|  |  |
| --- | --- |
| 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 | И9 |  | Систем управления и компьютерных технологий |
|  | шифр |  | наименование |
| Дисциплина | Системное ПО | | |

КУРСОВАЯ РАБОТА

на тему

|  |
| --- |
| Создание компилятора |
| для подмножества языка Go на MASM i586 |
|  |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Выполнили студенты группы | | | | | | И983 |
| Орловский М. Ю. | | | | | | |
| 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 | И9 |  | Систем управления и компьютерных технологий |
|  | шифр |  | наименование |
| Дисциплина | Системное ПО | | |

**ЗАДАНИЕ**

**НА КУРСОВУЮ РАБОТУ**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Студентам | | Орловскому М. Ю. | | | | | | |
|  | | (фамилия, имя, отчество) | | | | | | |
| Научный руководитель | | | Иванов К.С. | | | | | |
|  | | | (ученая степень, звание, Ф.И.О) | | | | | |
| Тема: | Создание компилятора для подмножества языка Go на | | | | | | | |
| MASM 32 | | | | | | | | |
| 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

Оглавление

[1. Введение. 4](#_Toc45120143)

[1.1. Входной язык. 4](#_Toc45120144)

[1.2 Выходной язык. 6](#_Toc45120145)

[2. Лексический анализатор 7](#_Toc45120146)

[2.1 Описание лексики языка. 7](#_Toc45120147)

[2.2 Граф. 8](#_Toc45120148)

[2.3 Реализация лексического анализатора. 10](#_Toc45120149)

[2.4 Результат работы. 11](#_Toc45120150)

[3. Синтаксический анализатор. 12](#_Toc45120151)

[3.1 Синтаксические конструкции. 13](#_Toc45120152)

[3.2 Реализация синтаксического анализатора. 17](#_Toc45120153)

[3.3 Абстрактное синтаксическое дерево. 20](#_Toc45120154)

[3.3.1 Реализация абстрактного синтаксического дерева. 21](#_Toc45120155)

[3.4 Результат работы. 21](#_Toc45120156)

[4. Семантический анализатор. 22](#_Toc45120157)

[4.1 Реализация семантического анализатора. 23](#_Toc45120158)

[5. Генерация кода 24](#_Toc45120159)

[5.1 Реализация генератора кода 24](#_Toc45120160)

[5.2 Результат работы 27](#_Toc45120161)

[6. Заключение 28](#_Toc45120162)

[7. Тестовый пример 29](#_Toc45120163)

[СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ 38](#_Toc45120164)

[ПРИЛОЖЕНИЕ 39](#_Toc45120165)

# 1. Введение

В данной курсовой работе идёт разработка компилятора на языке программирования Go.

**Компилятор** – это программа, которая считывает текст программы, написанной на одном языке – исходном, и транслирует (переводит) его в эквивалентный текст на другом языке – целевом. Одна из важных ролей компилятора состоит в сообщении об ошибках в исходной программе, обнаруженных в процессе трансляции.

Процесс компиляции состоит из нескольких этапов: лексический анализатор, синтаксический анализатор, генератор кода. Разберём чуть по подробнее каждый из них:

**Лексический анализатор** — это часть компилятора, которая выделяет из текста исходной программы токены входного языка. Проще говоря, текст программы преобразуется в набор **токенов**.

**Токен —** это наименьшая единица синтаксиса языка. Токены можно назвать *"строительными кирпичиками"* языка, то есть, это то из чего составляются предложения на этом языке.

**Синтаксический анализатор —** это часть компилятора, которая проверяет синтаксическую корректность кода по набору правил, также называемые грамматикой, а также строящий абстрактное синтаксическое дерево.

**Генерация кода** — это процесс перевода промежуточного представления, в частности абстрактного синтаксического дерева, в выходной код на некотором языке, в том числе и на языке ассемблера.

## 1.1. Входной язык

Go — компилируемый многопоточный язык программирования, разработанный внутри компании Google. Оригинальная архитектура языка была разработана Робертом Гризмером (Robert Griesemer) и корифеями ОС Unix – Робом Пайком (Rob Pike) и Кеном Томпсоном (Ken Thompson). 10 ноября 2009 года были опубликованы исходные тексты реализации языка Go под либеральной открытой лицензией.

Язые Go обладает рядом преимуществ:

* **Простой и понятный синтаксис.** Это делает написание кода приятным занятием.
* **Статическая типизация.** Позволяет избежать ошибок, допущенных по невнимательности, упрощает чтение и понимание кода, делает код однозначным.
* **Скорость и компиляция.** Скорость у Go в десятки раз быстрее, чем у скриптовых языков, при меньшем потреблении памяти. При этом компиляция практически мгновенна. Весь проект компилируется в один бинарный файл, без зависимостей. Также не надо заботиться о памяти, есть сборщик мусора.
* **Отход от ООП.** В языке нет классов, но есть структуры данных с методами. Наследование заменяется механизмом встраивания. Существуют интерфейсы, которые не нужно явно имплементировать, а лишь достаточно реализовать методы интерфейса.
* **Параллелизм.** Параллельные вычисления в языке делаются просто. Горутины легковесны, потребляют мало памяти.
* **Богатая стандартная библиотека.** В языке есть все необходимое для веб-разработки и не только. Количество сторонних библиотек постоянно растет. Кроме того, есть возможность использовать библиотеки C и C++.
* **Возможность писать в функциональном стиле.** В языке есть замыкания (closures) и анонимные функции. Функции являются объектами первого порядка, их можно передавать в качестве аргументов и использовать в качестве типов данных.
* **Авторитетные отцы-основатели и сильное комьюнити.** Роб Пайк, Кен Томпсон, Роберт Гризмер стояли у истоков. Сейчас у языка более 300 контрибьюторов. Язык имеет сильное сообщество и постоянно развивается.
* **Open Source**

Go – практичный язык, где во главу угла поставлены эффективность программ и удобство программиста. Например, встроенные и определяемые пользователем типы данных в языке Go существенно отличаются – операции с первыми из них могут быть значительно оптимизированы, что невозможно для последних. В языке Go также поддерживаются указатели, что позволяет с непринужденностью создавать собственные, весьма сложные типы данных, такие как сбалансированные двоичные деревья.   
Также в языке Go можно усмотреть попытку создать улучшенную версию языка C, даже при том, что простой и ясный синтаксис Go больше напоминает язык Python, что также являлось весомой причиной в выборе этого языка.

**Алфавит.**

Алфавит языка Go составляют символы включенные в 7-битную кодировку [ISO646/ECMA-6](https://en.wikipedia.org/wiki/ISO/IEC_646):   
space, !, ", %, &, (, ), \*, +, -, ., /, 0..9, :, ;, <, =, >, ?, A..Z, \_, a..z.  
Тем не менее, часть символов, используемых в алфавите C++, не входит в кодовую базу этой кодировки: {, }, [, ], #, \, ^, |, ~.

**Ключевые слова.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| break | default | func | interface | select |
| case | defer | go | map | struct |
| chan | else | goto | package | switch |
| const | fallthrough | if | range | type |
| continue | for | import | return | var |

## 1.2 Выходной язык

Ассемблер (от англ. assembler — сборщик) — транслятор исходного текста программы, написанной на языке ассемблера, в программу на машинном языке.

Macro Аssembler (MASM) — ассемблер для процессоров семейства x86. 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.

# 2. Лексический анализатор

Лексический анализатор (ЛА) – считывает и обрабатывает текст программы посимвольно и формирует из них лексемы, которые передаются синтаксическому анализатору. Лексема (token) – минимальная неделимая лексическая единица языка. Также лексический анализатор обрабатывает пробелы и удаляет комментарии.

1. ЛА необязательный этап компиляции, но желательный по следующим   
   причинам:   
   замена идентификаторов, констант, ограничителей и служебных слов   
   лексемами делает программу более удобной для дальнейшей обработки;
2. ЛА уменьшает длину программы, устраняя из ее исходного представления несущественные пробелы и комментарии;
3. Если будет изменена кодировка в исходном представлении программы, то это отразится только на ЛА.

В данной работе мы будем выделять следующие типы лексем:

1. Ключевые слова
2. Идентификаторы
3. Операторы, разделители
4. Литералы
5. Числа

Каждая лексема представляет собой пару чисел вида (n, k), где n – номер таблицы лексем, k - номер лексемы в таблице.   
Входные данные ЛА - текст транслируемой программы на входном языке.   
Выходные данные ЛА - файл лексем в числовом представлении.

## 2.1 Описание лексики языка

**1. Ключевые слова.**

Ключевые слова:

break default func interface select

case defer go map struct

chan else goto package switch

const fallthrough if range type

continue for import return var

**2. Идентификаторы**

<буква> ::= "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z" | "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z"|

<цифра> ::="0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

<идентификатор> ::= <буква> | "\_" {<буква> | <цифра>}

**3. Операторы, разделители.**

<разделитель> ::= '\r' | '\n' | '\t' | '\b' | '\f' | '\v' | ' ' |.

<оператор> ::= **`;`** | **`.`** | **`,`** | **`:`** | **`{`** | **`}`** |  **`(`** | **`)`** |  **`[`** | `**]`** |  **`>`** | **`<`** | | **`=`**| **`+`**| **`-`**| **`\*`** |  **`/`** |.

**4. Литералы**

<число> ::=<цифра> {<цифра>}

<вещественное число> ::={<цифра>} **`.`** {<цифра>}

<символьный> ::= <буква> | <цифра> | <оператор>

<строка> ::= {<буква> | <цифра> | <разделитель>}

**5. Число**

<число> ::=<цифра> {<цифра>}

<вещественное число> ::={<цифра>} **`.`** {<цифра>}

## 2.2 Граф

При считывании из файла очередного символа, пока не дошли до конца файла, проходимся по циклу, в котором определяется состояние автомата. Граф автомата представлен на рисунке.

****

**Пояснения к графу:**

num – переход, когда встречается цифра.

numb – конечное состояние целого числа.

dot – переход, когда встречена точка.

\_ - переход, когда встречен нижний слэш.

alp – переход, когда встречена буква.

indent – конечное состояние идентификатора.

kew w – переход, когда встречена ключевое слово.

Op – переход, когда встречена цифра.

oper – конечное состояние оператора.

liter – конечное состояние литерала

F – конечное состояние. добавление токена в вектор токенов при состоянии default.

## 2.3 Реализация лексического анализатора

Для реализации лексического анализатора было создана 2 класса, Lexan – содержащий КА, разбивающий исходный код на токены и Token – хранящий в себе поступающие токены.

В одном файле с классом Lexan также находится перечисление начальных состояний под названием STATE и содержит следующие данные:  
DEFAULT – используется в КА, когда необходимо задать новое состояние.  
INDETIFIER – используется в КА, когда идёт проверка на то, является ли поступающая строка идентификатором или ключевым словом.

OPERATOR – используется в КА, когда идёт проверка на то, является ли поступающая строка оператором.

STRING – используется в КА, когда идёт проверка на то, является ли поступающая строка литералом.

NUMBER – используется в КА, когда идёт проверка на то, является ли поступающая строка цифрой.

Класс Lexan содержит следующие поля:

keywords – хеш таблица, содержащая ключевые слова.

Operators – хеш таблица, содержащая операторы.

tokenStream – вектор, содержащий поток обработанных токенов.

filePatch – строка, содержащая имя файла.

curState – переменная, отвечающая за текущее состояние КА.

И функции:

setupLexemes() – функция, необходимая для чтения файла.

pushToken (string& lex, TOKEN\_TYPE type) – функция, занимающаяся добавлением токенов в поток.

split () – основную функцию, содержащую конечный автомат.

В одном файле с классом Token также находится перечисление начальных состояний под названием TOKEN\_TYPE и содержит следующие данные:

INDENTIFIER – используется для обозначения, что добавленный токен является идентификатором;

KEYWORD – используется для обозначения, что добавленный токен является ключевым словом;

OPERATOR – используется для обозначения, что добавленный токен является оператором;  
LITERAL – используется для обозначения, что добавленный токен является литералом;

NUMBER – используется для обозначения, что добавленный токен является цифрой;

Класс Token содержит следующие поля:

lex – строка, хранящая в себе значения токена;

type – переменная, хранящая в себе тип токена;

И функцию:

Token (const string& \_lex, TOKEN\_TYPE type) – конструктор, который присваивает необходимые значения полям класса.

## 2.4 Результат работы

Результатом работы лексического анализатора является набор токенов.

# 3. Синтаксический анализатор

Перед синтаксическим анализатором стоят две основные задачи: проверить правильность конструкций программы, которая представляется в виде уже выделенных слов входного языка, и преобразовать ее в вид, удобный для дальнейшей генерации кода. Одним из таких способов представления является дерево синтаксического разбора.

Синтаксически анализатор напрямую работает с цепочкой токенов, выданных лексическим анализатором и проверяет их пригодность определенным правилам языка. Структура конструкций синтаксического анализатора более сложна, чем структура идентификаторов и чисел. Поэтому для описания синтаксиса языка нужны более мощные грамматики нежели регулярные.

Для описания правил грамматики используется форма Бэкуса – Наура. В данной работе используется метод рекурсивного спуска для обработки языковых конструкций языка.

В ходе синтаксического анализа исходный текст преобразуется в структуру данных, обычно — в дерево, которое отражает синтаксическую структуру входной последовательности и хорошо подходит для дальнейшей обработки. Как правило, результатом синтаксического анализа является синтаксическое строение предложения, представленное в виде дерева.

## 3.1 Синтаксические конструкции

SourсeFile  
 [PackageClause](https://golang.org/ref/spec#PackageClause)

"package" [PackageName](https://golang.org/ref/spec#PackageName) [1]  
[ImportDecl](https://golang.org/ref/spec#ImportDecl)

"import" ( [ImportSpec](https://golang.org/ref/spec#ImportSpec) | "(" { [ImportSpec](https://golang.org/ref/spec#ImportSpec) ";" } ")" ) [2]

[TopLevelDecl](https://golang.org/ref/spec#TopLevelDecl)

[Declaration](https://golang.org/ref/spec#Declaration)

[ConstDecl](https://golang.org/ref/spec#ConstDecl)

[TypeDecl](https://golang.org/ref/spec#TypeDecl)

"type" ( [TypeSpec](https://golang.org/ref/spec#TypeSpec) | "(" { [TypeSpec](https://golang.org/ref/spec#TypeSpec) ";" } ")" ). [9]

[VarDecl](https://golang.org/ref/spec#VarDecl)

"var" ( [VarSpec](https://golang.org/ref/spec#VarSpec) | "(" { [VarSpec](https://golang.org/ref/spec#VarSpec) ";" } ")" ) . [12]

[FunctionDecl](https://golang.org/ref/spec#FunctionDecl)

"func" [FunctionName](https://golang.org/ref/spec#FunctionName) [Signature](https://golang.org/ref/spec#Signature) [ [FunctionBody](https://golang.org/ref/spec#FunctionBody) ] . [13],[14], [15]

[MethodDecl](https://golang.org/ref/spec#MethodDecl)  
 "func" [Receiver](https://golang.org/ref/spec#Receiver) [MethodName](https://golang.org/ref/spec#MethodName) [Signature](https://golang.org/ref/spec#Signature) [ [FunctionBody](https://golang.org/ref/spec#FunctionBody) ] . [21], [22], [14], [15]

[1] [PackageName](https://golang.org/ref/spec#PackageName) ::= [identifier](https://golang.org/ref/spec#identifier)

[2] ImportSpec = [ "." | [PackageName](https://golang.org/ref/spec#PackageName) ] [ImportPath](https://golang.org/ref/spec#ImportPath) .[1], [3]

[3] ImportPath = [string\_lit](https://golang.org/ref/spec#string_lit) . [4]

[4] string\_lit = [raw\_string\_lit](https://golang.org/ref/spec#raw_string_lit) | [interpreted\_string\_lit](https://golang.org/ref/spec#interpreted_string_lit) . [5], [6]

[5] raw\_string\_lit = "`" { [unicode\_char](https://golang.org/ref/spec#unicode_char) | [newline](https://golang.org/ref/spec#newline) } "`" .

[6] interpreted\_string\_lit = `"` { [unicode\_value](https://golang.org/ref/spec#unicode_value) | [byte\_value](https://golang.org/ref/spec#byte_value) } `"` [7], [8]

[7] [unicode\_char](https://golang.org/ref/spec#unicode_char) ::= произвольный код Unicode, кроме новой строки

[8] [newline](https://golang.org/ref/spec#newline) ::= кодовая точка Unicode U + 000A

[9] [TypeSpec](https://golang.org/ref/spec#TypeSpec) ::= [AliasDecl](https://golang.org/ref/spec#AliasDecl) | [TypeDef](https://golang.org/ref/spec#TypeDef) [10], [11]

[10] [AliasDecl](https://golang.org/ref/spec#AliasDecl) ::= [identifier](https://golang.org/ref/spec#identifier) "=" [Type](https://golang.org/ref/spec#Type)

[11] [TypeDef](https://golang.org/ref/spec#TypeDef) ::= [identifier](https://golang.org/ref/spec#identifier) [Type](https://golang.org/ref/spec#Type)

[12] VarSpec ::= [IdentifierList](https://golang.org/ref/spec#IdentifierList) ( [Type](https://golang.org/ref/spec#Type) [ "=" [ExpressionList](https://golang.org/ref/spec#ExpressionList) ] | "=" [ExpressionList](https://golang.org/ref/spec#ExpressionList) ) .

[13] FunctionName = [identifier](https://golang.org/ref/spec#identifier) .

[14] Signature ::= [Parameters](https://golang.org/ref/spec#Parameters) [ [Result](https://golang.org/ref/spec#Result) ] . [16], [17]

[15] FunctionBody = [Block](https://golang.org/ref/spec#Block) . [23]

[16] Parameters ::= "(" [ [ParameterList](https://golang.org/ref/spec#ParameterList) [ "," ] ] ")" . [18]

[17] Result ::= [Parameters](https://golang.org/ref/spec#Parameters) | [Type](https://golang.org/ref/spec#Type) . [16]

[18] ParameterList ::= [ParameterDecl](https://golang.org/ref/spec#ParameterDecl) { "," [ParameterDecl](https://golang.org/ref/spec#ParameterDecl) } . [19]

[19] ParameterDecl ::= [ [IdentifierList](https://golang.org/ref/spec#IdentifierList) ] [ "..." ] [Type](https://golang.org/ref/spec#Type) . [20]

[20] IdentifierList ::= [identifier](https://golang.org/ref/spec#identifier) { "," [identifier](https://golang.org/ref/spec#identifier) }

[21] Receiver ::= [Parameters](https://golang.org/ref/spec#Parameters) . [16]

[22] MethodName ::= [identifier](https://golang.org/ref/spec#identifier) .

[23] Block ::= "{" [StatementList](https://golang.org/ref/spec#StatementList) "}" . [24]

[24] StatementList ::= { [Statement](https://golang.org/ref/spec#Statement) ";" } . [25]

[25] Statement =

[Declaration](https://golang.org/ref/spec#Declaration) | [SimpleStmt](https://golang.org/ref/spec#SimpleStmt) | [ReturnStmt](https://golang.org/ref/spec#ReturnStmt) | [BreakStmt](https://golang.org/ref/spec#BreakStmt) | [ContinueStmt](https://golang.org/ref/spec#ContinueStmt) |

[Block](https://golang.org/ref/spec#Block) | [IfStmt](https://golang.org/ref/spec#IfStmt) | [ForStmt](https://golang.org/ref/spec#ForStmt) . [26], [27], [28], [29], [23], [30], [31]

[26] [SimpleStmt](https://golang.org/ref/spec#SimpleStmt) ::= [EmptyStmt](https://golang.org/ref/spec#EmptyStmt) | [ExpressionStmt](https://golang.org/ref/spec#ExpressionStmt) | [SendStmt](https://golang.org/ref/spec#SendStmt) | [IncDecStmt](https://golang.org/ref/spec#IncDecStmt) | [Assignment](https://golang.org/ref/spec#Assignment) | [ShortVarDecl](https://golang.org/ref/spec#ShortVarDecl) . [32], [33], [34], [35]

[27] [ReturnStmt](https://golang.org/ref/spec#ReturnStmt) ::= "return" [ [ExpressionList](https://golang.org/ref/spec#ExpressionList) ] . [39]

[28] [BreakStmt](https://golang.org/ref/spec#BreakStmt) ::= "break" [ [Label](https://golang.org/ref/spec#Label) ] . [40]

[29] [ContinueStmt](https://golang.org/ref/spec#ContinueStmt) ::= "continue" [ [Label](https://golang.org/ref/spec#Label) ] . [40]

[30] [IfStmt](https://golang.org/ref/spec#IfStmt) ::= "if" [ [SimpleStmt](https://golang.org/ref/spec#SimpleStmt) ";" ] [Expression](https://golang.org/ref/spec#Expression) [Block](https://golang.org/ref/spec#Block) [ "else" ( [IfStmt](https://golang.org/ref/spec#IfStmt) | [Block](https://golang.org/ref/spec#Block) ) ] . [41], [34], [23], [30], [23]

[31] [ForStmt](https://golang.org/ref/spec#ForStmt) ::= "for" [ [Condition](https://golang.org/ref/spec#Condition) | [ForClause](https://golang.org/ref/spec#ForClause) | [RangeClause](https://golang.org/ref/spec#RangeClause) ] [Block](https://golang.org/ref/spec#Block) . [42], [43], [44], [23]

[32] [EmptyStmt](https://golang.org/ref/spec#EmptyStmt) ::= .

[33] [ExpressionStmt](https://golang.org/ref/spec#ExpressionStmt) ::= [Expression](https://golang.org/ref/spec#Expression) [34]

[34] [Expression](https://golang.org/ref/spec#Expression) ::= [UnaryExpr](https://golang.org/ref/spec#UnaryExpr) | [Expression](https://golang.org/ref/spec#Expression) [binary\_op](https://golang.org/ref/spec#binary_op) [Expression](https://golang.org/ref/spec#Expression) . [35], [36]

[35] [UnaryExpr](https://golang.org/ref/spec#UnaryExpr) ::= [PrimaryExpr](https://golang.org/ref/spec#PrimaryExpr) | [unary\_op](https://golang.org/ref/spec#unary_op) [UnaryExpr](https://golang.org/ref/spec#UnaryExpr) . [37], [38]. [35]

[37] [PrimaryExpr](https://golang.org/ref/spec#PrimaryExpr) ::=

[38] unary\_op ::= "+" | "-" | "!" | "^" | "\*" | "&" | "<-" .

[39] [ExpressionList](https://golang.org/ref/spec#ExpressionList) ::= [Expression](https://golang.org/ref/spec#Expression) { "," [Expression](https://golang.org/ref/spec#Expression) } . [34]

[40] [Label](https://golang.org/ref/spec#Label) ::= [identifier](https://golang.org/ref/spec#identifier)

[41] [SimpleStmt](https://golang.org/ref/spec#SimpleStmt) ::= [EmptyStmt](https://golang.org/ref/spec#EmptyStmt) | [ExpressionStmt](https://golang.org/ref/spec#ExpressionStmt) | [SendStmt](https://golang.org/ref/spec#SendStmt) | [IncDecStmt](https://golang.org/ref/spec#IncDecStmt) | [Assignment](https://golang.org/ref/spec#Assignment) | [ShortVarDecl](https://golang.org/ref/spec#ShortVarDecl) . [32], [33], [47], [49], [50], [51].

[42] [Condition](https://golang.org/ref/spec#Condition) ::= [Expression](https://golang.org/ref/spec#Expression) [34]

[43] [ForClause](https://golang.org/ref/spec#ForClause) ::= [ [InitStmt](https://golang.org/ref/spec#InitStmt) ] ";" [ [Condition](https://golang.org/ref/spec#Condition) ] ";" [ [PostStmt](https://golang.org/ref/spec#PostStmt) ] . [45], [42], [46]

[44] [RangeClause](https://golang.org/ref/spec#RangeClause) ::= [ [ExpressionList](https://golang.org/ref/spec#ExpressionList) "=" | [IdentifierList](https://golang.org/ref/spec#IdentifierList) ":=" ] "range" [Expression](https://golang.org/ref/spec#Expression) . [39], [20], [34].

[45] [InitStmt](https://golang.org/ref/spec#InitStmt) ::= [SimpleStmt](https://golang.org/ref/spec#SimpleStmt) . [26]

[46] [PostStmt](https://golang.org/ref/spec#PostStmt) ::= [SimpleStmt](https://golang.org/ref/spec#SimpleStmt) . [26]

[47] [SendStmt](https://golang.org/ref/spec#SendStmt) ::= [Channel](https://golang.org/ref/spec#Channel) "<-" [Expression](https://golang.org/ref/spec#Expression) . [48], [34]

[48] Channel = [Expression](https://golang.org/ref/spec#Expression) . [34]

[49] IncDecStmt = [Expression](https://golang.org/ref/spec#Expression) ( "++" | "--" ) . [34]

[50] Assignment = [ExpressionList](https://golang.org/ref/spec#ExpressionList) [assign\_op](https://golang.org/ref/spec#assign_op) [ExpressionList](https://golang.org/ref/spec#ExpressionList) . [39], [51], [39]

[51] assign\_op = [ [add\_op](https://golang.org/ref/spec#add_op) | [mul\_op](https://golang.org/ref/spec#mul_op) ] "=". [52], [53]

[52] add\_op = "+" | "-" | "|" | "^" .

[53] mul\_op = "\*" | "/" | "%" | "<<" | ">>" | "&" | "&^" .

[54] ShortVarDecl ::= [IdentifierList](https://golang.org/ref/spec#IdentifierList) ":=" [ExpressionList](https://golang.org/ref/spec#ExpressionList) . [20], [39]

## 3.2 Реализация синтаксического анализатора

Для реализации класса реализации синтаксического анализатора был создан класс Parser, для разбивания входного текста программы на синтаксические конструкции, с целью проверки правильности грамматики языка Go и создания синтаксического дерева.

Класс Parser содержит следующие поля:

Lexan\* lexan – класс, описанный в разделе «Лексческий анализатор», необходимый для работы с токенами, на который он разбивает входной текст программы.

AST\* ast – класс, реализующий абстракное синтаксическое дерево для удобного приставления разобранных синтаксических конструкций.

Класс Parser содержит следующие методы:

Parse() – метод, начинающий работу парсера, содержит метод SourсeFile.

SourсeFile() – метод, содержащий в себе методы, package и topLvlDecl.

package() – метод в котором проверяется корректность написания имени программы.

import() – метод, отвечающий за правильность подключения других пакетов.

TopLvlDecl() – метод, отвечающий создания узла, с помощью метода tld и добавление его в дерево, если узел создался успешно.

tld() – метод, содержащий функции проверки наличия объявления функций или переменных и добавление их в аст.

varDecl() – метод, отвечающий за правильность реализации объявлений переменных.

varSpec() – метод, вложенный в varDecl, определяющий правильность определения переменных после ключевого слово “var”.

type() – метод, вложенный в varDecl, определяющий правильность определения типа переменных после ключевого слово “var”.

funcDecl() – метод, отвечающий за правильность реализации объявлений функций, содержит вложенные методы signature и bloc.

signature() – метод, проверяющий синтаксическую правильность части написанной в операторных скобках и наличие имени функции.

par() – метод, проверяющие наличие операторных скобок и содержащий метод parList между ними.

parList() – метод, проверяющий синтаксическую правильность перечисления выражений находящуюся между операторных скобок.

parDecl() – метод, проверяющий синтаксическую правильность выражения находящегося между операторных скобок.

block() – метод, проверяющий синтаксическую правильность блока функции.

forStmt() – метод, проверяющий синтаксическую правильность функции for.

forClause() – метод, проверяющий синтаксическую правильность условия функции for.

ifStmt() – метод, проверяющий синтаксическую правильность функции if.  
stmtList() – метод, проверяющий синтаксическую правильность перечисления объявлений и операторов.

stmt() – метод, проверяющий синтаксическую правильность объявлений и операторов.

simpleStmt() – метод, проверяющий синтаксическую правильность простых выражений.

shvd() – метод, проверяющий синтаксическую правильность конструкции присваивания.

incdec() – метод, проверяющий синтаксическую правильность конструкции префиксного инкремента и декремента.

assign() – метод, проверяющий синтаксическую правильность конструкции присваивания.

returnStmt() – метод, проверяющий синтаксическую правильность функции return.

breakStmt() – метод, проверяющий синтаксическую правильность функции break.  
continueStmt() – метод, проверяющий синтаксическую правильность функции continue.

exprList()– метод, проверяющий синтаксическую правильность перечисления выражений.

expr() – метод, проверяющий синтаксическую правильность выражения.

unary() – метод, проверяющий синтаксическую правильность унарных операндов.

binaryOp() – метод, проверяющий синтаксическую правильность бинарных операндов.

relOp() – метод, проверяющий синтаксическую правильность знаков сравнения.

addOp() – метод, проверяющий синтаксическую правильность сложения и вычитания.

mulOp() – метод, проверяющий синтаксическую правильность умножения и деления.

unaryOp() – метод, проверяющий синтаксическую правильность унарных плюсов и минусов.

primary() – метод, проверяющий синтаксическую правильность основных выражений.

operand() – метод, проверяющий синтаксическую правильность операндов в основных выражений.

idList() – метод, проверяющий синтаксическую правильность перечисления значений.

analyseStmt() – метод анализирующий выражение.

## 3.3 Абстрактное синтаксическое дерево

В результате работы синтаксического анализатора получается синтаксическое дерево. Синтаксическое дерево (дерево операций) - это структура, представляющая собой результат работы синтаксического анализатора. Она отражает синтаксис конструкций входного языка и явно содержит в себе полную взаимосвязь операций.

В синтаксическом дереве внутренние узлы (вершины) соответствуют операциям, а листья представляют собой операнды. Как правило, листья синтаксического дерева связаны с записями в таблице идентификаторов. Структура синтаксиче­ского дерева отражает синтаксис языка программирования, на котором написана исходная программа.

Для построения дерева используется структура, имеющая указатели на левого и правого сыновей. Таким образом, синтактическое дерево является бинарным.

В данной работе используется метод рекурсивного спуска для обработки языковых конструкций языка.

В целом, это значит, что дерево будет хранить в себе все конструкции языка, каждая из которых будет представлена отдельным поддеревом.

### **3.3.1 Реализация абстрактного синтаксического дерева**

Для реализации AST были созданы две структуры: Node – структура необходимая для создания узла и Info – структура содержащая информацию о переменных, которые добавляют в дерево. Также был реализован класс AST, необходимый для создания структурирования дерева.

Класс AST содержит следующие поля:

Node\* \_head – указатель на «голову» дерева.

Класс AST содержит следующие методы:

printRecursive (Node\*, int tabs, int isDeep) – метод отвечающий за вывод построенного дерева.

print() – метод вызывающий функцию printRecursive.

log(const string& path) – метод выводит название файла.

traverseTree(void (\*f)(Node \*, Info &)) – метод, для прохождения дерева.

CLR(Node\* node, void (\*f)(Node\*, Info&), Info&) – метод для прохождения через центр, лево, право.

LCR(Node\* node, void (\*f)(Node\*, Info&), Info&) - метод для прохождения через лево, центр, право.

Структура Node содержит следующие поля:

string lex – название добавляемой в дерево лексемы.  
TOKEN\_TYPE type {TOKEN\_TYPE::KEYWORD} – название типа токена, который добавляется в дерево.  
Node\* l {nullptr} – указатель на левый узел.  
Node\* r {nullptr} – указатель на правый узел.  
bool isDecl {false} – булевский тип данных отвечающий за то, был ли продекларирована переменная.

Структура Info содержит следующие поля:

int depth – глубина токена, который добавляют в дерево.  
string curType – текущий тип добавляемого токена.

## 3.4 Результат работы.

Результатом работы синтаксического анализатора является абстрактное синтаксическое дерево, в котором в абстрактном виде описаны все конструкции кода.

# 4. Семантический анализатор.

Семантический анализ — это этап сбора дополнительной информации, например, выделения всех переменных или всех функций и т.д., а также этап для проверок корректности кода, например, проверки, что переменная была объявлена до её использования. Синтаксический анализ для своей работы использует AST.

Семантический анализ представляет из себя многократные обходы дерева синтаксического анализа. Суть в том, что во время обхода мы посещаем каждый узел дерева, а значит мы можем в зависимости от типа данного узла делать некоторые действия, необходимые для текущей проверки.

Благодаря семантическому анализу мы решаем следующие проблемы:

1. Различные проверки корректности кода, как уже было сказано ранее, например, использование переменной до ее объявления.

2. Различие переменных с одинаковым названием в разных блоках. Так например в Go блоки for могут содержать переменную i в несколько подряд идущих циклах.

3. Проверка бинарных операторов на корректность операндов.

4. Сборка особой информации, используемой в генераторе кода.

5. Проверка корректной инициализации переменных – количество операндов и их типы.

## 4.1 Реализация семантического анализатора

Для реализации семантического анализатора был создан класс Translator.

Класс Translator содержит следующие поля необходимые для реализации семантического анализатора:

ofstream out – структура необходимая для записи в файл  
string path – строка хранящая название файла.  
AST\* ast – указатель на абстрактное синтаксическое дерево.  
static RuntimeVars\* run – объект класса хранящий информацию о текущей переменной.

Класс Translator содержит следующие методы поля необходимые для реализации семантического анализатора:

semAn() – главный метод, производящий семантический анализ.  
  
appendVars(Node\*, Info&) – метод, добавляющий переменную в хеш таблицу.  
addVar(Node\*, Info&) – метод, добавляет лист.  
countVars(Node \*node, Info &info) – метод, который запускает счётчик переменных в дереве.  
countVar(Node \*node, Info &info, int &count) – метод, который последовательно увеличивает счётчик при прохождение каждой новой переменной.  
checkVars(Node\* node, Info& info) – метод, который просматривает переменные на их объявление.  
checkVar(Node\*, Info&) – метод, который просматривает была ли переменная до этого объявлена.  
  
binOps(Node\*, Info&) – метод, подсчитывающий количество переменных при бинарных операторах.  
srcHead(Node\* node, Info& info) – метод, проверяющий, является ли текущий узел «головой» дерева.  
  
appendImports(Node\* node, Info& info) – метод, добавляющий пакеты в хэш таблицу.  
checkImports(Node\* node, Info& info) – метод, проверяющий был ли инициализирован необходимый пакет.

Результатом работы семантического анализатора является таблица переменных. А также, в случае ошибок, отчет о ошибке.

# 5. Генерация кода

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

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

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

## 5.1 Реализация генератора кода

Для реализации генератора кода в классе Translator, были добавлены следующие поля и методы.

Поля реализованные в классе Translator:

string asmData – поле содержащая сегмент данных программы.

string asmPreMain – поле содержащая блок перед телом.  
string asmMain – поле содержащая тело программы.  
  
string\* asmCur {nullptr} – поле содержащая указатель на текущей.  
int stackBytes {8} – поле содержащая количество бит для выделения.  
bool isLitPushed {false} – поля являющаяся флагом для добавления в лист.  
bool isPrintln {false} – поле являющаяся флагом для функции println.

Также была реализовано перечисление asmPlace, которая содержит следующие значения

DATA – обозначающая блок, хранящий переменные.

MAIN – обозначающий блок тела программы.

PREMAIN – обозначающий блок перед телом программы.

Методы реализованные для генерации кода:

void generate() – главный метод, который отвечает за запуск генерации кода.  
  
void asmBlock() - метод, для разбора конструкций  
void \_asmBlock(Node\* node) – рекурсивная функция разбирающая конструкции: var, --, ++, =, :=, (), for, else.  
  
void asmIf(Node\* node) – метод, который отвечает за генерацию процедур if.

void \_asmIf(Node\* node, int unqLabel, size\_t elseLabels) – рекурсивная функция метода asmIf.  
void asmVarDecl(Node \*varList, Node\* expList) – метод, который отвечает за генерацию инициализации переменных.  
void \_asmVarDecl(Node \*&var, Node \*node) - ) – рекурсивная функция метода asmIf.  
  
void asmExpr(Node \*node) – метод, который отвечает за генерацию выражения.  
void asmCond(Node\* node) – метод, который отвечает за генерацию состояния.  
  
void asmPrint(Node \*node) – метод, который отвечает за генерацию процедур print.   
void asmSqrt(Node \*node) – метод, который отвечает за генерацию процедур print.   
  
inline void stringLits() – записать в сегмент данных литералы.  
inline void sqrtOp() – запись операнда для извлечения квадратного корня.  
inline void localVars() – записываем в хэш таблицу локальные переменные.  
inline void printProc() – объявляем процедуру print.

Дальше мы реализовываем список команд для работы с Assembler Masm.  
inline void push(const string& str) – метод, для записи операнда в стек.

inline void pop(const string& str) – метод, для пересылки слова из стека в регистр.   
inline void add(const string& lhs, const string& rhs) – метод, для сложения двух операндов.  
inline void sub(const string& lhs, const string& rhs) – метод, для вычитания двух операндов.  
inline void imul(const string& lhs, const string& rhs) – метод, для знакового умножения.  
inline void div(const string& str) – метод, для беззнакового деления.  
inline void mov(const string& lhs, const string& rhs);  
inline void raw(const string& str)   
inline void cmp(const string& lhs, const string& rhs) – метод, для сравнения операндов.  
inline void jmp(const string& str) – метод, для прямого перехода.  
inline void je(const string& str) – метод, для установки флага, что результат равен нулю.  
inline void jne(const string& str) – метод, для установления флага, что результат не равен нулю.  
inline void jl(const string& str) – метод, для перехода после команды для знака меньше.  
inline void jle(const string& str) - метод, для перехода после команды для знака меньше или равно.  
inline void jg(const string& str) - метод, для перехода после команды для знака больше.  
inline void jge(const string& str) - метод, для перехода после команды для знака больше или равно.  
  
inline void label(const string& str) – метод, для перехода на новую строку.  
inline void finit() – метод для реализации сопроцессора.  
inline void fild(const string& str) – метод для загрузи операнда в вершину стека.  
inline void fdiv(const string& lhs, const string& rhs) – метод, для реализации вещественного деления.  
inline void fistp(const string& str) – метод, сохранения вершины стека в память с выталкиванием.  
inline void inc(const string& str) – метод, который в ассемблере увеличивает число на единицу.  
inline void dec(const string& str) - метод, который в ассемблере уменьшает число на единицу.  
  
inline void setPtr(asmPlace place) – метод, для перемещения указателя на часть программы.

## 5.2 Результат работы

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

# 6. Заключение

Результатом выполнения данной курсовой работы стало создание простейшего компилятора для языка высокого уровня Oberon-2 на язык низкого уровня ассемблер MASM. В процессе работы были получены знания о поэтапной разработке компилятора, получен опыт взаимодействия с грамматикой языка.

Полученная программа работает корректно. Компилятор был протестирован на различны примерах и результаты были полностью работоспособными и выполняли те же действия, что и программы на исходном языке.

# 7. Тестовый пример

## 7.1 Ниже представлен пример кода на языке Go и выходная программа на MASM32

|  |  |  |
| --- | --- | --- |
| Go | MASM32 | |
| package main  import "fmt"  import "math"  func main() {  var a, t, j = 5, -2 \* 4 + 9, 5  alpha, beta, gamma := (-400 + 1000) / 32, 8400, -7 / 3  var b = math.sqrt(25)  fmt.Println(alpha, beta, gamma)  fmt.Println("I can print like anything!", 1, 2, 3 + (8 / 2) \* 4 - 4, gamma)  for i := 0; i < a; i++ {  if c := math.sqrt(3 \* 3 + 4 \* 4); b == 5 {  fmt.Println("It is b:", b)  fmt.Println("And c:", c)  b = 8  } else if c = 3; a < 5 {  fmt.Println("It will not be printed")  } else if gamma < 0 {  fmt.Println("It will be printed eventually, gamma:", gamma)  fmt.Println("And c:", c)  break;  }  }  } | | .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  sqrtOp dd 0  print\_0 db "%d", 0  print\_1 db "%s", 0  print\_2 db "%d ", 0  print\_3 db "%s ", 0  print\_nl db 10, 0  LIT3534916879 db "fmt", 0  LIT2971646047 db "It will not be printed", 0  LIT1722247484 db "math", 0  LIT4057494348 db "It will be printed eventually, gamma:", 0  LIT3068837375 db "And c:", 0  LIT2212957155 db "I can print like anything!", 0  LIT2791304878 db "It is b:", 0  data ends  text segment  gamma\_int\_1 = -8  j\_int\_1 = -12  b\_int\_1 = -16  c\_int\_3 = -20  a\_int\_1 = -24  beta\_int\_1 = -28  alpha\_int\_1 = -32  i\_int\_2 = -36  t\_int\_1 = -40  print PROC  enter 0, 0  mov ecx, [ebp + 8]  mov eax, [ebp + 12]  invoke crt\_printf, ecx, eax  leave  ret 4  print ENDP  \_\_start:  enter 40, 0  push 5  pop eax  mov a\_int\_1[ebp], eax  push 2  pop eax  imul eax, -1  push eax  push 4  pop eax  pop ebx  imul eax, ebx  push eax  push 9  pop eax  pop ebx  add eax, ebx  push eax  pop eax  mov t\_int\_1[ebp], eax  push 5  pop eax  mov j\_int\_1[ebp], eax  push 400  pop eax  imul eax, -1  push eax  push 1000  pop eax  pop ebx  add eax, ebx  push eax  push 32  mov edx, 0  pop ebx  cdq  pop eax  idiv ebx  push eax  pop eax  mov alpha\_int\_1[ebp], eax  push 8400  pop eax  mov beta\_int\_1[ebp], eax  push 7  pop eax  imul eax, -1  push eax  push 3  mov edx, 0  pop ebx  cdq  pop eax  idiv ebx  push eax  pop eax  mov gamma\_int\_1[ebp], eax  push 25  pop sqrtOp  finit  fild sqrtOp  fsqrt  fistp sqrtOp  push sqrtOp  pop eax  mov b\_int\_1[ebp], eax  push 25  pop sqrtOp  finit  fild sqrtOp  fsqrt  fistp sqrtOp  push sqrtOp  mov ecx, 0  push alpha\_int\_1[ebp]  push offset print\_2  call print  push beta\_int\_1[ebp]  push offset print\_2  call print  push gamma\_int\_1[ebp]  push offset print\_2  call print  push offset print\_nl  call print  mov ecx, 0  push offset LIT2212957155  push offset print\_3  call print  push 1  push offset print\_2  call print  push 2  push offset print\_2  call print  push 3  push 8  push 2  mov edx, 0  pop ebx  cdq  pop eax  idiv ebx  push eax  push 4  pop eax  pop ebx  imul eax, ebx  push eax  push 4  pop ebx  pop eax  sub eax, ebx  push eax  pop eax  pop ebx  add eax, ebx  push eax  push offset print\_2  call print  push gamma\_int\_1[ebp]  push offset print\_2  call print  push offset print\_nl  call print  push 0  pop eax  mov i\_int\_2[ebp], eax  LOOPSTART\_25043:  push i\_int\_2[ebp]  pop ecx  push a\_int\_1[ebp]  pop edx  cmp ecx, edx  jge CMPNE\_515  push 1  jmp CMPEND\_515  CMPNE\_515:  push 0  CMPEND\_515:  pop eax  cmp eax, 0  je LOOPEND\_25043  push 3  push 3  pop eax  pop ebx  imul eax, ebx  push eax  push 4  push 4  pop eax  pop ebx  imul eax, ebx  push eax  pop eax  pop ebx  add eax, ebx  push eax  pop sqrtOp  finit  fild sqrtOp  fsqrt  fistp sqrtOp  push sqrtOp  pop eax  mov c\_int\_3[ebp], eax  push b\_int\_1[ebp]  pop ecx  push 5  pop edx  cmp ecx, edx  jne CMPNE\_13983  push 1  jmp CMPEND\_13983  CMPNE\_13983:  push 0  CMPEND\_13983:  pop eax  cmp eax, 0  je IFELSE\_320880  mov ecx, 0  push offset LIT2791304878  push offset print\_3  call print  push b\_int\_1[ebp]  push offset print\_2  call print  push offset print\_nl  call print  mov ecx, 0  push offset LIT3068837375  push offset print\_3  call print  push c\_int\_3[ebp]  push offset print\_2  call print  push offset print\_nl  call print  push 8  pop eax  mov b\_int\_1[ebp], eax  jmp IFEND\_32088  IFELSE\_320880:  push 3  pop eax  mov c\_int\_3[ebp], eax  push a\_int\_1[ebp]  pop ecx  push 5  pop edx  cmp ecx, edx  jge CMPNE\_3108  push 1  jmp CMPEND\_3108  CMPNE\_3108:  push 0  CMPEND\_3108:  pop eax  cmp eax, 0  je IFELSE\_320881  mov ecx, 0  push offset LIT2971646047  push offset print\_3  call print  push offset print\_nl  call print  jmp IFEND\_32088  IFELSE\_320881:  push gamma\_int\_1[ebp]  pop ecx  push 0  pop edx  cmp ecx, edx  jge CMPNE\_15862  push 1  jmp CMPEND\_15862  CMPNE\_15862:  push 0  CMPEND\_15862:  pop eax  cmp eax, 0  je IFELSE\_320882  mov ecx, 0  push offset LIT4057494348  push offset print\_3  call print  push gamma\_int\_1[ebp]  push offset print\_2  call print  push offset print\_nl  call print  mov ecx, 0  push offset LIT3068837375  push offset print\_3  call print  push c\_int\_3[ebp]  push offset print\_2  call print  push offset print\_nl  call print  jmp IFEND\_32088  IFELSE\_320882:  IFEND\_32088:  LOOPINC\_25043:  push i\_int\_2[ebp]  pop eax  inc eax  mov i\_int\_2[ebp], eax  jmp LOOPSTART\_25043  LOOPEND\_25043:  leave  ret  text ends  end \_\_start |

# СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. [https://golang.org/ref/](https://golang.org/ref/spec#ImportDecl) The Go Programming Language Specification
2. <https://tproger.ru/translations/golang-basics/> Golang: основы для начинающих.
3. https://habr.com/ru/post/423077/ Руководство по ассемблеру x86 для начинающих
4. <https://c9x.me/x86/> x86 Instruction Set Reference
5. Вирт Н. Построение компиляторов. — М.: ДМК Пресс, 2010

# ****ПРИЛОЖЕНИЕ****

Текст программы на прилагаемом компакт-диске.