Министерство науки и высшего образования Российской Федерации

**Федеральное государственное бюджетное образовательное учреждение**

**высшего образования**

**«Владимирский государственный университет**

**имени Александра Григорьевича и Николая Григорьевича Столетовых»**

**(ВлГУ)**

разработка компилятора подмножества

процедурно-ориентированного языка

Пояснительная записка

RU. 643.02068048.0001-01 81 01

На 15 листах

|  |  |  |
| --- | --- | --- |
| Руководитель |  | к.т.н. доцент кафедры ИЗИ  Монахов Ю.М. |
|  |
| Исполнитель |  | студентка гр. ИБ-117  Ильиных Т.А. |

**Владимир 2020**

# АННОТАЦИЯ

В данном документе приведена работа по созданию компилятора собственного языка программирования.  Грамматика разработанного языка представляет совмещение двух языков программирования Java и C. Использовался генератор анализаторов для формальных языков ANTLR. Основная задача компилятора – преобразовать исходный код программы в байт код JAVA машины и запустить его на исполнение.

Разработка компилятора подмножества процедурного языка в ассемблер состоит из следующих стадий:

1) построение лексического анализатора;

2) построение синтаксического анализатора;

3) построение генератора объектного кода;

4) построение оптимизатора объектного кода.

**Оглавление**

[**АННОТАЦИЯ 2**](#_Toc42105405)

[**1 ПРОЕКТИРОВАНИЕ КОМПИЛЯТОРА 4**](#_Toc42105406)

[**1.1 Основные требования 4**](#_Toc42105407)

[**1.2. Создание лексического анализатора 5**](#_Toc42105408)

[**1.3. Разработка синтаксического анализатора 7**](#_Toc42105409)

[**1.4 Построение генератора объектного кода 8**](#_Toc42105410)

[**Синтаксическое дерево 8**](#_Toc42105411)

[**Генерация кода 9**](#_Toc42105412)

[**1.5. Оптимизатор 10**](#_Toc42105413)

[**2 ПРОВЕРКА НА СООТВЕТСТВИЕ ОСНОВНЫМ ТРЕБОВАНИЯМ 11**](#_Toc42105414)

# 1 ПРОЕКТИРОВАНИЕ КОМПИЛЯТОРА

## Основные требования

Разработка будет производиться в соответствии со следующими требованиями:

* Требования к входному языку:

1. Должны присутствовать операторные скобки;
2. Должна игнорироваться индентация программы;
3. Должны поддерживаться комментарии любой длины;
4. Входная программа должна представлять собой единый модуль, но поддерживать вызов функций.

* Требования к операторам:

1. Оператор присваивания;
2. Арифметические операторы;
3. Логические операторы (И, ИЛИ, НЕ);
4. Условный оператор (ЕСЛИ);
5. Оператор цикла (while, break, continue);
6. Базовый вывод (строковой литерал, переменная);
7. Типы (целочисленный, вещественный).

Для реализации некоторых функций моей работы были использованы сторонние библиотеки

* org.antlr:antlr4:4.8 （для лексического и грамматического анализа）
* org.ow2.asm:asm:8.0.1 （используется для создания байт-кода JVM）

Структурная схема проекта представлена на рисунке 1



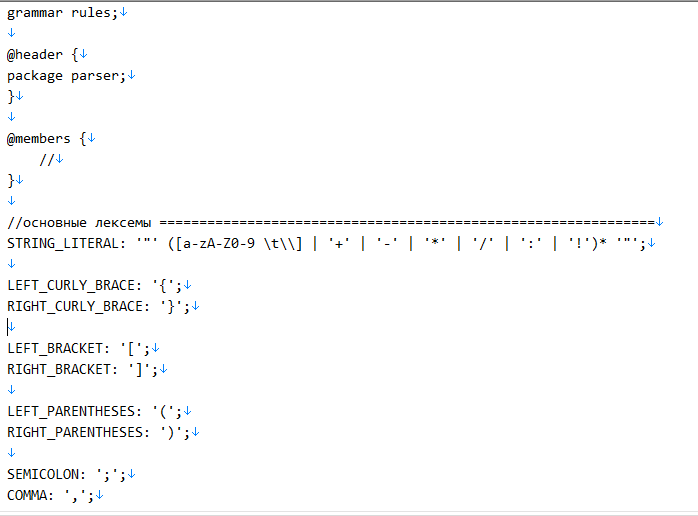
Рисунок 1

## 1.2. Создание лексического анализатора

Лексический анализатор является первым этапом работы компилятора. Фаза лексического анализа отвечает за разделение входной строки (или потока символов) программы на более мелкие фрагменты, называемые токенами . Токены несут информацию об их типе (если это числа, операторы, ключевые слова, идентификаторы и т. Д.), Подстроке программы, которую они представляют, и их положении в программе.

Грамматика синтаксического и лексического анализатора основаны на грамматике языка java и C с некоторыми изменениями. Лексический и синтаксический анализатор разработаны с помощью Antlr 4 под Java.

На рисунке 2 приведен список регулярных выражения для описания лексем языка.



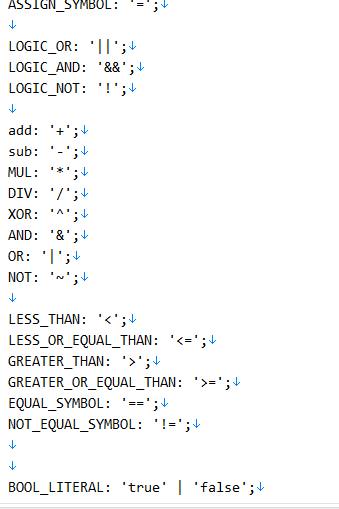


Рисунок 2 Регулярные выражения для описания лексем

## 1.3. Разработка синтаксического анализатора

Вторым этапом компилятора является синтаксический анализ. На вход синтаксическому анализатору подаётся набор токенов из лексического анализатора. На основе грамматики языка (рисунок 3) строится дерево разбора грамматики.

Дерево разбора или синтаксическое дерево (Parse Tree) — Структура, представляющая собой результат работы синтаксического анализатора. Она отражает синтаксис конструкций входного языка и явно содержит в себе полную взаимосвязь операций.

Абстрактное синтаксическое дерево (AST — Abstract Syntax Tree) отличается от дерева разбора тем, что в нём отсутствуют узлы и рёбра для тех синтаксических правил, которые не влияют на семантику программы.

ANTLR использует шаблон Visitor для предоставления доступа к дереву разбора и может рекурсивно обходить дерево, генерируя синтаксическое дерево и добавляя атрибуты. Через наследование XXXBaseVisitor<...> для получения доступа к дереву разбора. Исходный код которого можно посмотреть на <https://github.com/Tanya339916/kursovik>

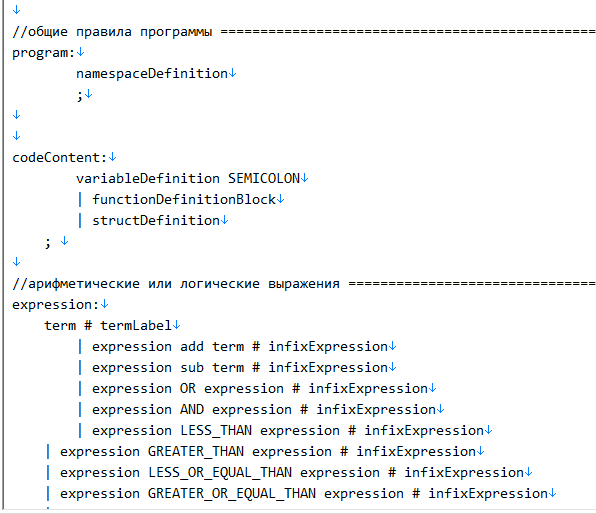


Рисунок 3 Грамматика \

## 1.4 Построение генератора объектного кода

Дерево обхода начинается с корневого узла и постепенно спускается, но оно не является строго обходом глубины, поскольку не все атрибуты могут быть получены путем обхода глубины. В процессе обхода дерева другим XXXBaseVisitor<...> подклассом SymbolTableGenerator информация о символах получается из текущего контекста.

Узлы синтаксического дерева можно условно разделить на два типа: один - это узел, в котором хранится фактическая информация, например, узел, в котором хранится предложение, узел, в котором хранится переменная, а другой - узел, который задает структуру, например, узел, который задает структуру цикла и структуру логического суждения. Эти узлы сами по себе не несут информацию, но она используется в качестве организатора дочерних узлов для указания появления новых структур. Узлы синтаксического дерева являются Node типами подклассов. Все дочерние узлы каждого узла хранятся в List структуре. Node. Далее делятся на подклассы, такие как Definition Node, RefNode другие подклассы. Основная информация, хранящаяся в узле, - это информация о типе (если она соответствует сущности), информация о значении (если она буквальная), справочная информация (если это ссылка на имя) и т. д.

### Синтаксическое дерево

Иллюстрация синтаксического дерева, созданного в предыдущем примере, выглядит следующим образом (рисунок 4):

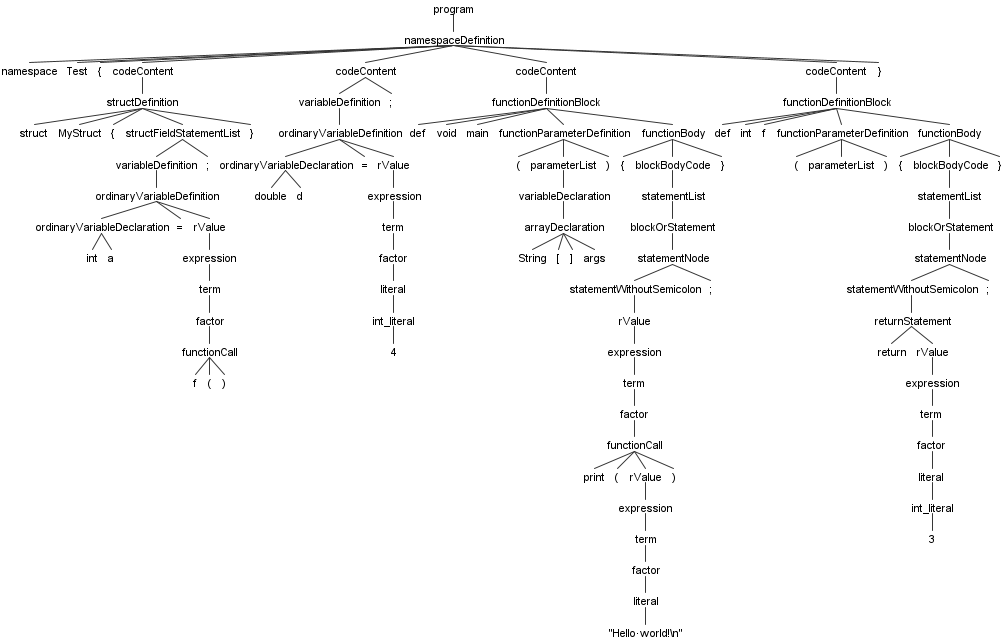


Рисунок 4 Синтаксическое дерево

### Генерация кода

Настоящий компилятор преобразует исходный код в окончательный байт-код JVM, то есть Byte[]массив, массив и записывает файл, файл может быть запущен JVM. Для генерации кода использует стороннюю библиотеку Java ASM.

Java ASM предоставляет ClassVisitor, MethodVisitor например, интерфейсы, чтобы мы могли легко генерировать код сборки JVM. Промежуточным кодом этого компилятора является байт-код JVM. Атрибуты автоматически заполняется сторонней библиотекой ASM.

## 1.5. Оптимизатор

Основная оптимизация этого проекта произошла на этапе генерации кода. Для той же семантики JVM предоставляет несколько инструкций, а компилятор оптимизирует при выборе инструкций.

# ПРОВЕРКА НА СООТВЕТСТВИЕ ОСНОВНЫМ ТРЕБОВАНИЯМ

При проектировании компилятора к основному языку были установлены следующие минимальные требования: наличие операторных скобок, игнорирование пробелов и идентации программы, поддержка многострочных комментариев и вызова функций. Наличие операторов присваивания, условных, цикла, break-continue, арифметических, логических. Должны присутствовать два типа данных. И выходная программа должна быть байт код на JAVA Машине.

Далее приведено тестирование компилятора.

Рисунки 5-7 – Присвоение строк

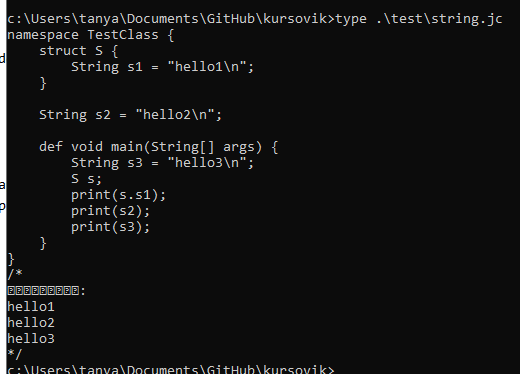


Рисунок 5 Присвоение строк

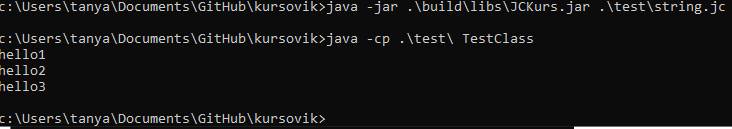


Рисунок 6 Результат выполнения программы

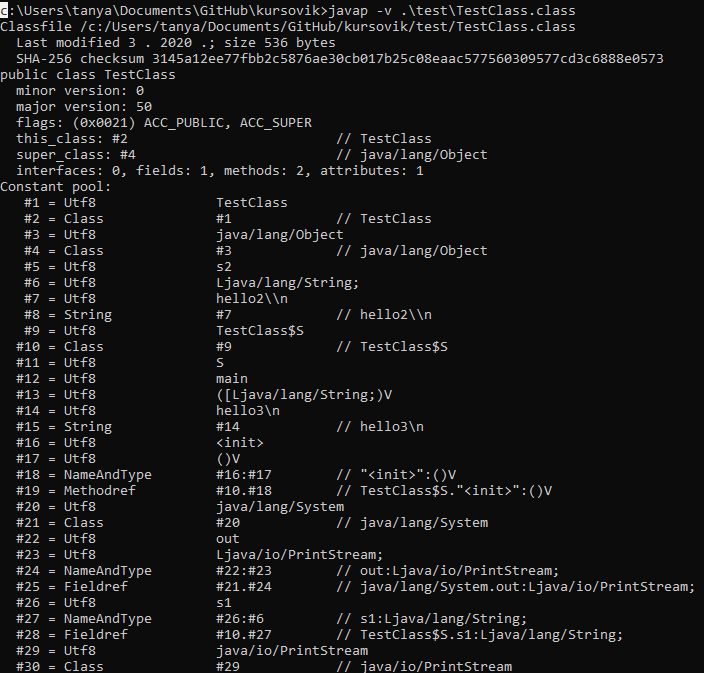


Рисунок 7 Код на языке JVM

Рисунки 8-9 – проверка условного оператора

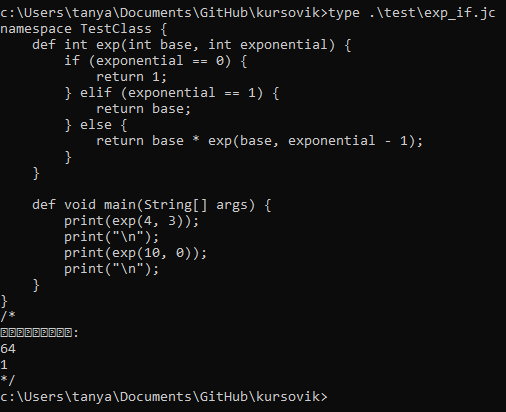


Рисунок 8 Код на исходном языке

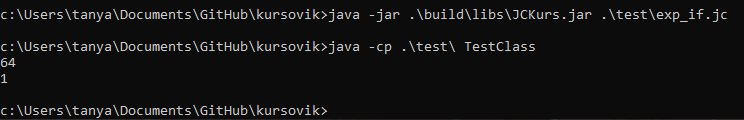


Рисунок 9 Результат выполнения программы

Рисунки 10-11 – проверка циклических выражений



Рисунок 10 Код на исходном языке

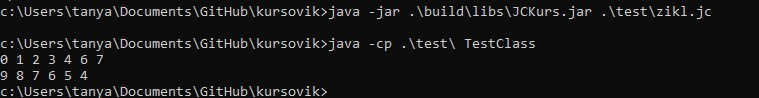


Рисунок 11 Результат выполнения программы

Рисунки 12-13 – тестирование арифметических выражений

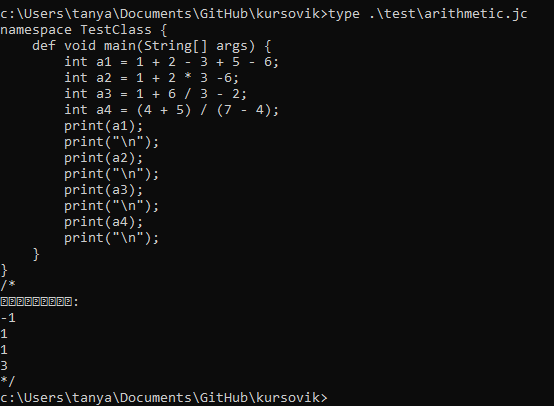


Рисунок 12 Исходный код

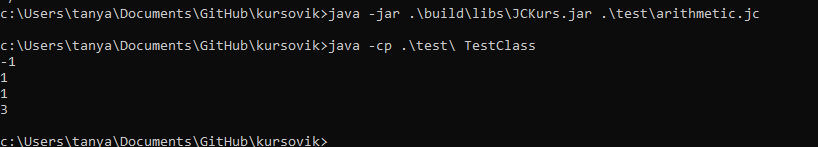


Рисунок 13 Результат выполнения программы

Исходные коды с кратким описание можно посмотреть: <https://github.com/Tanya339916/kursovik>