Министерство науки и высшего образования Российской Федерации

ФГБОУ ВО «АЛТАЙСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Институт цифровых технологий, электроники и физики (ИЦТЭФ)

Кафедра вычислительной техники и электроники (ВТиЭ)

Отчет по производственной практике

(Научно-исследовательская работа)

РАЗРАБОТКА ЯЗЫКА ПРОГРАММИРОВАНИЯ:

КОМПИЛЯТОР И ВИРТУАЛЬНАЯ МАШИНА

Выполнил студент 506 группы:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Д.В. Осипенко

«\_\_\_»\_\_\_\_\_\_\_\_\_\_\_ 2024 г.

Проверил: ст. преп. кафедры ВТиЭ.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ П.Н. Уланов

«\_\_\_»\_\_\_\_\_\_\_\_\_\_\_ 2024 г.

РЕФЕРАТ

Полный объем работы составляет 14 страниц, включая 7 рисунков.

В рамках научно-исследовательской работы была произведена разработка простого компилятора, состоящего из: лексического, синтаксического анализаторов и кодогенератора. Целью компиляции является также разработанная байт кодовая виртуальная машина, поддерживающая базовые типы данных (целочисленные, с плавающей точкой, строки, булевы значения).

Цель работы – разработать компилятор для простого языка и языковую виртуальную машину.

В результате выполнения научно-исследовательской работы были разработаны компилятор и стековая байт кодовая виртуальная машина.

Ключевые слова: Go, компилятор, кодогенератор, виртуальная машина, программное обеспечение.

СОДЕРЖАНИЕ

[Введение 4](#_539pho8uv2vx)

[1. Разработка компилятора 5](#_lrojk42a2zqt)

[1.1. Краткая теория 5](#_bsknec3hmxzr)

[1.2. Проектирование 5](#_8r3jf6jywcoi)

[1.3. Реализация 6](#_sfsdnfv6quym)

[2. Разработка виртуальной машины 9](#_cvuo9myha603)

[2.1. Краткая теория 9](#_d5s8gxz8vzog)

[2.2. Проектирование 9](#_oldd4p2dii76)

[2.3. Реализация 11](#_yff3es88ut8l)

[Заключение 13](#_diutxh2gbnyc)

[Список использованной литературы 14](#_wgjbpd8v0rn5)

# Введение

На сегодняшний день любая программа написана с помощью какого-либо языка программирования (далее ЯП), от самого низкоуровневого, например ассемблер, до высокоуровневых представителей, таких как: Python, Ruby, Go, Rust, C/C++ и т.д.. Сам по себе ЯП представляет собой [1]:

● Спецификацию;

● Программу компилятор или интерпретатор.

Компилятор (или интерпретатор) являются главным элементом любого ЯП, определяющее основные показатели языка, как: быстродействие, поддерживаемые платформы и функциональные возможности. В следствии чего это направление можно назвать одним из основополагающих и до сих пор актуальных [2].

В данной научно-исследовательской работе (далее НИР) объектом исследования является компилятор и виртуальная машина.

Цель работы – разработать компилятор для простого языка и языковую виртуальную машину.

Задачи НИР:

1. Разработать компилятор для простого языка;

2. Разработать стековую виртуальную машину;

3. Протестировать работоспособность созданных компилятора и виртуальной машины.

# 

# Разработка компилятора

## Краткая теория

**Компилятор** – это программа, которая на вход принимает исходный код на определенном языке, проводит различные преобразования и оптимизации, затем на выходе генерирует программу на некотором целевом языке или исполняемый код под определенную архитектуру [3].

По своему устройству компилятор можно разделить на две части:

- frontend – занимается распознаванием и анализом поступающего на вход текста программа, созданием синтаксического дерева, проверкой смысловой верности построенного дерева, и в качестве выхода возвращает некое промежуточное представление (чаще всего абстрактное синтаксическое дерево) и в некоторых случаях таблицу символов;

- backend – целью данной части является преобразование поступившего на вход представления в более низкоуровневые, проведение необходимых оптимизаций и как итог генерация целевого языка, будь то другой язык программирования или машинный код [4].



Рис. 1.1 Поверхностная структура типичного компилятора.

## Проектирование

Язык, для которого был разработан компилятор, довольно прост, содержит в себе всего 2 примитивных типа данных (целочисленные и числа с плавающей точкой), возможность создавать функции, переменные, константы. Грамматика языка приведена в листинге 1.1.

**Листинг 1.1:** Грамматика языка.

ProgramAST = Global+ .

Global = FunctionAST | VaribleAST .

FunctionAST = "@fn" <symbol> "[" SymbolDeclAST\* "]" ("<" DataType+ ">")? "{" Local+ "}" .

SymbolDeclAST = <symbol> ":" DataType .

DataType = SimpleDataTypeAST .

SimpleDataTypeAST = <symbol> .

Local = IfAST | ReturnAST | SymbolExpressionAST | VaribleAST | SetAST .

IfAST = "@if" Expression "{" Local+ "}" ("else" "{" Local+ "}")? .

Expression = SymbolExpressionAST | SymbolAST | IntegerAST | FloatAST .

SymbolExpressionAST = "(" <symbol> Expression\* ")" .

SymbolAST = <symbol> .

IntegerAST = <integer> .

FloatAST = <float> .

ReturnAST = "@return" Expression? ";" .

VaribleAST = "@var" SymbolDeclAST "=" Expression ";" .

SetAST = "@set" <symbol> "=" Expression ";" .

Говоря о структуре созданного компилятора (рис. 1.2), frontend включает в себя 2 этапа:

- Лексический анализ (Lexer) – разбор входного текста программы на понимаемые компилятором и предопределенные в спецификации слова (токены).

- Синтаксический анализ (Parser) – группировка поступающих на вход токенов по смысловым группам (выражения, определения, инструкции), основываясь на их последовательности. Также данный этап включает в себя семантический анализ (за исключением проверки типов для выражений), ввиду однозначности грамматики языка.

Backend, в свою очередь, состоит всего из одного этапа – генерация байт кода для для стековой виртуальной машины.

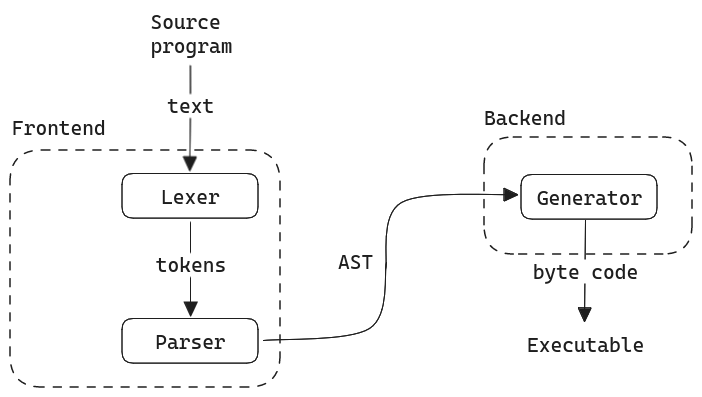


Рис. 1.2 Структура созданного компилятора.

## Реализация

Разработка компилятора была произведена на языке Go (Golang), языка общего назначения, созданного компанией Google для системного программирования и облачных технологий. Данный язык имеет строгую статическую типизацию, простой синтаксис, быструю нативную компиляцию и встроенный сборщик мусора. Данный набор особенностей позволяет создавать быстродействующие и эффективные программы с наименьшей затратой времени [5].

Лексический и синтаксический анализаторы были созданы с помощью сторонней библиотеки **participle** [6]. Данная библиотека позволяет быстро и без каких-либо трудностей создать парсер. Единственное что необходимо было сделать, это создать элементы АСД и их связи, все остальное (разбор и анализ входного кода) библиотека выполнит сама. Примеры работы парсера и лексера приведены на рисунках 1.3.

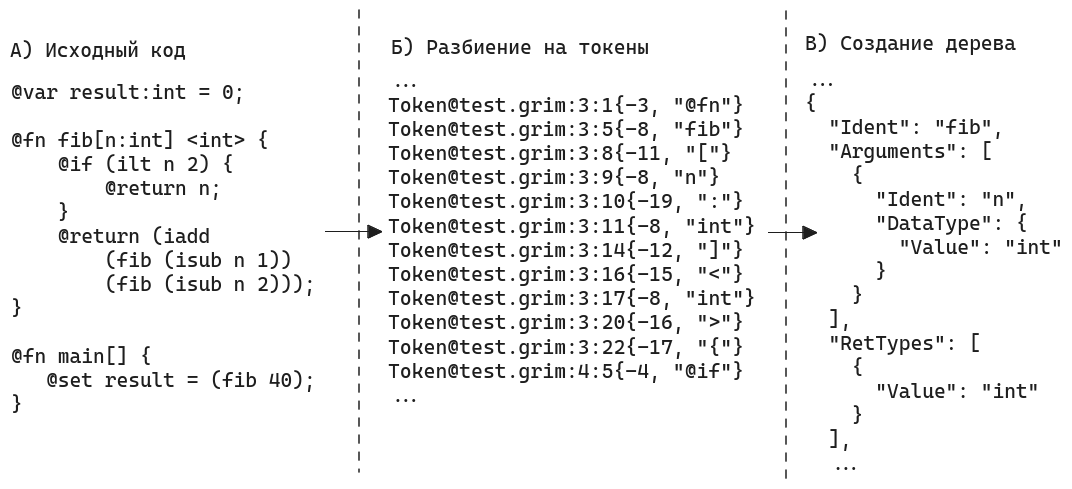


Рис. 1.3 Результат работы лексера (Б) и парсера (В).

Генератор представлен структурой Generator, объявление которой приведено в листинге 1.2. Поле Program служит для хранения генерируемой программы в общем. Поле Function используется для хранения каждой конкретной функции, с последующем сохранением в поле Program. Данная структура имеет методы:

- GenerateProgram – используется для генерации программы целиком, проходя по всем глобальным конструкциям языка и вызывая соответствующие им методы;

- GenerateDataType – используется для генерации типа данных;

- GenerateDefinition – используется для генерации определений функций и переменных;

- GenerateFunction – используется для генерации тела функции, проходя по всем локальным конструкциям;

- GenerateLocal – используется для генерации кода, соответствующего каждой отдельной локальной конструкции;

- GenerateExpression – используется для генерации кода, соответствующего каждому отдельному выражению;

**Листинг 1.2:** Объявление структуры Generate и ее методов.

type Generator struct {

Program ir.Program

Function ir.Function

}

func (g \*Generator) GenerateProgram(program ProgramAST) (ir.Program, error)

func (g \*Generator) GenerateDataType(dt DataType) (ir.IDataType, error)

func (g \*Generator) GenerateDefenition(node AST) (ir.ISymbolDef, error)

func (g \*Generator) GenerateFunction(fn FunctionAST) (ir.Function, error)

func (g \*Generator) GenerateLocal(l Local) (\*ir.Code, error)

func (g \*Generator) GenerateExpression(expr Expression) (\*ir.Code, error)

Пример работы генератора приведен на рисунке 1.4. Можно заметить что программа работает верно.

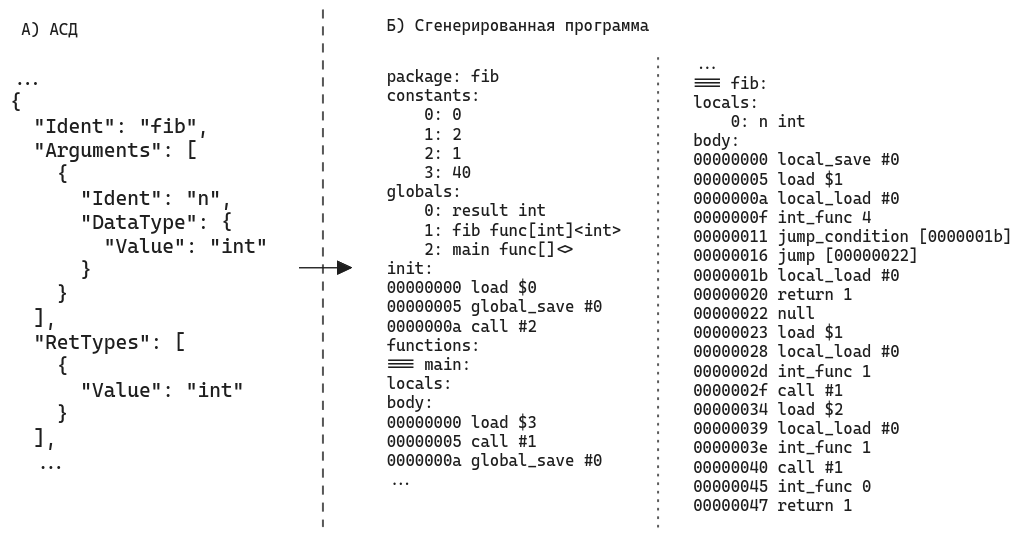


Рис. 1.4 Пример генерации из Абстрактного Синтаксического Дерева (А) в байт код (Б).

# Разработка виртуальной машины

## Краткая теория

Технологии виртуальных машин (далее ВМ) обычно выражаются в нескольких сферах:

- Операционные системы;

- Языки программирования и компиляторы;

- Архитектура компьютера.

Связанно это с тем, что данные подходы позволяют открыть новые возможности в решении различных задач и проблем в плане взаимодействия компонентов компьютерных систем. В области операционных систем, ВМ получают новый всплеск развития после некоторых лет относительно небольшой активности, потому что они позволяют эффективно распределять ресурсы машины, при этом сохраняя высокую степень безопасности. Виртуализация становится популярной для серверов и других сетевых приложений, особенно где безопасность является одним из основополагающим фактором. В сфере языков программирования (о чем пойдет речь дальше), ВМ позволяют разрабатывать платформонезависимый код, и помимо этого, дают возможность реализовать более динамичную трансляцию и оптимизацию кода. В процессорной архитектуре, ВМ открывают двери для новых наборов инструкций, которые могут привести к увеличению производительности, уменьшению энергопотребления и осуществлению динамической оптимизации [7].

Разрабатывая компилятор, наш выбор падает на языковые ВМ, которые делятся по принципам основных способов хранения промежуточной информации и передачи аргументов и результатов работы функции. Из этих принципов выделяют два типа ВМ:

- Стековая – в качестве способа хранения и передачи данных используется стек. Проста в реализации, более простое поддержание кроссплатформенности, но уступает в производительности регистровой ВМ. Представители: JVM, Forth, P-code;

- Регистровая – в качестве способа хранения и передачи данных выступают глобальные регистры, количество которых ограничено только реализацией. Имеет большее быстродействия, по сравнению с стековой ВМ, но более трудную реализацию. Примеры: Parrot VM, Dalvik, Lua VM, LLVM bitcode.[8]

## Проектирование

В рамках данной работы была создана виртуальная машина для языка программирования, компилятор которого описан ранее. Для реализации была выбрана стековая ВМ, которая будет выполнена в виде байт кодового интерпретатора (инструкции и их аргументы закодированы в списки байтов)[9]. Она поддерживает 4 базовых типа данных: целые числа, числа с плавающей точкой, строки и булевы значения. Имеет два стека: для хранения и передачи значений; для хранения фрейма вызова функции. Фрейм, в свою очередь, имеет список локальных регистров. Графическое представление ВМ приведено на рисунке 2.1.

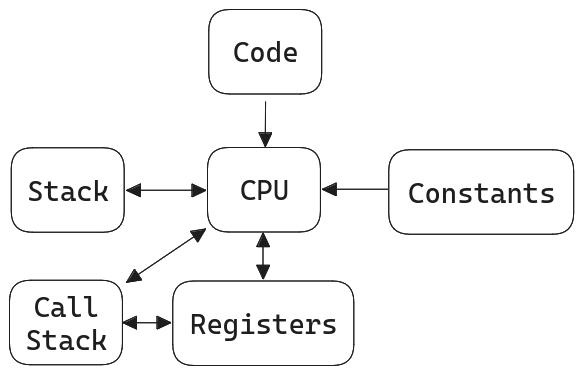


Рис. 2.1 Примерная структура ВМ и взаимодействие ее компонентов.

Программа для данной ВМ (рис. 2.2) содержит в себе:

- название программы

- список констант, обращение к которым происходит по их индексу;

- список глобальных символов (переменных, функций), доступ к которым осуществляется по их индексу;

- блок init – который отвечает за инициализацию глобальных переменных и вызов функции main;

- блоки функций, которые содержат в себе наборы инструкций и локальные переменные.



Рис. 2.2 Пример программы для данной ВМ.

## Реализация

Обычно для реализации данного рода ПО, требующего высокую производительность и возможный доступ к низкоуровневому интерфейсу системы принято использовать низкоуровневые языки для системного программирования, наподобие С/С++[10]. Но для простоты работы и возможности осуществления более высокой интеграции с ранее написанным компилятором, ВМ была разработана на языке Go.

ВМ реализована с помощью структуры VM, код которой приведен в листинге 2.1. Она содержит в себе поступающую на вход программу, список констант, стек, стек вызовов, и индекс текущего вызова. В качестве методов используются:

- MustExecute – входная точка в ВМ, которая осуществляет загрузку констант из программы, создание глобального фрейма, и запуск программы с блока init;

- GlobalFrame – возвращает глобальный (нулевой) фрейм;

- CurrentFrame – возвращает верхушку стека вызовов;

- PushFrame – заталкивает на верхушку стека вызовов новый фрейм

- PopFrame – выталкивает из стека вызовов его верхушку;

- run – выполняет код текущего фрейма.

**Листинг 2.1:** Объявление структуры VM и ее методов.

type VM struct {

Program \*ir.Program

Constants []Object

Stack Stack

Frames [512]Frame

frameIndex int

}

func (vm \*VM) MustExecute(program \*ir.Program)

func (vm \*VM) GlobalFrame() \*Frame

func (vm \*VM) CurrentFrame() \*Frame

func (vm \*VM) PushFrame(frame Frame)

func (vm \*VM) PopFrame()

func (vm \*VM) run()

Пример работы программы (рис. 2.3), приведенной на рисунке 2.2. Можно заметить что программа работает верно.

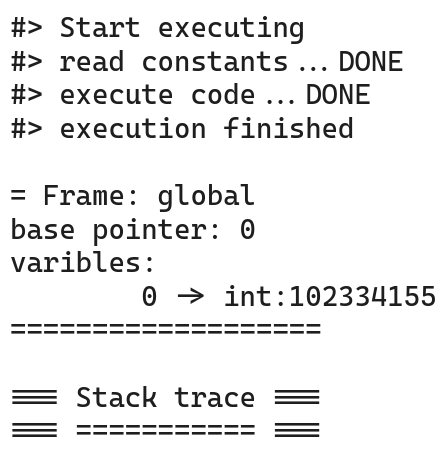


Рис. 2.3 Пример выполнения программы.

# Заключение

В ходе данной НИР были выполнены все поставленные задачи. В частности:

1. Разработать компилятор для простого языка;

2. Разработать стековую виртуальную машину;

3. Протестировать работоспособность созданных компилятора и виртуальной машины.

Дополнительно получены навыки разработки и проектирования комплексных и многомодульных программных систем, использования системы контроля версий.

По итогам работы цель “разработать компилятор для простого языка и стековую виртуальную машину” была достигнута.

Весь исходный код расположен в открытом доступе на Github репозитории по адресу: <https://github.com/SpectralJager/grimlang>

## 

# Список использованной литературы

1. Modern Compiler Design: 2nd ed. /Dick Grune, Kees van Reeuwijk, Henri E. Bal, Ceriel J. H. Jacobs, Koen Langendoen. – New York: Publisher Springer, 2012.- 843 p.
2. Torben Ægidius Mogensen*.* Basics of Compiler Design. – Copenhagen: Publisher lulu.com, 2010.-319 p.
3. Terence Parr. Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages: 1st ed. – Raleigh: Publisher Pragmatic Bookshelf, 2010. 389 p.
4. Engineering of compiler: 2nd ed. /Keith D. Cooper, Linda Torczon. – Burlington: Publisher Morgan Kaufmann, 2011.-824 p.
5. The Go Programming Language [Электронный ресурс]. – Режим доступа: <https://go.dev/>, свободный. – Загл. с экрана. - Яз. англ.
6. A parser library for Go - GitHub [Электронный ресурс]. – Режим доступа: [https://github.com/ alecthomas/participle](https://github.com/alecthomas/participle), свободный. – Загл. с экрана. - Яз. англ.
7. Virtual Machines: Versatile platforms for systems and processes: 1st ed. /James E. Smith, Ravi Nair. – San Francisco: Publisher Morgan Kaufmann, 2005.-660 p.
8. Два мира виртуальных машин [Электронный ресурс]. – Режим доступа: <https://habr.com/en/companies/intel/articles/254793/>, свободный. – Загл. с экрана. - Яз. рус.
9. A Bytecode Virtual Machine – Crafting Interpreters [Электронный ресурс] . – Режим доступа: <https://craftinginterpreters.com/a-bytecode-virtual-machine.html>, свободный. – Загл. с экрана. – Яз. англ.
10. Andrew W. Appel. Modern compiler implementation in C. – Cambridge: Publisher Cambridge University Press, 1998.-544 p.