**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)**

**Институт информационных технологий и**

**радиоэлектроники**

**Кафедра информатики и защиты информации**

Компилятор подмножества процедурного языка в ассемблер

Пояснительная записка

#### АННОТАЦИЯ

В данном документе приведён код компилятора подмножества процедурного языка. Код программы реализован на языке программирования JAVA (JDK 15).

Разработанная программа состоит из лексического анализатора, синтаксического анализатора (parsing), таблица символов, генератора промежуточного кода, транслятора в целевой код.

Лексический анализатор считывает текстовый файл, разбирает содержимое на токены и, затем, направляет их синтаксическому анализатору.

В свою очередь синтаксический анализатор сопоставляет поток токенов с описанной грамматикой.

Генератор объектного кода переделывает полученную программу в последовательность инструкций, которые в дальнейшем могут выполняться на машине.

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 также должна быть поддержка вызова функций.

Операторы:

1. Оператор присваивания.
2. Арифметика (\*, /, +, -, >, <, =).
3. Логические операторы (И, ИЛИ, НЕ).
4. Условный оператор (ЕСЛИ).
5. Операторыцикла (while, break, continue).
6. Базовый вывод (строковый литерал, переменная).
7. Типы (целочисленный 32 бита, с плавающей запятой 32 бита).

Требования к выходному языку:

1. В ассемблере.

Оформление программного документа произведено по требованиям ЕСПД (ГОСТ 19.101-77 [[1]](#footnote-2), ГОСТ 19.103-77 [[2]](#footnote-3), ГОСТ 19.104-78\* [[3]](#footnote-4), ГОСТ 19.105-78\* [[4]](#footnote-5), ГОСТ 19.106-78\* [[5]](#footnote-6), ГОСТ 19.401-78 [[6]](#footnote-7), ГОСТ 19.604-78\* [[7]](#footnote-8)).

**СОДЕРЖАНИЕ**

1. ЭТАПЫ РАЗРАБОТКИ КОМПИЛЯТОРА................................................................5

1.1. Разработка, описание языка..................................................................................5

1.2. Контекстно-свободная грамматика......................................................................6

1.3. Лексический анализатор........................................................................................7

1.4. Синтаксический анализатор..................................................................................9

1.5. Таблица символов.................................................................................................12

1.6. Генерация промежуточного кода........................................................................13

1.7. Трансляция в целевой код....................................................................................15

2. РАБОТОСПОСОБНОСТЬ КОМПИЛЯТОРА И ПРИМЕРЫ..................................18

СПИСОК ЛИТЕРАТУРЫ...............................................................................................21

ПРИЛОЖЕНИЯ...............................................................................................................22

Лист регистрации изменений.........................................................................................23

**1. ЭТАПЫ РАЗРАБОТКИ КОМПИЛЯТОРА**

**1.1. Разработка, описание языка**

Ключевые слова подмножества языка:

break; continue; while; double; int; if; else; private; public; static; class; return; void; new; true; false; null.

Существует несколько правил в синтаксисе языка, которые необходимо помнить об идентификаторе, и которые нужно было реализовать:

* Каждый идентификатор должен начинаться с «A» до «Z» или «a» до «z», «$» или «\_».
* После первого символа может иметь любую комбинацию символов.
* Ключевое слово не может быть использовано в качестве идентификатора.
* Самое главное — идентификатор чувствителен к регистру.

Идентификаторы — имена, используемые для классов, переменных и методов. Все компоненты требуют имена.

Логические операции — это элементы формального языка, позволяющие составлять правильные выражения на том языке путем объединения более простых. Если первое значение в операции «&» (логическое И) ложно, то второе значение не проверяется, в операции «||» (логическое ИЛИ) наоборот, если первое значение истинно, то второе не проверяется.

Таблица истинности для операций «&», «||»:

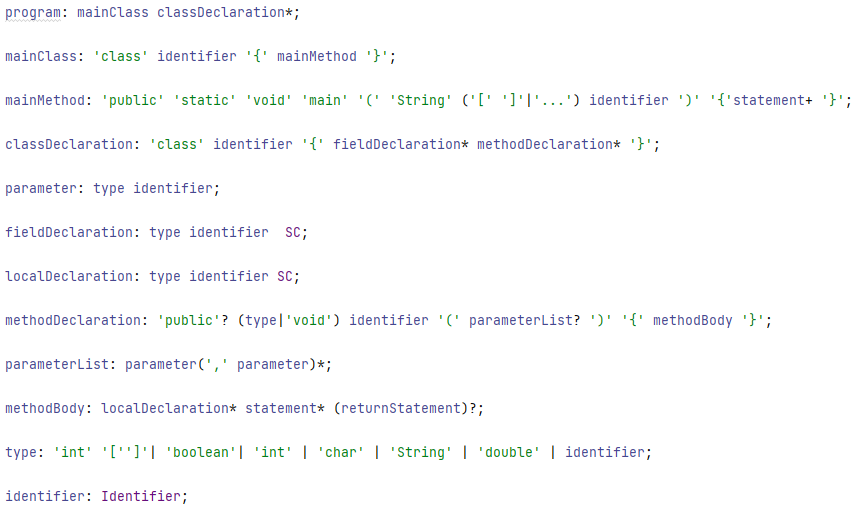
|  |  |  |  |
| --- | --- | --- | --- |
| **A** | **B** | **A & B** | **A | | B** |
| true | true | true | true |
| true | false | false | true |
| false | true | false | true |
| false | false | false | false |

**1.2. Контекстно-свободной грамматики**

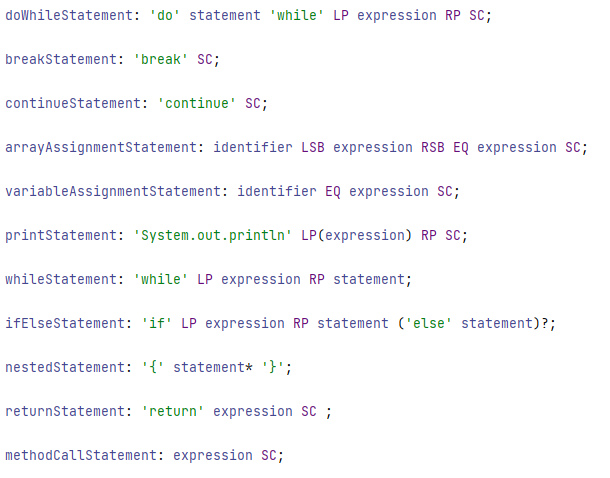
Контекстно-свободная грамматика (КС-грамматика, бесконтекстная грамматика) — частный случай формальной грамматики (тип 2 по иерархии Хомского), у которой левые части всех продукций являются одиночными нетерминалами (объектами, обозначающими какую-либо сущность языка и не имеющими конкретного символьного значения).

При разработке контекстно-свободной грамматики стоит понимать, что:

* каждое правило описывается, как нетерминальный символ, который может быть «заменен» или «расширен» строкой, состоящей из нетерминальных символов и терминальных символов;
* правила написано с синтаксисом, как регулярные выражения;
* правила можно применять рекурсивно;
* в конце концов должна получится строка, состоящая только из терминальных символов;
* конечная строка синтаксически верна в соответствии с грамматическими правилами;



**Рисунок 1.** Часть грамматики языка.



**Рисунок 2.** Часть грамматики языка.

**1.3. Лексический анализатор**

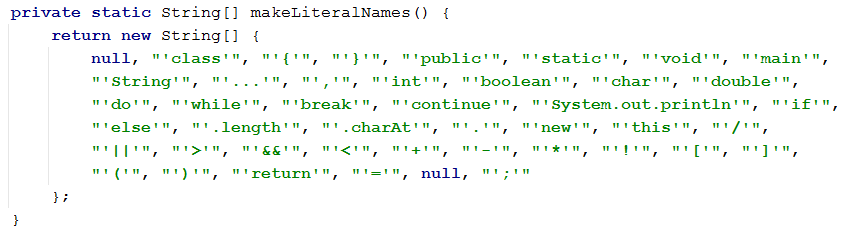
**Лексический анализ** **(«токенизация»)** — процесс аналитического разбора входной последовательности символов на распознанные группы — лексемы — с целью получения на выходе идентифицированных последовательностей, называемых «токенами».

Лексический анализатор реализован с помощью мощного инструмента ANTLR.

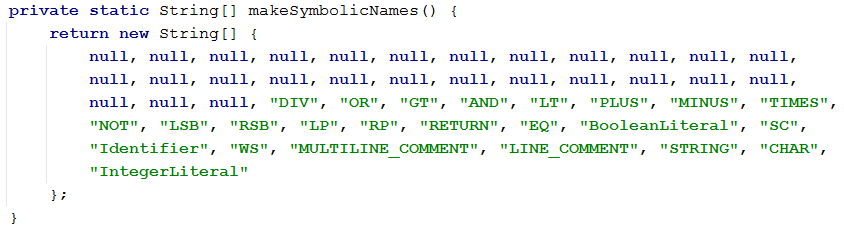
**ANTLR** — это мощный генератор парсеров для чтения, обработки, выполнения или перевода структурированных текстовых или двоичных файлов. Он широко используется для создания языков, инструментов и фреймворков. На основе грамматики ANTLR генерирует синтаксический анализатор, который может строить и обходить синтаксические деревья.

**Преимущества ANTLR:**

* Свободное программное обеспечение.
* Использование единой нотации для описания лексических и синтаксических анализаторов.
* Применение нисходящего, а не восходящего анализа.
* Удобство работы с абстрактным синтаксическим деревом.
* Предоставление сообщений об ошибках и восстановление после них.
* Наличие визуальных сред разработки (ANTLR Works, ANTLR Studio, плагинов к Eclipse и IntellliJ IDEA), которые позволяют создавать и отлаживать грамматики, поддерживают подсветку синтаксиса, автодополнение, визуальное отображание грамматик, строящееся в реальном времени по мере ввода, отладчик, рефакторинг.



**Рисунок 3.** Ключевые слова лексического анализатора.

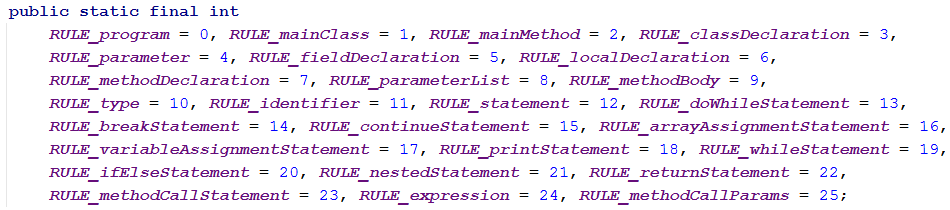


**Рисунок 4.** Символьные обозначения лексического анализатора.

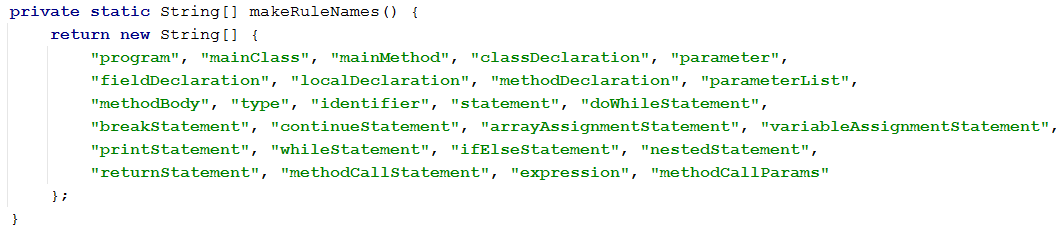
**1.4. Синтаксический анализатор**

Парсер, или синтаксический анализатор, — часть программы, преобразующей входные данные (как правило, текст) в некий структурированный формат, нужный для задач последующего их (данных) анализа и использования.

Из формального описания языка, называемого грамматикой, ANTLR генерирует синтаксический анализатор для этого языка, который может автоматически строить деревья синтаксического анализа, которые представляют собой структуры данных, представляющие, как грамматика соответствует входным данным. ANTLR также автоматически генерирует обходчики деревьев, которые можно использовать для посещения узлов этих деревьев для выполнения кода, специфичного для приложения.

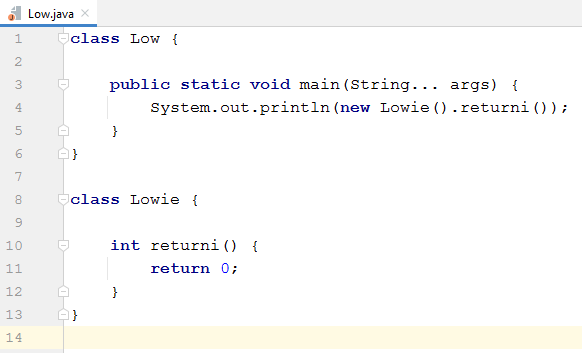


**Рисунок 5.** Правила языка.

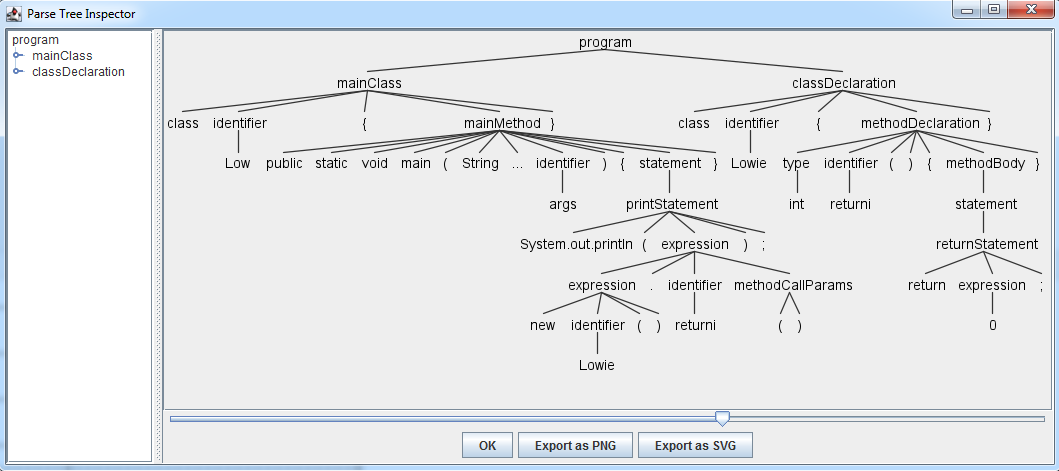


**Рисунок 6.** Название правил языка.

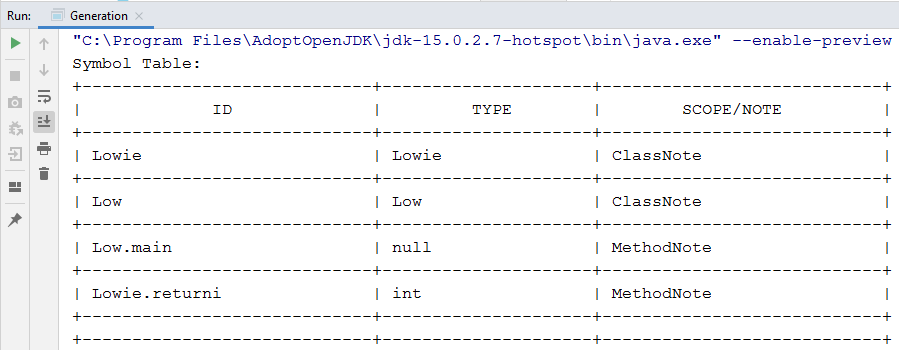
На рисунках 3 и 4 в разделе *«1.3. Лексический анализатор»* вы можете увидеть также ключевые слова лексического анализатора и символьные обозначения лексического анализатора, которые совпадают с синтаксическим анализатором.



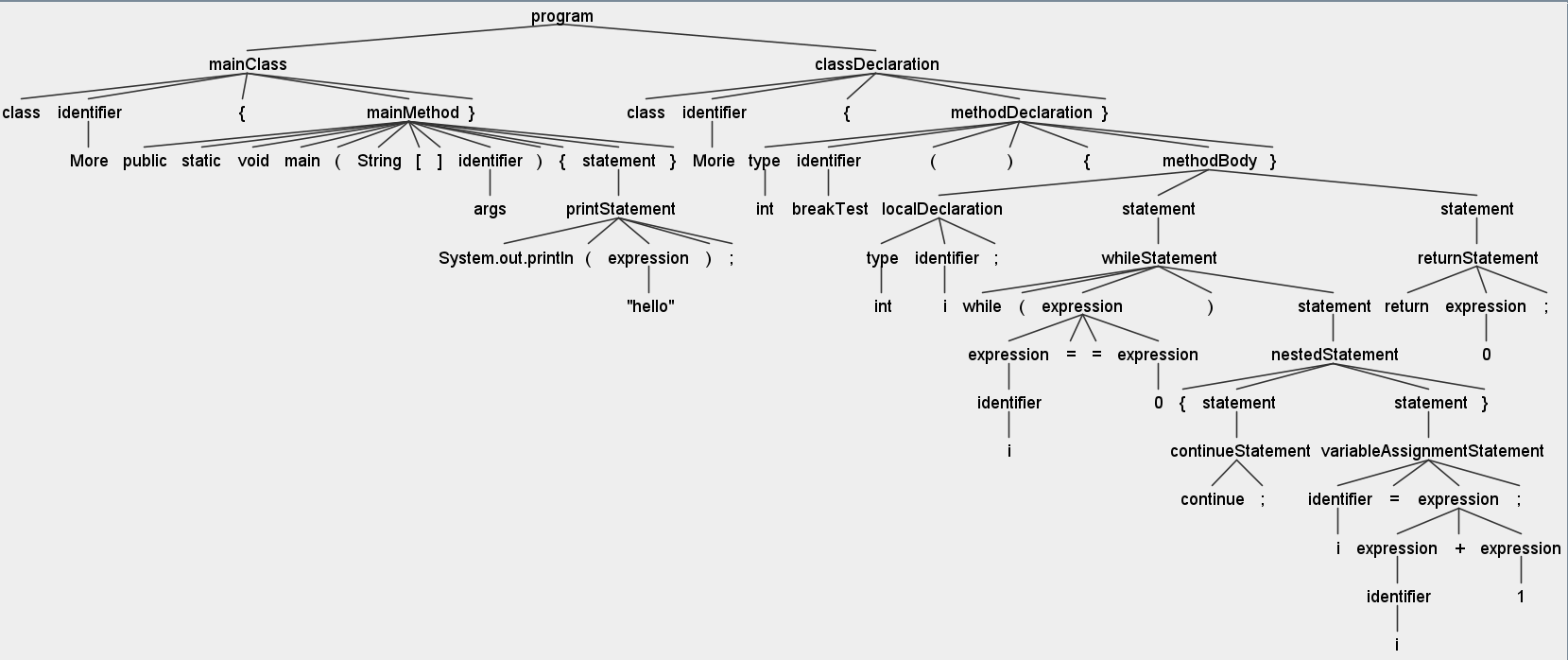
**Рисунок 7.** Пример входной программы Low.java для синтаксического анализатора.



**Рисунок 8.** Результат работа синтаксического анализатора (дерево).



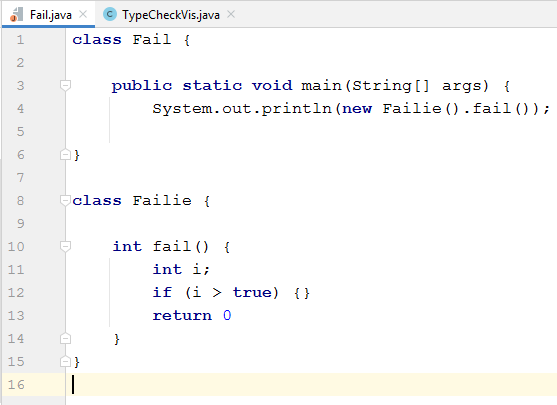
**Рисунок 9.** Результат работы синтаксического анализатора (таблица символов).



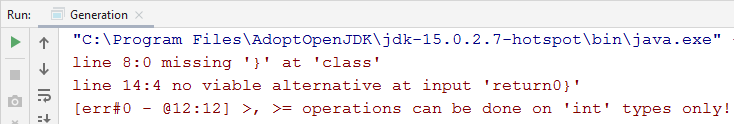
**Рисунок 10.** Результат работы синтаксического анализатора другой программы.

В компиляторе предусмотрен вывод ошибок (TypeCheckVisitor) для комфортного пользования программистом. Type Checking, который реализован в программе, охватывает достаточно большое количество ошибок: грамматические ошибки, неправильные сравнения, отсутствия return, неправильное использование expression, вызовы функций, область видимости, логические операции.

Например, на вход подается программа:



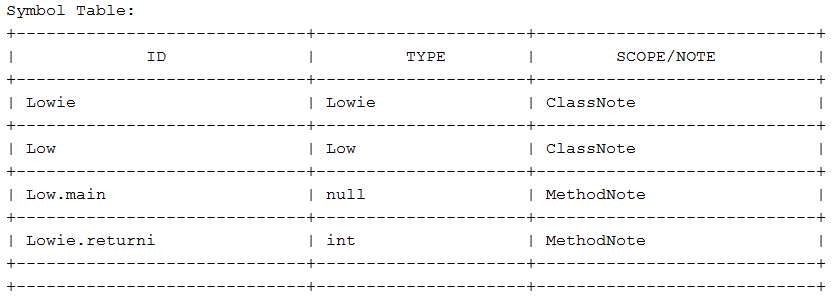
**Рисунок 11.** Программа с ошибками Fail.java.



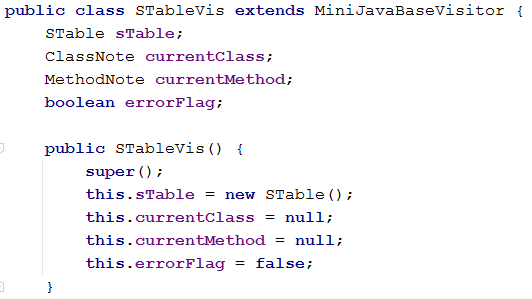
**Рисунок 12.** Работа класса TypeCheckVisitor.java.

**1.5. Таблица символов**

Таблица символов — это структура данных, используемая транслятором (компилятором или интерпретатором), в которой каждый идентификатор переменной или функции из исходного кода ассоциируется с информацией, связанной с его объявлением или появлением в коде: типом данных, областью видимости и в некоторых случаях местом в памяти (смещением).



**Рисунок 13.** Вывод символьной таблицы для файла Low.java (рисунок 7).



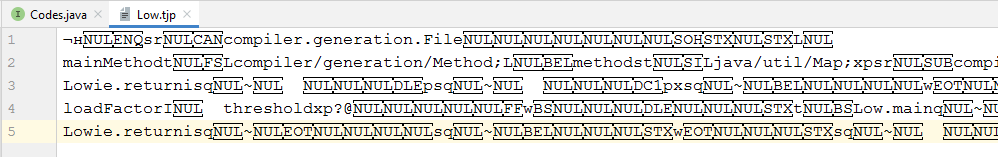
**Рисунок 14.** STableVis.java

**1.6. Генерация промежуточного кода**

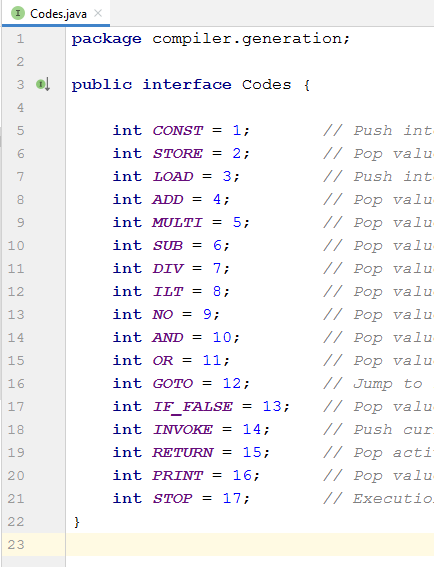
Кодогенерация — часть процесса компиляции, когда специальная часть компилятора, кодогенератор, конвертирует синтаксически корректную программу в последовательность инструкций, которые могут выполняться на машине. При этом могут применяться различные, в первую очередь машинно-зависимые оптимизации. Часто кодогенератор является общей частью для множества компиляторов. Каждый из них генерирует промежуточный код, который подаётся на вход кодогенератору.

Выходные данные из синтаксического анализатора передаются в транслятор, который продолжает обработку. Итогом работы транслятора является файл «название файла».tjp, в котором записывается промежуточный JAVA BYTECODE код.

Генератор промежуточного кода преобразует выходные данные синтаксического анализатора в JAVA BYTECODE , который в последствии будет обработан интерпретатором в JVM. Тем самым сохраняется правило Sun Microsystem «Write once, run everywhere».



**Рисунок 15.** Файл Low.tjp - выходные данные генератора промежуточного кода.



**Рисунок 16.** Инструкции для генерации промежуточного кода.

**1.7. Трансляция в целевой код.**

Машинный код — система команд (набор кодов операций) конкретной вычислительной машины, которая интерпретируется непосредственно процессором или микропрограммами этой вычислительной машины.

Каждая машинная инструкция выполняет определённое действие, такое как операция с данными (например, сложение или копирование машинного слова в регистре или в памяти) или переход к другому участку кода (изменение порядка исполнения; при этом переход может быть безусловным или условным, зависящим от результатов предыдущих инструкций). Любая исполнимая программа состоит из последовательности таких атомарных машинных операций.

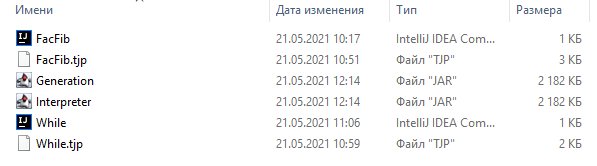
****

**Рисунок 17.** Часть машинных инструкций.

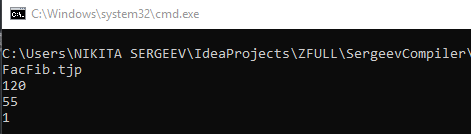
В компиляторе используется виртуальная машина Java (JVM) — это механизм, который обеспечивает среду выполнения для управления кодом Java или приложениями. Он преобразует байт-код Java в машинный язык.

Байт-код Java - это набор инструкций для виртуальной машины Java. Он действует аналогично ассемблеру, который является псевдонимом кода C++. Как только программа Java компилируется, создается байт-код Java. Набор инструкций для JVM может отличаться от системы к системе, но все могут интерпретировать байт-код. Следует иметь в виду, что байт-коды не являются исполняемыми кодами и зависят от доступности интерпретатора для выполнения, и поэтому в игру вступает JVM.

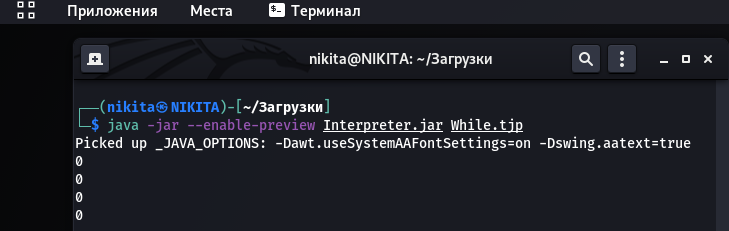
В компиляторе для удобства были использованы Java args, которые содержат предоставленные аргументы командной строки в виде массива String объектов. Также были созданы jar файлы, чтобы программисту, который будет писать код на данном языке, было проще запускать и отлаживать свою программу через командную строку.



**Рисунок 18.** Jar-файлы компилятора и программы, написанные на представленном языке.

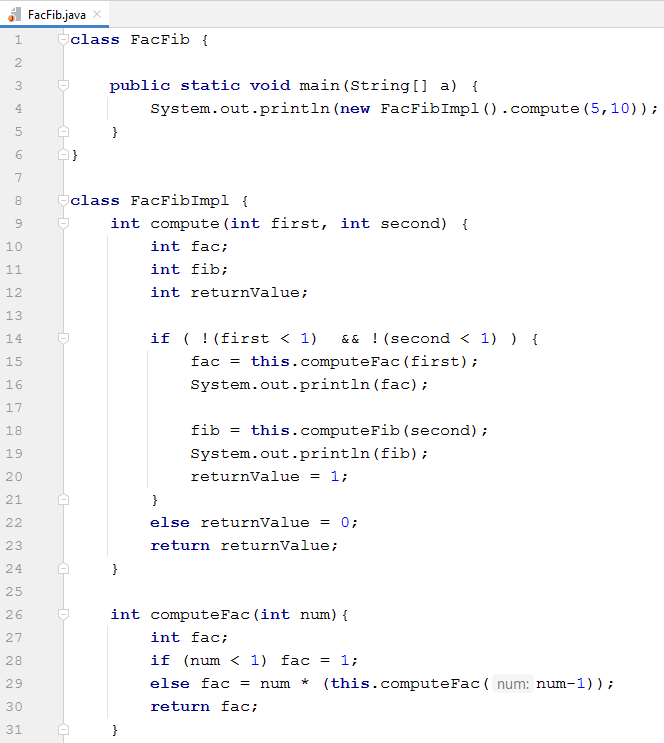


**Рисунок 19.** Запуск компилятора на Windows 10.

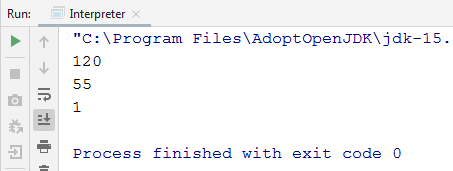


**Рисунок 20.** Запуск компилятора на Kali Linux.

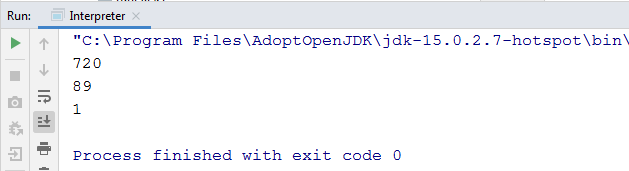
**2. РАБОТОСПОСОБНОСТЬ КОМПИЛЯТОРА И ПРИМЕРЫ**



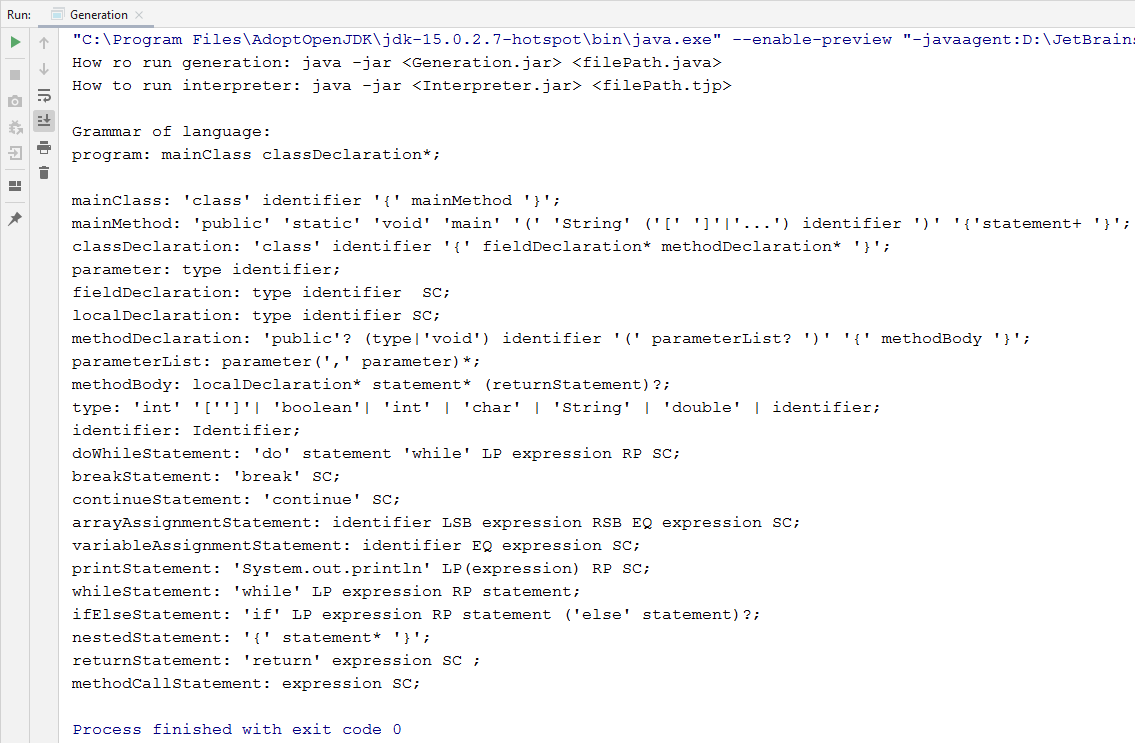
**Рисунок 2.1.** Программа, которая вычисляет факториал и Фибоначчи (FacFib.java).



**Рисунок 2.2.** Выполнение программы 2.1.



**Рисунок 2.3.** Выполнение программы 2.2 с другими данными (6, 11).



**Рисунок 2.4.** Запуск с аргументом --help.

**СПИСОК ЛИТЕРАТУРЫ**

**1.** Aho A. V., Sethi R., Ullman J. D. Compilers, principles, techniques //Addison wesley. – 1986.

**2.** Lindholm T. et al. The Java Virtual Machine Specification, Java SE 8 Edition. – Dian zi gong ye chu ban she, 2016.

**3.** Parr T. The definitive ANTLR 4 reference. – Pragmatic Bookshelf, 2013.

**4.** Chomsky N. Syntactic structures. – De Gruyter Mouton, 2009.

**5.** Hutton D. M. Clean Code: A Handbook of Agile Software Craftsmanship //Kybernetes. – 2009.

**ПРИЛОЖЕНИЯ**

**Исходный код программы (ссылка на github репозиторий):**

<https://github.com/Sergeevme/SergeevCompiler>



**Консольные команды:**

java -jar --enable-preview <Generation.jar> <filePath.java>

java -jar --enable-preview <Interpreter.jar> <filePath.tjp>

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Лист регистрацииизменений | | | | | | | | | |
| Номералистов (страниц) | | | | | Всего  листов  (страниц)  вдокум | №  документа | Входящий  № сопрово  дительного  документа  идата | Подп. | Дата |
| Изм | изменен  ных | заме  ненных | новых | Анулиро  ванных |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |

1. ГОСТ 19.101-77 ЕСПД. Виды программ и программных документов [↑](#footnote-ref-2)
2. ГОСТ 19.103-77 ЕСПД. Обозначение программ и программных документов [↑](#footnote-ref-3)
3. ГОСТ 19.104-78\* ЕСПД. Основные надписи [↑](#footnote-ref-4)
4. ГОСТ 19.105-78\* ЕСПД. Общие требования к программным документам [↑](#footnote-ref-5)
5. ГОСТ 19.106-78\* ЕСПД. Общие требования к программным документам, выполненным печатным способом [↑](#footnote-ref-6)
6. ГОСТ 19.401-78 ЕСПД. Текст программы. Требования к содержанию и оформлению [↑](#footnote-ref-7)
7. ГОСТ 19.604-78\* ЕСПД. Правила внесения изменений в программные документы, выполненные печатным способом [↑](#footnote-ref-8)