t.Function t._
    /* Активные вызовы игнорируем */
    , <IsPassiveCall e.Args> : False

    = <MakeColdSolution t.Function e.Args>;
  (s.DriveMode s.IntrinsicMode)
  ((e.Left) (e.Expr)) (e.Args)
  (s.FuncMode s.ScopeClass (e.Name) Sentences e.Body) t._
```

```
    , <OneOf s.DriveMode Drive Inline> : True
    =/*правая часть пропущена*/;
  t.Mode
  ((e.Left) (e.Expr)) (e.Args)
  t.Function t._
    = <MakeColdSolution t.Function e.Args>;
}
```

Листинг 25 Функция OptSentence-MakeSubstitutions

Оптимизация Mu. Эта функция оптимизируется, если включены флаги –OiI или –OiD, т.к. сначала она встраивается, а затем заменяется при оптимизации внутренней функции __Meta_Mu. Если функция в аргументе определена, как указатель на функцию, или задана объектом замыкания, то при оптимизации заменяем вызов __Meta_Mu на вызов функции (эта логика описана в строчках 8-11 в листинге 26). Когда функция указана одиночным арифметическим символом в двойных или одинарных кавычках (‘+’, “+” и т.д.), вызов заменяется на результат вычисления соответствующих внутренних арифметических функций (Add, Sub и т.д.). В случаях, когда функция указана, как идентификатор или как строка литер в скобках, в метатаблице под названием e.MetatableName ищется указатель на функцию по соответствующему имени (эта логика описана в строчках 12-13 листинга 26). Затем вызов __Meta_Mu заменяется на вызов функции по этому указателю.

```
DoOptSentence-MakeSubstitutions-Intrinsic {
  t.IntrinsicFunction t.Metatables (e.Args) e.BehaviorName =
  e.BehaviorName : {
    '__Meta_Mu' =
      e.Args : {
```

```
      t.Function e.MuArg (Symbol Name e.MetatableName) =
      t.Function : {
            (Symbol Name e.Name) = (
            ((CallBrackets (Symbol Name e.Name) e.MuArg))
            (()()) ());
         (ClosureBrackets e.Body) = /**/;
          (Symbol Identifier s.Name), <OneOf s.Name '+-
     /%*?'> : True = /**/;
          (Symbol Char s.Name), <OneOf s.Name '+-/%*?'> : True
     = /**/;
     (Symbol Identifier e.Name) = /**/;
     (Brackets e.Chars) = /**/;
      e._ = <MakeColdSolution t.IntrinsicFunction e.Args>
}
```

Листинг 26 Оптимизация __Meta_Mu

Оптимизация Add.

Аргументы данной функции имеют три формы. Когда первый аргумент написан в скобках, а второй без, проверяем оба аргумента: в нём должны быть только символы-макроцифры. Затем мы складываем два этих числа, и возвращаем результат как замену (реализация на листинге 27).

Когда первый аргумент является макроцифрой 0, а второй вызовом некоторой арифметической функции, как результат возвращаем вызов этой функции.

Когда оба аргумента являются макроцифрами с возможными перед ними знаками + и -, мы вычисляем функцию Add с соответствующими аргументами и возвращаем результат – макроцифру с возможной перед ним знаком + или -.

Оптимизации Sub, Mul, Div, Divmod, Mod, Compare аналогичны.

```
'Add' = e.Args : {
      (Brackets e.Numbers1) e.Numbers2,
      <Number-Check e.Numbers1> : True, <Number-
Check e.Numbers2> : True
        = <Add (<SymbolsToNumber e.Numbers1>)
         <SymbolsToNumber e.Numbers2>> : e.Result
        = <NumberToSymbols e.Result> : e.Result^
        = ((e.Result) (()()) ());
           (Symbol Number 0) (ColdCallBrackets
           (Symbol Name e.Name) e.Body),
           <OneOf (e.Name) <ArithmeticFunctions>> : True =
             (((ColdCallBrackets (Symbol Name e.Name) e.Body))
               (()()) ());
           /*подобное предложение, симметричное предыдущему*/
           e.Sign1 (Symbol Number s.1) e.Sign2
          (Symbol Number s.2) =
           e.Sign1 : {
             (Symbol Char s.sign1)
             = e.Sign2 : {
               (Symbol Char s.sign2)
                 = <Add s.sign1 s.1 s.sign2 s.2> :
                 e.ResSign s.Res
                 = <SignedArithmeticResult s.Res e.ResSign>;
                 = <Add s.sign1 s.1 s.2> : e.ResSign s.Res
```

```
                = <SignedArithmeticResult s.Res e.ResSign>
            }; /*обработка случая, когда знака нет*/}
          };
          e._  = <MakeColdSolution t.IntrinsicFunction e.Args>
        };
```

Листинг 27 Оптимизация Add

Оптимизация Chr.

Сначала мы проверяем аргумент – в нём должны присутствовать только числа, переменные и вызовы функций (см. листинг 28). Затем осуществляем замены. Для чисел вызываем функцию Chr и вставляем в замену литеру, для переменных вставляем в замену вызов функции от переменной, вызовы функций окружаем вызовом функции Chr.

Оптимизации Ord, Upper, Lower аналогичны.

```
DoOptSentence-MakeSubstitutions-Intrinsic {
  t.IntrinsicFunction t.Metatables (e.Args) e.BehaviorName =
  e.BehaviorName : {
        /* Другие оптимизации */
        'Chr'
        , <CheckArgs-Intrinsic-
StaticSubstitution (e.Args) Number> : True
            = <StaticSubstitution (e.Args)
              ('Chr') &Chr Number Char> : e.Substitution
            = (
               (e.Substitution) (()()) ()
            );
```

```
    }
}
StaticSubstitution {
  (e.Args) (e.FuncName) s.Func s.SymbolFrom s.SymbolTo =
  <MapAccum {
      (e.Result)(Symbol s.SymbolFrom s.n) e.Rest =
      (e.Result (Symbol s.SymbolTo <s.Func s.n>)) e.Rest;
      (e.Result) (TkVariable e.Var) e.Rest =
      (e.Result (ColdCallBrackets (Symbol Name e.FuncName)
      (TkVariable e.Var))) e.Rest;
      (e.Result) (ColdCallBrackets e.Body) e.Rest =
      (e.Result (ColdCallBrackets (Symbol Name e.FuncName)
      (ColdCallBrackets e.Body))) e.Rest
  }
    () e.Args
  > : (e.Result) e._
  = e.Result
}
```

Листинг 28 Оптимизация Chr

Оптимизация Numb.

Сначала проверяем аргументы: они должны быть литерами (см. листинг 29). Затем извлекаем из узлов дерева литеры, соединяем их и вызываем для них функцию Numb. Результат функции конвертируем в узел дерева и вставляем в замену.

Оптимизации Symb, Implode, Explode, Implode_Ext, Explode_Ext, аналогичны.

```
'Numb'
    , <CheckArgs-Intrinsic (e.Args) Char> : True
      = <NodeToChar e.Args> : e.Chars
      = <NumberToSymbols <Numb e.Chars>> : e.Result
      = (
        (e.Result) (()()) ()
      );
```

Листинг 29 Оптимизация Numb

Оптимизация Type.

Отделяем первый элемент аргумента от остальных, вычисляем функцию Type от этого элемента. Результат функции конвертируем в узел дерева. Затем конкатенируем результат функции Type и аргументы оптимизируемой функции в замене (см. листинг 30).

```
'Type'
    = e.Args : {
      (Symbol Number s.Num) e.Rest =
      <Type s.Num> : s.Type s.SubType e._
      = <CharToNode s.Type s.SubType> : e.Result
      = (
        (e.Result (Symbol Number s.Num) e.Rest) (()()) ()
      );
      (Symbol Char s.Char) e.Rest = /*реализация скрыта для
краткости*/;
      (Symbol Identifier e.Name) e.Rest = /*реализация скрыта
для краткости*/;
```

```
      e._ = <MakeColdSolution t.IntrinsicFunction e.Args>
    };
```

Листинг 30 Оптимизация Type

Оптимизация First.

Сначала проверяем второй аргумент: в нём не должно быть e-переменных. Извлекаем число из первого аргумента и вычисляем функцию First от этого числа и выражения в аргументе. Результат конвертируем в узел и вставляем в замену (см. листинг 31).

Оптимизации Last, Lenw аналогичны.

```
'First'
    = e.Args : {
      (Symbol Number s.N) e.Expr
      , <CheckArgs-Intrinsic-EVariable e.Expr> : True
      = <First s.N e.Expr> : (e.Prefix) e.Suffix
      = (
        ((Brackets e.Prefix) e.Suffix) (()()) ()
      );
      e._ = <MakeColdSolution t.IntrinsicFunction e.Args>
    };
```

Листинг 31 Оптимизация First

4 Тестирование

4.1 Тестирование корректности оптимизаций

Различные варианты вызовов функций были написаны, для того чтобы проверить работоспособность новой возможности в компиляторе. Тестирование оптимизации функций Mu, Add, Sub, Mul, Div, Divmod, Mod, Compare приведены на листинге 32.

```
$INTRINSIC __Meta_Mu;
$INTRINSIC Add, Div, Divmod, Mod, Mul, Sub, Compare;
$META Mu;


$ENTRY Go {
       /* empty */
         = <Mu &Rev 'abracadabra'> : 'arbadacarba'
         = <Mu Rev 'abracadabra'> : 'arbadacarba'
         = <Mu ('Rev') 'abracadabra'> : 'arbadacarba'
         = 'abra' : e.X
         = <Mu { e.A = e.X e.A e.X } 'cad'> : 'abracadabra'
         = <Mu &Add 1 2> : 3
         = <Mu '+' 1 2> : 3
         = <Mu "+" 1 2> : 3
         = <Add 1 2> : 3
         = <Add '-' 1 2> : 1
         = <Add 1 '-' 2> : '-' 1
         = <Add '-' 1 '-' 2> : '-' 3
```

```
= <Add (2 3) 1> : 2 4

= <Add (1) 2 3> : 2 4

= <ConstValue 1> : e.z

= <Add 0 <Mul 2 e.z>> : 4

= <Sub 8 5> : 3

= <Sub '-' 8 5> : '-' 13

= <Sub 8 '-' 5> : 13

= <Sub '-' 8 '-' 5> : '-' 3

= <Sub <Mul 2 e.z> 0> : 4

= <Mul 3 3> : 9

= <Mul '-' 3 3> : '-' 9

= <Mul 3 '-' 3> : '-' 9

= <Mul '-' 3 '-' 3> : 9

= <Mul 1 <Mul 2 e.z>> : 4

= <Mul <Mul 2 e.z> 1> : 4

= <Mul (1 1) 1 1> : 1 2 1

= <Div 12 4> : 3

= <Div '-' 12 4> : '-' 3

= <Div 12 '-' 4> : '-' 3

= <Div '-' 12 '-' 4> : 3

= <Div <Mul 2 e.z> 1> : 4

= <Div (1 2) 1 1> : 1

= <Divmod 12 5> : (2) 2

= <Divmod '-' 12 5> : ('-' 2) '-' 2

= <Divmod 12 '-' 5> : ('-' 2) 2
```

```
        = <Divmod '-' 12 '-' 5> : (2) '-' 2

        = <Divmod <Mul 2 e.z> 1> : (4) 0

        = <Divmod (1 2) 1 1> : (1) 1

        = <Mod 12 5> : 2

        = <Mod '-' 12 5> : '-' 2

        = <Mod 12 '-' 5> : 2

        = <Mod '-' 12 '-' 5> : '-' 2

        = <Mod (1 2) 1 1> : 1

        = <Compare 10 20> : '-'

        = <Compare '-' 10 20> : '-'

        = <Compare 10 '-' 20> : '+'

        = <Compare '-' 10 '-' 20> : '+'

        = <Compare 20 10> : '+'

        = <Compare 10 10> : '0'

        = <Compare (1 2) 2 3> : '-'

        = <Compare ('-' 1 2) 2 3> : '-'

        = /* empty */
}


ConstValue {
      1 = 2
}


Rev {
      t.First e.Middle t.Last = t.Last <Rev e.Middle> t.Fir
st;
```

```
        t.One = t.One;
        /* empty */ = /* empty */;
}

__Meta_Mu {
        Rev e.Arg s.Metatable = <Rev e.Arg>;
        ('Rev') e.Arg s.Metatable = <Rev e.Arg>;
        '+' e.Arg s.Metatable = <Add e.Arg>;
        "+" e.Arg s.Metatable = <Add e.Arg>;
        s.Ptr e.Arg s.Metatable = <s.Ptr e.Arg>;
}
     /* определения остальных функций скрыты для краткости */
```

Листинг 32 Тестирование арифметических функций и функции Mu

Рассмотрим как происходит оптимизация на примере строчки `= <Mu &Add 1 2> : 3` в листинге 32. Эта строчка после обессахаривания превращается в две функции, которые затем прогоняются (см. Листинг 33). После оптимизации прогонки и интринсиков вызов функции Go=6 удаляется, также удаляются все последующие вызовы вплоть до вызова связанного со строчкой `= <ConstValue 1> : e.z` (см. Листинг 34). Таким образом, функция Mu сначала встроилась, а затем в ней вызов __Meta_Mu был заменён вызов `<Add 1 2>`, т.к. он объявлен как интринсик, то он тоже был оптимизирован. А затем из-за того, что все эти функции прогоняются их использование, полностью удаляется. Подобным образом происходит оптимизация и для других функций.

```
Go=6 {
    3 = <Go=7 <Mu '+' 1 2>>;
```

```
}
$DRIVE Go=6;
Go=5 {
    'abracadabra' = <Go=6 <Mu &Add 1 2>>;
}
$DRIVE Go=5;
```

Листинг 33 Вызов Mu после обессахаривания

```
Go=5 {
    'abracadabra' = <Go=15 <ConstValue 1>>;
}
```

Листинг 34 Вызов Mu после оптимизаций

Тестирование оптимизации остальных встроенных функций приведено на листингах 35 и 36.

```
$INTRINSIC Chr, Ord, Upper, Lower;
$ENTRY Go {
        /* empty */
         = <ConstValue 4> : e.z
         = <Chr 10 115 97> : '\nsa'
         = <Chr e.z 115 97> : '\nsa'
         = <Chr <Numb '10'> 115 97> : '\nsa'
         = <ConstValue 3> : e.f
         = <Ord '\nsa'> : 10 115 97
         = <Ord '\ns' e.f> : 10 115 97
         = <ConstValue 1> : e.x
```

```
        = <Upper 'abcd'> : 'ABCD'

        = <Upper 'a' e.x 'cd'> : 'ABCD'

        = <Upper 'abcd'> : 'ABCD'

        = <ConstValue 2> : e.y

        = <Lower 'ABCD'> : 'abcd'

        = <Lower 'A' e.y 'CD'> : 'abcd'

        = <Lower 'ABCD'> : 'abcd'

        = /* empty */

}


ConstValue {

        1 = 'b';

        2 = 'B';

        3 = 'a';

        4 = 10

}
```

Листинг 35 Тестирование функций Chr, Ord, Lower, Upper

```
$INTRINSIC Numb, Symb;

$INTRINSIC Explode, Implode, Implode_Ext, Explode_Ext;

$INTRINSIC Type;

$INTRINSIC First, Last, Lenw;

$ENTRY Go {

      /* empty */

        = <Numb '101'> : 101

        = <Numb '-101'> : '-' 101
```

```
= <Symb 101> : '101'
= <Symb '-' 101> : '-101'

= <Implode 'abcd'> : abcd
= <Explode abcd> : 'abcd'
= <Implode_Ext 'abcd'> : abcd
= <Explode_Ext abcd> : 'abcd'

= <Type 'a' b 1> : 'Lla' b 1
= <Type 1 b 1> : 'N0' 1 b 1
= <Type a b 1> : 'Wi' a b 1

= <Lenw 'abc'> : 3 'abc'
= <ConstValue 1> : s.x
= <Lenw 'a' s.x 'c'> : 3 'abc'
= <ConstValue 1> : e.y
= <Lenw 'a' e.y 'c'> : 3 'abc'

= <First 2 a (1 2) b c> : (a (1 2)) b c
= <ConstValue 2> : s.x2
= <First 2 s.x2 (1 2) b c> : (a (1 2)) b c
= <ConstValue 2> : e.y2
= <First 2 e.y2 (1 2) b c> : (a (1 2)) b c

= <Last 1 a (1 2) b c> : (a (1 2) b) c
```

```
        = <ConstValue 2> : s.x2

        = <Last 1 s.x2 (1 2) b c> : (a (1 2) b) c

        = <ConstValue 2> : e.y2

        = <Last 1 e.y2 (1 2) b c> : (a (1 2) b) c

        = /* empty */
}


ConstValue {
        1 = 'b';
        2 = a
}
```

Листинг 36 Тестирование функций Numb, Symb, Implode, Implode_Ext, Explode, Explode_Ext, First, Last, Lenw, Type

4.2 Оценка времени работы компилятора после внедрения оптимизаций

Оценка реализованных оптимизаций была проведена на исходном коде компилятора Рефала.

Тестирование проводилось на компьютере с характеристиками:

- Объём оперативной памяти: 15.9 ГБ
- Модель процессора: Intel Core i7
- Частота процессора: 3.1 ГГц
- Количество ядер процессора: 8
- Операционная система: Linux Ubuntu 18.04 bionic
- Разрядность системы: 64
- Компилятор C++: GNU g++ версии 7.5.0

Во всех замерах проводится 13 компиляций исходников Рефала без флагов и с флагами: -ODS(оптимизация прогонки и специализации), –OiDS(оптимизация интринсиков, прогонки и специализации). Время компиляции при каждом проходе разное, поэтому по всем замерам вычисляется медиана и доверительный интервал из границ первого и четвёртого квантилей. Также в каждом случае было вычислено количество шагов рефал-машины.

Сначала был произведена оценка на исходниках, в которых не было объявлено интринсиков и в функции Apply не было произведено замены вызова входной функции на вызов функции Mu. Результаты в таблице 1.

Таблица 1 Результаты тестирования оптимизации на компиляторе Рефала

Флаги	Количество шагов	Время компиляции, медиана, с	Время компиляции, I квартиль, с	Время компиляции, IV квартиль, с
Нет флагов	25817261	38.14	38.03	39.62
-ODS	23506400	35.88	35.77	36.09
-OiDS	23506400	35.95	35.67	36.15

Затем предыдущие исходники были модифицированы. Был добавлен вызов Mu в функции Apply. Результаты приведены в таблице 2. Из первых двух замеров видно, что при флаге –ODS(-OiDS) компилятор медианное время компиляции уменьшилось на 4.3% и увеличилось количество рефал-шагов на 13%. Это происходит из-за того, что функция Mu мешает оптимизации прогонки.

Таблица 2 Результаты тестирования оптимизации на компиляторе Рефала

Флаги	Количество шагов	Время компиляции, медиана, с	Время компиляции, I квартиль, с	Время компиляции, IV квартиль, с
Нет флагов	35616565	44.44	44.16	44.96
-ODS	27082390	37.90	37.38	38.03
-OiDS	27082390	37.48	37.34	37.92

В третьем замере в компилятор были включены все интринсики, реализованные в рамках данной работы. Результаты в таблице 3. Из этих результатов видно, что оптимизация встроенных функций в компиляторе уменьшает количество шагов на 8.7% и медианное время компиляции на 4.8% по сравнению с первым замером, если при сборке используются флаги –OiDS.

Таблица 3 Результаты тестирования оптимизации на компиляторе Рефала

Флаги	Количество шагов	Время компиляции, медиана, с	Время компиляции, I квартиль, с	Время компиляции, IV квартиль, с
Нет флагов	25913165	38.36	38.32	38.75
-ODS	22184258	35.54	34.43	34.73
-OiDS	21471802	34.21	33.82	34.43

В четвёртом замере была произведена замена на Mu аналогично второму замеру и функция __Meta_Mu была объявлена интринсиком. Результат приведён в таблице 4. Из замера видно, что при флаге –OiDS медианное время компиляции

уменьшилось на 3.6% и количество шагов уменьшилось на 8.2% по сравнению со вторым замером, но по сравнению с первым замером медианное время компиляции увеличилось на 0.5% и количество шагов увеличилось на 6.4%.

Таблица 4 Результаты тестирования оптимизации на компиляторе Рефала

Флаги	Количество шагов	Время компиляции, медиана, с	Время компиляции, I квартиль, с	Время компиляции, IV квартиль, с
Нет флагов	35626333	44.15	43.94	44.72
-ODS	25605537	36.37	36.32	36.71
-OiDS	25012700	36.12	35.77	36.27

В пятом замере была произведена замена на Mu аналогично второму замеру и были оставлены объявления интринсиков. Результаты в таблице 5. Из этих результатов видно, что медианное время компиляции при флаге -OiDS по сравнению с первым замером уменьшилось на 1.1% и число шагов увеличилось на 3.8%.

Таблица 5 Результаты тестирования оптимизации на компиляторе Рефала

Флаги	Количество шагов	Время компиляции, медиана, с	Время компиляции, I квартиль, с	Время компиляции, IV квартиль, с
Нет флагов	35739082	44.38	44.29	44.97
-ODS	25696622	36.89	36.57	37.09
-OiDS	24391125	35.56	35.51	35.66

5 Руководство пользователя

5.1 Установка и раскрутка компилятора Рефала-5λ

Для установки компилятора нужно скачать и установить систему управления версиями git и любой компилятор языка C++ (например, g++ из gcc [10]). Для установки компилятора необходимо скачать исходники, которые хранятся в репозитории Рефала-5λ на github [11], следующей командой: `git clone https://github.com/bmstu-iu9/refal-5-lambda.git`

Для раскрутки самоприменимого компилятора нужно выполнить команду из корня репозитория:

```
./bootstrap.sh --no-tests
```

На Unix-like системах

```
bootstrap.bat --no-tests
```

На Windows.

После этой команды нужно проверить файл c-plus-plus.conf.bat (sh), выставить в нём правильный компилятор C++. Если файл был изменён, нужно заново запустить сборку.

Затем нужно добавить папку bin с исполняемыми файлами компилятора к переменной окружения `PATH`.

5.2 Использование интринсиков

Для того, чтобы пометить функцию, как интринсик используется директива `$INTRINSIC`, через неё перечисляются имена функций. Оптимизация сработает, если указать компилятору флаг –Oi. Если нужно оптимизировать функцию Mu, нужно указать флаг –OiD либо –OiI, потому что функция Mu во время компиляции помечается, как встраиваемая функция. Функции, которые можно помечать, как интринсики: __Meta_Mu, Add, Sub, Mul, Div, Divmod, Mod,

Compare, Numb, Symb, Implode, Implode_Ext, Explode, Explode_Ext, Chr, Ord, Upper, Lower, First, Last, Lenw, Type.

ЗАКЛЮЧЕНИЕ

В процессе выполнения этой работы были выполнены все поставленные задачи. Был изучен язык Рефал-5λ, устройство его компилятора и его библиотека.

Компилятор языка был модифицирован: реализована поддержка синтаксиса для оптимизации встроенных функций, произведено тестирование оптимизации.

Оптимизации были применены внутри компилятора, что дало прирост производительности компилятора в 4.8%. Оптимизация функции Mu определении Apply не смогла полностью убрать эффект замедления компиляции от внедрения этой функции в Apply, но заметно снизила этот эффект. Компилятор с оптимизациями внутренних функций и с заменой Mu в Apply ускорился на 1.1%.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. С.А.Романенко. Прогонка для программ на РЕФАЛе-4. Препринт №211 — Институт Прикладной Математики АН СССР, 1987.
2. Турчин Валентин Фёдорович Эквивалентные преобразования рекурсивных функций, описанных на языке РЕФАЛ [Конференция] // Труды симпозиума "Теория языков и методы построения систем программирования". - Киев-Алушта : [б.н.], 1972. - стр. 31-42.
3. Microsoft C++ Compiler Intrinsics [Электронный ресурс] – Режим доступа: https://docs.microsoft.com/en-us/cpp/intrinsics/compiler-intrinsics?view=vs-2019. Дата обращения: 30.05.2020
4. Microsoft C++ Intrinsics available on all architectures [Электронный ресурс] – Режим доступа: https://docs.microsoft.com/en-us/cpp/intrinsics/intrinsics-available-on-all-architectures?view=vs-2019. Дата обращения: 30.05.2020
5. А. П. Немытых Лекции по языку программирования Рефал. Сборник трудов по функциональному языку программирования Рефал, том I // Издательство «СБОРНИК». - 2014. - С. 120.
6. Компилятор Рефала-5λ [Электронный ресурс] – https://github.com/bmstu-iu9/refal-5-lambda. Дата обращения: 30.05.2020.
7. V. Turchin REFAL-5 programming guide & reference manual [Электронный ресурс] – Режим доступа: http://refal.botik.ru/book/html. Дата обращения: 30.05.2020.
8. А. Ахо, М. Лам, Р. Сети, Д. Ульман. Компиляторы: принципы, технологии и инструментарий — 2 изд. — М.: Вильямс, 2008
9. Набор библиотек для Рефала-5 [Электронный ресурс] – Режим доступа: https://mazdaywik.github.io/refal-5-framework/LibraryEx.html. Дата обращения: 30.05.2020
10. GCC, the GNU Compiler Collection [Электронный ресурс] – Режим доступа: https://gcc.gnu.org. Дата обращения: 30.05.2020.
11. Github [Электронный ресурс] – Режим доступа: https://github.com. Дата обращения: 30.05.2020.

ПРИЛОЖЕНИЕ А