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 (шифр) Информационные и управляющие системы (наименование)

Кафедра И9 (шифр) Систем управления и компьютерных технологий (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

Студенту 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
2) Синтаксический анализ, грамматика, дерево разбора
3) Семантический анализ, проверка переменных
4) 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

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 Описание входного и выходного языка

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.

1.1 Алфавит

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

1.2 Ключевые слова

Таблица 1 – 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

1.3 Выходной язык

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

Macro Assembler (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.

В ЛА реализуются следующие достоинства:

1) Замена идентификаторов, констант, ограничителей и служебных слов лексемами делает программу более удобной для дальнейшей обработки;

2) Он 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:

- ключевые слова;
- идентификаторы;
- операторы, разделители;
- литералы;
- 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

Ключевые слова рассмотрены в разделе 1.2.

Идентификаторы:

1) <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";

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

3) <идентификатор> ::= <буква> | "_" {<буква> | <цифра>}.

Операторы, разделители:

1) <разделитель> ::= '\r' | '\n' | '\t' | '\b' | '\f' | '\v' | ' ' ;

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

Литералы:

1) <число> ::=<цифра> {<цифра>};

2) <вещественное число> ::={<цифра>} `.` {<цифра>};

3) <символьный> ::= <буква> | <цифра> | <оператор>;

4) <строка> ::= {<буква> | <цифра> | <разделитель>}.

Число:

1) <число> ::=<цифра> {<цифра>};

2) <вещественное число> ::={<цифра>} `.` {<цифра>}.

2.2 Граф переходов автомата

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 1.

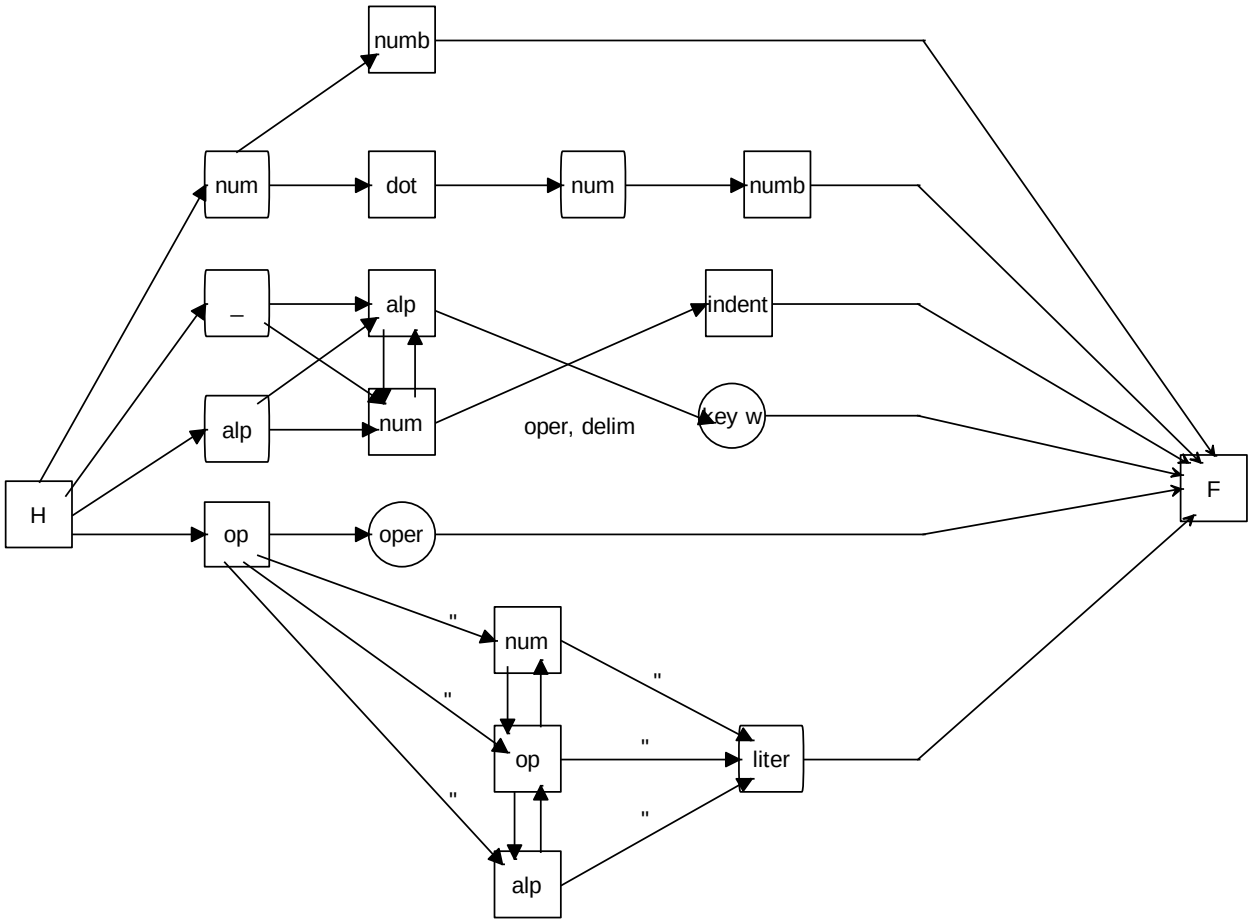

Рисунок 1 – Граф переходов автомата

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:

1. DEFAULT – используется в КА, когда необходимо задать новое состояние.

2. INDETIFIER – используется в КА, когда идёт проверка на то, является ли поступающая строка идентификатором или ключевым словом.

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

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

5. 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. Также содержит 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:

1. INDENTIFIER – используется для обозначения, что добавленный токен является идентификатором.

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

3. OPERATOR – используется для обозначения, что добавленный токен является оператором.

4. LITERAL – используется для обозначения, что добавленный токен является литералом.

5. NUMBER – используется для обозначения, что добавленный токен является цифрой.

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 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.

Методы класса Parser записаны в таблице 2.

Таблица 2 – Методы класса Parser

Parse()	метод, начинающий работу парсера, содержит метод SourceFile.
SourceFile()	метод, содержащий в себе методы, package и topLvlDecl. Реализует полный разбор программы.
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. Скобки нужны для арифметических выражений.

Продолжение таблицы 2.

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.

Продолжение таблицы 2.

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.

3.2 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.

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 поле: Node* _head – указатель на «голову» дерева.

Класс AST содержит следующие методы:

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

2) print() – метод вызывающий функцию printRecursive;

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

4) traverseTree(void (*f)(Node *, Info &)) – метод, для прохождения дерева;

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

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

Структура Node содержит следующие поля:

1) string lex – название добавляемой в дерево лексемы;

2) TOKEN_TYPE type {TOKEN_TYPE::KEYWORD} – название типа токена, который добавляется в дерево;

3) Node* l {nullptr} – указатель на левый узел;

4) Node* r {nullptr} – указатель на правый узел;

5) bool isDecl {false} – булевский тип данных отвечающий за то, был ли продекларирована переменная.

3.3 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:

1) ofstream out – структура необходимая для записи в файл;

2) string path – строка хранящая название файла;

3) AST* ast – указатель на абстрактное синтаксическое дерево;

4) static RuntimeVars* run – объект класса хранящий информацию о текущей переменной.

Класс Translator следующий метод реализации анализа: semAn() – главный метод, производящий семантический анализ.

4.2 Результат работы

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:

1) DATA – обозначающая блок, хранящий переменные;

2) MAIN – обозначающий блок тела программы;

3) PREMAIN – обозначающий блок перед телом программы.

Методы реализованные для генерации кода описаны в таблице 3.

Таблица 3 – методы 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

Продолжение таблицы 3

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.
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.

Продолжение таблицы 3

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);	команда mov
inline void raw(const string& str)	запись текста
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.

Окончание таблицы 3

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 либо 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.

6 Тестовый пример

Ниже представлен пример кода на языке Go и выходная программа на MASM32.

Таблица 4 – Тестовый пример

Go	MASM32

package main
 	
import "fmt"
import "math"
 	
func main() {
  var a = 10
  var b = 20
  fmt.Println(a, b)
  for i := 0; i < a; i++ {
  if i > 5 {
  break;
  }
  }
  return;
}
	.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
	  LIT1722247484 db "math", 0
	
	data ends
	
	text segment
	
	b_int_1 = -8
	a_int_1 = -12
	i_int_2 = -16
	print PROC
	  enter 0, 0
	  mov ecx, [ebp + 8]
	  mov eax, [ebp + 12]
	  invoke crt_printf, ecx, eax

Продолжение таблицы 4

Go	MASM32

	Leave
	  ret 4
	print ENDP
	
	__start:
	  enter 16, 0
	  push 10
	  pop eax
	  mov a_int_1[ebp], eax
	  push 20
	  pop eax
	  mov b_int_1[ebp], eax
	  mov ecx, 0
	  push a_int_1[ebp]
	  push offset print_2
	  call print
	  push b_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_22386:
	  push i_int_2[ebp]
	  pop ecx
	  push a_int_1[ebp]
	  pop edx
	  cmp ecx, edx
	  jge CMPNE_14483
	  push 1
	  jmp CMPEND_14483
	CMPNE_14483:
	  push 0
	CMPEND_14483:

Окончание таблицы 4

Go	MASM32

	cmp eax, 0
	je LOOPEND_22386
	push i_int_2[ebp]
	pop ecx
	push 5
	pop edx
	cmp ecx, edx
	jle CMPNE_30620
	push 1
	jmp CMPEND_30620
	CMPNE_30620:
	push 0
	CMPEND_30620:
	pop eax
	cmp eax, 0
	je IFELSE_272570
	jmp LOOPEND_22386
	jmp IFEND_27257
	IFELSE_272570:
	IFEND_27257:
	LOOPINC_22386:
	push i_int_2[ebp]
	pop eax
	inc eax
	mov i_int_2[ebp], eax
	jmp LOOPSTART_22386
	LOOPEND_22386:
	leave
	ret
	leave
	ret
	text ends
	end __start

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.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. The Go Programming Language Specification, - URL: https://golang.org/ref/ (дата обращения 2020-06-28)

2. Golang: Основы, - URL: https://tproger.ru/translations/golang-basics/ (дата обращения 2020-06-15)

3. Руководство по ассемблеру x86 для начинающих, - URL: https://habr.com/ru/post/423077/ (дата обращения 2020-06-25)

4. x86 Instruction Set Reference, - URL: https://c9x.me/x86/ (дата обращения 2020-06-23)

5. Вирт Н. Построение компиляторов - М., 2010. – 192с

ПРИЛОЖЕНИЕ А

Текст программы на прилагаемом компакт-диске.