|  |  |
| --- | --- |
| https://studfiles.net/html/2706/219/html_4NnFGVyFmL.LWVf/img-KuTuVC.png | МИНОБРНАУКИ РОССИИ  федеральное государственное бюджетное образовательное учреждение  высшего образования  **«Балтийский государственный технический университет «ВОЕНМЕХ» им. Д.Ф. Устинова»**  **(БГТУ «ВОЕНМЕХ» им. Д.Ф. Устинова»)** |
| БГТУ.СМК-Ф-4.2-К5-01 |

|  |  |  |  |
| --- | --- | --- | --- |
| Факультет | И |  | Информационные и управляющие системы |
|  | шифр |  | наименование |
| Кафедра | И5 |  | Информационные системы и программная инженерия |
|  | шифр |  | наименование |
| Дисциплина | Системное ПО | | |

КУРСОВАЯ РАБОТА

на тему

|  |
| --- |
| Создание компилятора |
| для подмножества языка Go на ASM i386 |
|  |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Выполнил студент группы | | | | | | И582 |
| Васильев М.С. | | | | | | |
| Фамилия И.О. | | | | | | |
|  | | | | | | |
| **РУКОВОДИТЕЛЬ** | | | | | | |
|  | | | | | | |
| Иванов К.С. | | | |  |  | |
| Фамилия И.О. | | | |  | Подпись | |
|  | | | | | | |
| Оценка | | |  | | |  |
| « |  | » |  | | | 2020 г. |

Санкт-Петербург  
2020

**Балтийский государственный технический университет «ВОЕНМЕХ» им. Д.Ф. Устинова»  
(БГТУ «ВОЕНМЕХ» им. Д.Ф. Устинова»)**

|  |  |  |  |
| --- | --- | --- | --- |
| Факультет | И |  | Информационные и управляющие системы |
|  | шифр |  | наименование |
| Кафедра | И5 |  | Информационные системы и программная инженерия |
|  | шифр |  | наименование |
| Дисциплина | Системное ПО | | |

**ЗАДАНИЕ**

**НА КУРСОВУЮ РАБОТУ**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Студенту | | Васильеву Максиму Сергеевичу | | | | | | |
|  | | (фамилия, имя, отчество) | | | | | | |
| Научный руководитель | | |  | | | | | |
|  | | | (ученая степень, звание, Ф.И.О) | | | | | |
| Тема: | Создание компилятора для подмножества языка Go на | | | | | | | |
| ASM i386 | | | | | | | | |
| Основный вопросы, подлежащие разработке: | | | |  | | | | |
| 1. Лексический анализ, алфавит, символы | | | | | | | | |
| 1. Синтаксический анализ, грамматика, дерево разбора | | | | | | | | |
| 1. Семантический анализ, проверка переменных | | | | | | | | |
| 1. Генерация кода, создание программ на языке ассемблера | | | | | | | | |
| Задание выдал: Иванов Константин Сергеевич | | | | |  |  |  |  | |
|  | | | | |  | дата |  | подпись | |
| Задание принял: Иванов Константин Сергеевич | | | | |  |  |  |  | |
|  | | | | |  | дата |  | подпись | |

Санкт-Петербург  
2020

**СОДЕРЖАНИЕ**

[ВВЕДЕНИЕ 4](#_Toc42888355)

[1 Лексический анализ 5](#_Toc42888356)

[1.1 Описание 7](#_Toc42888357)

[1.2 Результат работы 10](#_Toc42888358)

[2 Синтаксический анализ 11](#_Toc42888359)

[2.1 Описание 11](#_Toc42888360)

[2.2 Реализация 12](#_Toc42888361)

[2.3 Результат работы 13](#_Toc42888362)

[3 Семантический анализ 14](#_Toc42888363)

[3.1 Описание 14](#_Toc42888364)

[3.2 Реализация 14](#_Toc42888365)

[3.3 Результат работы 15](#_Toc42888366)

[4 Генерация кода 16](#_Toc42888367)

[4.1 Описание 16](#_Toc42888368)

[4.2 Реализация 16](#_Toc42888369)

[4.3 Результат работы 18](#_Toc42888370)

[5 Тестовый пример 19](#_Toc42888371)

[ЗАКЛЮЧЕНИЕ 23](#_Toc42888372)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 24](#_Toc42888373)

[ПРИЛОЖЕНИЕ 25](#_Toc42888374)

# ВВЕДЕНИЕ

Целью работы является написать компилятор с заданного подмножества языка Go с незначительными модификациями и упрощениями.

Компилятор построен из следующих составных частей:

1. Лексический анализатор;
2. Синтаксический анализатор;
3. Семантический анализатор;
4. Генератор результирующего кода.

Компилятор должен запускаться командной строкой с двумя входными параметрами. Первый входной параметр имя входного файла, второй параметр − быть имя результирующего файла.

Входной язык компилятора должен удовлетворяет следующим требованиям:

* Входная программа разбита на строки произвольным образом, все пробелы и символы табуляции должны игнорироваться компилятором, то есть пропускаться на этапе лексического анализа.
* Текст входной программы может содержать комментарии любого типа и любой длины, которые должны игнорироваться компилятором.
* Входная программа должна представлять собой единый модуль, содержащий линейную последовательность операторов, вызовы функций также предусматриваются.
* Должны быть предусмотрены следующие варианты операторов входной программы:

1. Оператор присваивания;
2. Условный оператор;
3. Оператор цикла for.

* Выражения в операторах содержат следующие операции:

1. Арифметические операции сложение, вычитание, умножение и деление;
2. Операции сравнения – меньше, больше, равно, не равно, меньше или равно, больше или равно;
3. Логические операции – и, или, не;

* Операндами в выражениях могут выступать идентификаторы (переменные) и константы (двоичные).
* Не допускается переприсвоение значений константам.
* В условном операторе в случае истиного (ложного) условии должен присутствовать составной оператор.

В операторе цикла, в качестве тела цикла должен присутствовать составной оператор.

Алфавит языка:

1. Прописные буквы латинского алфавита;
2. Строчные буквы латинского алфавита;
3. Арабские цифры;
4. Разделители: , . ; : ? ! ' " | / \ ~ \_ ^ ( ) { } [ ] < > # % & - = + \*

В идентификаторах могут использоваться латинские буквы, цифры и знак нижнего подчеркивания ( \_ ). В C++ различаются строчные и прописные буквы (т. е. имена регистро-зависимы), так: Name, name и NAME — это разные идентификаторы. Правила использования идентификаторов:

1. Можно использовать латинские символы: A..Z, a..z;
2. Можно использовать арабские цифры: 0..9 и символ нижнего подчеркивания, но не в начале;
3. Пробелы в имени не допускаются;
4. Не допускается использования ключевых слов.

Особенности выходного языка:

Выходным языком выбран ассемблер для процессоров семейства x86 – MASM. Первоначально был произведён компанией Microsoft для написания программ в операционной системе MS-DOS и был в течение некоторого времени самым популярным ассемблером, доступным для неё. MASM поддерживал широкое разнообразие макросредств и структурированность программных идиом, включая конструкции высокого уровня для повторов, вызовов процедур и чередований (поэтому MASM — ассемблер высокого уровня). Позднее была добавлена возможность написания программ для Windows. MASM — один из немногих инструментов разработки Microsoft, для которых не было отдельных 16- и 32-битных версий.

Есть много развивающихся проектов для разработки программного обеспечения, которые поддерживают MASM, включая IDE (например RadASM), отладчики (вроде OllyDbg), и дизассемблеры (включая IDAPro, интерактивный дизассемблер).

Разработка ПО, является неотъемлемой частью программиста. Но в отличие от высокоуровневых языков программирования, Ассемблер учит глубоко понимать работу компьютера, оптимизировать работку с аппаратными ресурсами, а также программировать любую технику, тем самым развиваясь, например в таком направлении, как машинное обучение. Именно поэтому MASM можно использовать в качестве выходного языка.

# Лексический анализ

## Описание

Задача лексического анализа заключается в разборе входной программы. Лексический анализатор разбивает входную программу на токены, у которого есть тип и название, которым служит само ключевое слово. Тип токена фактически описывает, что означает токен, так, например, токен ">" описывает оператор сравнения. Тип токена понадобится при синтаксическом анализе, чтобы не проверять его каждый раз, например, проверяя является ли строка "123" числом.

На вход лексического анализатора подается файл, далее функция GetAllTextInFile получает текст из файла. Далее, программа передает в конструктор объекта класса TokenBraking текст из файла. После чего, TokenBraking вызывает метод SplitIntoTokens, который последовательно анализирует текст и создает токены. То есть создает новые объекты класса Token и помещает в поле списка \_tokens класса TokenBraking.

Лексический и синтаксический анализатор взаимодействуют последовательно, т.е. сначала работает лексический анализатор, если на этапе лексического анализа не обнаружено ошибок, то начинает работу синтаксический анализатор.

Методы лексического анализатора класса TokenBraking:

* void SplitIntoTokens() − Разбивает текст на токены.
* bool IsCompositesToken(char sym1, char sym2) − Проверка на то, является ли токен составным. Принимает 2 последовательных символа.
* bool IsCompositesSeparator(char sym1, char sym2) − проверка символа на то, находится ли рядом 2 разделителя. Принимает 2 последовательных символа.
* vector<Token\*>\* GetTokens() − возвращение списка токенов.
* void ShowTokens () − вывод токенов на экран.
* bool IsSymNumber (char sym) − проверка на то, является ли символ цифрой или нет.
* bool IsSymLit (char sym) −проверка на то, является ли символ маленькой или большой буковой латинского или русского алфавита или нет.

Алгоритм разбиения на токены:

1. Начало
2. Последовательно начинаем смотреть символы с текущей позиции. Если это конец текста – переходим к пункту 7.
3. Если это не строковой литерал
   1. Если это разделитель, то все что находится в буфере символов – это наш токен, создаем его. Очищаем буфер.
4. Если это знак (т.е. один из следующих символов: ,()[]{}:=+-\*%/!;<>&|\n)
   1. Если это заново разделитель, то возвращаемся к пункту 2.
   2. Если это символ “-” и крайний токен не литерал и не число, то добавляем символ в буфер и переходим к пункту 2.
   3. Если в связке со следующим символом это составной токен, то добавляем этот токен в список и принудительно переходим через один символ. Возвращаемся к пункту 2.
   4. Создаем токен с тем, что сейчас находится в буфере. Очищаем буфер. Переходим к пункту 2.
5. Добавляем символ в буфер.
6. Переходим к пункту 2.
7. Конец

начало

конец

да

нет

Строковой литерал

Знак ,()[]{}:=+-\*%/!;<>&|\n

Предыдущий токен разделитель

Символ “-” и крайний токен не буква и не число

Составной токен со следующим символом

Создаем токен, очищаем буфер

Добавляем символ в буфер

Разделитель и буфер не пустой

пока не конец текста

нет

да

нет

нет

нет

да

да

нет

да

да

## Результат работы

Результатом работы лексического анализатора является набор токенов.

# Синтаксический анализ

## Описание

Задача синтаксического анализатора — провести разбор текста программы, сопоставив его с эталоном, данным в описании языка [1]. Структура конструкций синтаксического анализатора более сложна, чем структура идентификаторов и чисел. Поэтому для описания синтаксиса языка нужны более мощные грамматики нежели регулярные. Для этих целей были использованы контекстно-свободные грамматики.

Для синтаксического анализа, в данной работе, используется метод рекурсивного спуска. Суть метода, заключается в создании ряда функций с именами, как у продукций грамматики. Процесс разбора начинается с главной продукции, и в процессе её разбора, происходят переходы между правилами, а значит и функциями обработчиками, каждая из которых разбирает свою часть правила.

Задача синтаксического анализа состоит в том, чтобы определить имеет ли цепочка лексем конструкцию, заданную синтаксисом языка. Этот этап является основным для поиска ошибок в написанной программе.

Результатом синтаксического анализа является AbstractSyntaxTree, класс, где содержится абстрактное синтаксическое дерево — это дерево, которое в абстрактном виде представляет структуру программы. AST содержит полную синтаксическую модель программы без лишних деталей (таких, как пробельные символы или комментарии).

AST состоит из узлов, каждый узел задается с помощью класса Node. Каждый узел имеет значение, тип, а также 4 потомка на другие Node. Имеются указатели на объект класса Function и указатель на объект класса Variable. Нужны для более удобной работы с Node в ASM. Также имеется поле типа int Id. Необходимо, например, для правильной нумерации строковых переменных в блоке данных ASM.

## Реализация

Для реализации синтаксического анализа был создан класс Parser. Так как анализатор работает с помощью метода рекурсивного спуска, для каждой продукции грамматики была создана отдельная функция в классе с именем, совпадающим с именем продукции.

Ниже описаны все функции для нетерминалов грамматики, для каждой функции написано, за что она отвечает:

1. Node\* Parse(); — начало работы с деревом, создается корневой узел “PROG”;
2. Node\* Statemets() — обрабатываются основные блоки Statement, создается их последовательная цепочка, через указатели друг на друга. Добавляем блоки, которые могут быть вне функций;
3. Node\* Statement(); — основной метод по обработке конструкций в языке. Основные: переменные, условия, циклы, выражения;
4. Node\* GetSignatureFunc(); — получаем сигнатуру функции;
5. Node\* GetListParameters(); — получаем список параметров в функции;
6. Node\* GetResultFunc(); — получаем тип возвращаемого значения функции;
7. void InitializationArray(); — инициализация массива;
8. Node\* ParentExprSBra(); — обработка выражений с квадратными скобками (то есть с массивами).
9. Node\* ParentExpr (); — обработка выражений с круглыми скобками
10. Node\* Expr(); — обработка основных операций
11. Node\* LogOr(); — выражения с логическим или
12. Node\* LogAnd(); — выражения с логическим и
13. Node\* LogUnarOr(); — выражения с унарным логическим или
14. Node\* LogUnarAnd(); — выражения с унарным логическим и
15. Node\* Compare(); — выражения с различными сравнениями
16. Node\* Add(); — выражения со сложением и вычитанием
17. Node\* Mul(); — выражения с умножением и делением
18. Node\* Unar(); — операции инкремента и декремента. Постфиксного и префиксного;
19. Node\*Inversion(); — выражения с инверсией
20. Node\* GetNodeValue(); — получаем член выражения (константа, массив, переменная, функция)
21. Node\* GetListParametersAccess(); — получаем список параметров, которые передали в функцию
22. Node\* GetNodeHead(); — Получить первый узел дерева.

Так как у языка Go есть особенности с возможностями инициализации и присваивания переменным значений, логика по добавлению новых переменных вынесена в другой класс – QueueVariableNode;

## Результат работы

Результатом работы синтаксического анализатора является абстрактное синтаксическое дерево, в котором в абстрактном виде описаны все конструкции кода. Также обработаны некоторые ошибки, связанные в основном с опечатками программиста языка.

# Семантический анализ

## Описание

Семантический анализ заключается в проверке правильности типов данных, используемых в программе. Кроме того, на этом этапе компилятор должен также проверить, соблюдаются ли определенные контекстные условия входного языка. В современных языках программирования одним из примеров контекстных условий может служить обязательность описания переменных: для каждого использующего вхождения идентификатора должно существовать единственное определяющее вхождение.

## Реализация

Для реализации семантического анализа был выбран класс RecursiveTraversal.

Так как задача семантического анализа, в данной работе, заключается в выявлении ошибок связанных с необъявленными или переобъявленными переменными, и другими подобными ошибками, то логично обработать все переменные и функции, которые есть в дереве заранее, до генерации кода.

Для хранения переменных был создан класс Variable, который представляет простое название переменной, так как поддерживается только тип integer. Для хранения всех переменных был создан класс TableVariable, который предоставляет удобный интерфейс, так, например, функция IsVariable по имени возвращает истину, если в таблице уже есть такая переменная.

Помимо этого, был создан класс Function, который содержит таблицу TableVariable для локальных переменных и другую для аргументов. Для хранения всех Function был создан класс TableFunction по подобию TableVariable.

Также есть поддержка глобальных переменных, но реализованная не до конца.

В конструкторе заполняется таблица функций и в них таблицы локальных переменных и аргументов, а в методе Traversal проверяются вышеописанные ошибки.

## Результат работы

Результатом работы семантического анализатора является таблица функций, внутри которых есть таблицы переменных. А также, в случае ошибок, отчет о ошибке.

# Генерация кода

## Описание

Генерация кода является последней стадией процесса компиляции. Для данной курсовой работы конечным языком является ассемблер, что усложняет работу и требует знаний этого языка. Следует отметить, что в ассемблере используется стек. На нем удобно считать как арифметические операции, так и операции сравнения.

Ассемблер предоставляет возможности для трансляции регулярных языков, но, чтобы использовать контекстно-свободный язык, надо предварительно его видоизменить, что, собственно, и сделано при обходе дерева разбора.

Ассемблирование может быть не первым и не последним этапом на пути получения исполнимого модуля программы. Так, многие компиляторы с языков программирования высокого уровня выдают результат в виде программы на языке ассемблера, которую в дальнейшем обрабатывает ассемблер.

В данной работе использовался синтаксис ассемблера MASM, так как разработка велась на Windows.

Помимо стека для расчетов в ассемблере используются регистры. Регистр — это сверхбыстрая память, которая расположена в процессоре.

## Реализация

За генерацию кода в данной работе отвечает класс Asm. Основными полями являются следующие поля:

1. string \_writing — текст выходного файла;
2. string \_dataWriting — сигмент данных;
3. TableFunction\* \_functions – указатель на таблицу функций;
4. TableImport\* \_imports – указатель на таблицу импортов;

Для более удобной работы с ассемблером были созданы следующие поля:

1. int \_byteArguments — число байт, которое надо сместить, чтобы перейти к определенному аргументу;
2. bool \_isFirstFunc — первая ли функция;
3. bool \_isMainFuncTraversal — главная ли функция;
4. map<string, string> \_stringsConst — соответствия между литералой и переменной в сигменте данных;
5. Function\* \_functionLast — предыдущая функция;

Для генерации созданы две основные функции:

1. Node\* RecursiveTraversal(Node\* currentNode) — функция для генерации основных блоков;
2. void ExprAsm(Node\* currentNode) — функция для генерации различных выражений.

Для генерации ассемблерного кода были созданы функции с именами, как у команд ассемблера, а также с количеством аргументов, как у команды.

Далее приведены все реализованные команды:

1. void Push(const string& value)
2. void Pop(const string& value)
3. void Add(const string& value1, const string& value2)
4. void Sub(const string& value1, const string& value2)
5. void Imul(const string& value1, const string& value2)
6. void Mov(const string& value1, const string& value2)
7. void Cmp(const string& value1, const string& value2)
8. void Jmp(const string& value)
9. void Je(const string& value)
10. void Jne(const string& value)
11. void Jl(const string& value)
12. void Jle(const string& value)
13. void Jg(const string& value)
14. void Jge(const string& value)
15. void Ret(const string& value = "")
16. void Finit()
17. void Fild(const string& value)
18. void Fdiv(const string& value1, const string& value2)
19. void Fist(const string& value)

## Результат работы

Результатом работы генератора кода, является корректный код на языке ассемблера, который компилируется в верную по функционалу программу.

# Тестовый пример

Тестовая программа на Go:

package main

import(

"fmt"

"math"

)

func main(){

var a,b,c int

fmt.Println("a = ")

fmt.Scan(a)

fmt.Println("b = ")

fmt.Scan(b)

fmt.Println("c = ")

fmt.Scan(c)

SqrEq(a,b,c)

}

func SqrEq(a,b,c int) {

fmt.Println(a, "x^2", " + ", b, "x", " + ", c, " = 0; \n")

var d int = b\*b - 4\*a\*c

if d < 0 {

fmt.Println("The discriminant < 0 !\n")

} else if d == 0 {

math.Sqrt(d)

var x int = -b / (2\*a)

fmt.Println("x = ")

fmt.Println(x)

} else {

math.Sqrt(d)

var x1 int= (-b + d) / (2\*a)

var x2 int= (-b - d) / (2\*a)

fmt.Println("x1 = ")

fmt.Println(x1)

fmt.Println("\n")

fmt.Println("x2 = ")

fmt.Println(x2)

}

}

}

}

Выходной код на ассемблере:

.586

.model flat, stdcall

include <\masm32\include\msvcrt.inc>

include <\masm32\include\kernel32.inc>

includelib <\masm32\lib\msvcrt.lib>

includelib <\masm32\lib\kernel32.lib>

data segment

printd\_format db "%d", 0

prints\_format db "%s", 0

input\_format db "%d", 0

input\_result dd 0

sqrt\_result dd 0

div\_operand\_1 dd 0

div\_operand\_2 dd 0

string\_const\_0 db "a = ", 0

string\_const\_1 db " ", 13,10,0

string\_const\_2 db "b = ", 0

string\_const\_3 db " ", 13,10,0

string\_const\_4 db "c = ", 0

string\_const\_5 db " ", 13,10,0

string\_const\_6 db "x^2", 0

string\_const\_7 db " + ", 0

string\_const\_8 db "x", 0

string\_const\_9 db " + ", 0

string\_const\_10 db " = 0; ", 13,10,0

string\_const\_11 db "The discriminant < 0 ! ", 13,10,0

string\_const\_12 db "x = ", 0

string\_const\_13 db "x1 = ", 0

string\_const\_14 db " ", 13,10,0

string\_const\_15 db "x2 = ", 0

data ends

text segment

a0 = -4

b0 = -8

c0 = -12

d0 = -4

x0 = -8

x10 = -12

x20 = -16

c1 = 8

b1 = 12

a1 = 16

printD PROC

enter 0, 0

mov eax, [ebp + 8]

invoke crt\_printf, offset printd\_format, eax

leave

ret 4

printD ENDP

printS PROC

enter 0, 0

push eax

push ebx

push ecx

push edx

mov eax, [ebp + 8]

invoke crt\_printf, offset prints\_format, eax

pop eax

pop ebx

pop ecx

pop edx

leave

ret 4

printS ENDP

sqrt PROC

enter 0, 0

push ebx

push ecx

push edx

mov eax, [ebp + 8]

mov sqrt\_result, eax

finit

fild sqrt\_result

fsqrt

fist sqrt\_result

mov eax, sqrt\_result

pop ebx

pop ecx

pop edx

leave

ret 4

sqrt ENDP

input PROC

enter 16, 0

push ebx

push ecx

push edx

invoke crt\_scanf, offset input\_format, offset input\_result

mov eax, input\_result

pop ebx

pop ecx

pop edx

leave

ret

input ENDP

SqrEq PROC

enter 16, 0

mov eax, a1[ebp]

push eax

call printD

push offset string\_const\_6

call printS

push offset string\_const\_9

call printS

mov eax, b1[ebp]

push eax

call printD

push offset string\_const\_8

call printS

push offset string\_const\_9

call printS

mov eax, c1[ebp]

push eax

call printD

push offset string\_const\_10

call printS

push b1[ebp]

push b1[ebp]

pop eax

pop ebx

imul eax, ebx

push eax

push 4

push a1[ebp]

pop eax

pop ebx

imul eax, ebx

push eax

push c1[ebp]

pop eax

pop ebx

imul eax, ebx

push eax

pop ebx

pop eax

sub eax, ebx

push eax

pop d0[ebp]

push d0[ebp]

pop ecx

push 0

pop edx

cmp ecx, edx

jge \_compare\_not\_equal489113596909200

push 1

jmp \_compare\_end489113596909200

\_compare\_not\_equal489113596909200:

push 0

\_compare\_end489113596909200:

pop eax

cmp eax, 0

je \_\_if\_else\_489113596894600

\_\_if\_start\_489113596894600:

push offset string\_const\_11

call printS

jmp \_\_if\_end\_489113596894600

\_\_if\_else\_489113596894600:

push d0[ebp]

pop ecx

push 0

pop edx

cmp ecx, edx

jne \_compare\_not\_equal489113597158600

push 1

jmp \_compare\_end489113597158600

\_compare\_not\_equal489113597158600:

push 0

\_compare\_end489113597158600:

pop eax

cmp eax, 0

je \_\_if\_else\_489113597137700

\_\_if\_start\_489113597137700:

mov eax, d0[ebp]

push eax

call sqrt

mov d0[ebp], eax

push 0

push b1[ebp]

push 2

push a1[ebp]

pop eax

pop ebx

imul eax, ebx

push eax

pop ebx

pop eax

mov div\_operand\_1, eax

mov div\_operand\_2, ebx

finit

fild div\_operand\_2

fild div\_operand\_1

fdiv st(0), st(1)

fist div\_operand\_1

push div\_operand\_1

pop ebx

pop eax

sub eax, ebx

push eax

pop x0[ebp]

push offset string\_const\_12

call printS

mov eax, x0[ebp]

push eax

call printD

jmp \_\_if\_end\_489113597137700

\_\_if\_else\_489113597137700:

\_\_if\_start\_489113597894800:

mov eax, d0[ebp]

push eax

call sqrt

mov d0[ebp], eax

push 0

push b1[ebp]

pop ebx

pop eax

sub eax, ebx

push eax

push d0[ebp]

pop eax

pop ebx

add eax, ebx

push eax

push 2

push a1[ebp]

pop eax

pop ebx

imul eax, ebx

push eax

pop ebx

pop eax

mov div\_operand\_1, eax

mov div\_operand\_2, ebx

finit

fild div\_operand\_2

fild div\_operand\_1

fdiv st(0), st(1)

fist div\_operand\_1

push div\_operand\_1

pop x10[ebp]

push 0

push b1[ebp]

pop ebx

pop eax

sub eax, ebx

push eax

push d0[ebp]

pop ebx

pop eax

sub eax, ebx

push eax

push 2

push a1[ebp]

pop eax

pop ebx

imul eax, ebx

push eax

pop ebx

pop eax

mov div\_operand\_1, eax

mov div\_operand\_2, ebx

finit

fild div\_operand\_2

fild div\_operand\_1

fdiv st(0), st(1)

fist div\_operand\_1

push div\_operand\_1

pop x20[ebp]

push offset string\_const\_13

call printS

mov eax, x10[ebp]

push eax

call printD

push offset string\_const\_14

call printS

push offset string\_const\_15

call printS

mov eax, x20[ebp]

push eax

call printD

jmp \_\_if\_end\_489113597894800

\_\_if\_else\_489113597894800:

\_\_if\_end\_489113597894800:

\_\_if\_end\_489113597137700:

\_\_if\_end\_489113596894600:

leave

ret 12

SqrEq ENDP

\_\_main:

enter 12, 0

push offset string\_const\_0

call printS

call input

mov a0[ebp], eax

push offset string\_const\_14

call printS

push offset string\_const\_2

call printS

call input

mov b0[ebp], eax

push offset string\_const\_14

call printS

push offset string\_const\_4

call printS

call input

mov c0[ebp], eax

push offset string\_const\_14

call printS

push a0[ebp]

push b0[ebp]

push c0[ebp]

call SqrEq

leave

ret

text ends

end \_\_main

# ЗАКЛЮЧЕНИЕ

В результате выполнения данной курсовой работы были получены знания в области разработки компиляторов. Получен опыт в работе с низкоуровневым языком ассемблера. Получены знания по разработке каждого из этапов компилятора.

Разработка происходила на языке программирования C++ в среде Visual Studio 2020 с использованием компилятора MVSC. Был реализован компилятор для языка высокого уровня – Go на язык низкого уровня – ассемблер, включающий в себя этапы лексического, синтаксического, семантического анализов, а также генератор кода. Программа была протестирована на различных примерах.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Е. Н. Ишакова. Разработка компиляторов: Методические указания к курсовой работе. — Ориенбург: ГОУ ОГУ, 2005
2. А. Е. Пентус, М. Р. Пентус. Теория формальных языков: Учебное пособие — М.:ЦПИ, 2004
3. А. Ю. Молчанов. Системное программное обеспечение. — СПб.: Питер, 2003
4. Альфред В. Ахо, Моника С. Лам, Рави Сети, Джеффри Д. Ульман. Компиляторы: принципы, технологии и инструментарий — 2 изд. — М.: Вильямс, 2008
5. Вирт Н. Построение компиляторов. — М.: ДМК Пресс, 2010

# **ПРИЛОЖЕНИЕ**

Текст программы на прилагаемом компакт-диске.