Министерство цифрового развития, связи и массовых коммуникаций Российской Федерации Ордена Трудового Красного Знамени

федеральное государственное бюджетное образовательное учреждение

высшего образования

«Московский технический университет связи и информатики»

Разрешаю допустить к защите

Зав. Кафедрой

Городничев М.Г.

\_\_\_\_\_\_\_\_\_\_\_\_\_2024г.

**ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА**

НА ТЕМУ

Разработка высокоуровневого типизированного языка программирования

JadeLoom на основе виртуальной машины JVM для упрощения разработки

программного обеспечения и улучшения производительности

Студент: Aбакаров Г.Г. Руководитель: Городничев М.Г.

Москва 2024г.

**МИНИСТЕРСТВО ЦИФРОВОГО РАЗВИТИЯ, СВЯЗИ И МАССОВЫХ КОММУНИКАЦИЙ РОССИЙСКОЙ ФЕДЕРАЦИИ**

**Ордена Трудового Красного Знамени федеральное государственное бюджетное образовательное учреждение высшего образования**

**«Московский технический университет связи и информатики»**

Абакаров Гасан Гаджирабаданович

Кафедра

Математическая кибернетика и информационные технологии

(название полностью)

«Утверждаю»

Зав. кафедрой Городничев М. Г.\_

«\_\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_2024г.

**З А Д А Н И Е**

**на выпускную квалификационную работу**

Студенту Абакарову Гасану Гаджирабадановичу гр.\_\_ БВТ2003

Направление (специальность) 09.03.01 Информатика и вычислительная техника

Форма выполнения выпускной квалификационной работы Бакалаврская работа

(Дипломный проект, дипломная работа, магистерская диссертация, бакалаврская работа)

Тема выпускной квалификационной работы «Разработка высокоуровневого типизированного языка программирования JadeLoom на основе виртуальной машины JVM для упрощения разработки программного обеспечения и улучшения производительности».

Утверждена приказом ректора № \_\_\_\_\_\_ от \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 20 23 г.

|  |  |
| --- | --- |
| 1. Исходные данные  Документация C++ stdlib  Документация Java  Документация по структуре JVM  2. Содержание расчетно-пояснительной записки  (перечень подлежащих разработке вопросов)  Введение  Раздел 1. Обзор структуры JVM  Раздел 2. Обзор конструкции языка програмирования  Раздел 3. Реализация языка программирования  Заключение | Объем работы в % и сроки выполнения по разделам  5% - 17.02.2024  30% - 17.03.2024  30% - 06.04.2024  30% - 12.05.2024  5% - 14.05.2024 |

3. Консультанты по ВКР (с указанием относящихся к ним разделов проекта):

(ФИО) \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(подпись)

(ФИО) \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(подпись)

4. Срок сдачи студентом законченной ВКР: 5 июня 2023 года

Дата выдачи задания: 6 февраля 2023 года

Руководитель Потапченко Т.Д.

(подпись) (ФИО)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_штатная\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_нагрузка

(штатная или почасовая)

Задание принял к исполнению\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(подпись студента)

**Примечание: Настоящее задание прилагается к законченной ВКР**

**Аннотация.**

В данной выпускной квалификационной работе была продемонстрирована возможность разработки высокоуровневого типизированного, компилируемого языка программирования для виртуальной машины jvm, а также был реализован язык программирования JadeLoom для демонстрации возможности разрааботки.

На основе проведенной работы был разработан новый язык программирования. Язык написан для виртуальной машины JVM на языке программирования Java и C++, что позволяет использовать существующую экосистему инструментов и библиотек, является обратно совместимой с языком программирования Java. Это означает, что программы, написанные на языке Java, могут быть выполнены без изменений на разработанном языке. Код, написанный на языке Java, будет компилироваться при использовании разработанного языка программирования, в том числе код компилятора.

Язык предлагает высокоуровневые абстракции и улучшенные механизмы типизации, что способствует упрощению процесса разработки и повышению качества программного обеспечения. Также язык предлагает наследование модификаторов, поддержку оператора goto, перегрузку арифметических операторов, regex, текстовые блоки, конфигурацию, используемую в ходе компиляции, которая, например, содержит информацию, какой файл содержит класс, или какой класс генерирует файл по умолчанию.

СОДЕРЖАНИЕ

[ВВЕДЕНИЕ 6](#__RefHeading___Toc4280_3138745730)

[1. Обзор структуры JVM 7](#__RefHeading___Toc4282_3138745730)

[1.1 Принцип работы JVM, основные компоненты JVM 7](#__RefHeading___Toc4284_3138745730)

[1.2 Структура файла, содержащий байткод JVM. 13](#__RefHeading___Toc4284_3138745730_Copy_1)

[1.3 Инструкции JVM, используемые для исполнения программ. 17](#__RefHeading___Toc4284_3138745730_Copy_2)

[2. Oбзор конструкций языка программирования. 18](#__RefHeading___Toc7046_1908015852)

[2.1 Введение в конструкции языка программирования JadeLoom. 18](#__RefHeading___Toc7048_1908015852)

[2.2 Синтаксис и семантика языка программирования JadeLoom. 24](#__RefHeading___Toc7048_1908015852_Copy_1)

[3. Реализация языка программирования JadeLoom. 28](#__RefHeading___Toc7046_1908015852_Copy_1)

[3.1. Разработка лексического и синтаксического анализатора кода. 28](#__RefHeading___Toc4335_383907082)

[3.1.1 Лексический анализатор кода. 33](#__RefHeading___Toc4051_2733629750_Copy_1)

[3.1.2 Синтаксический анализатор кода. 39](#__RefHeading___Toc4051_2733629750_Copy_2)

[3.2 Разработка генератора байткода для виртуальной машины JVM. 42](#__RefHeading___Toc4051_2733629750)

[3.2.1. Поиск перегруженого метода. 44](#__RefHeading___Toc4051_2733629750_Copy_4)

[3.2.2. Приведения типов, уровни наследования. 46](#__RefHeading___Toc4051_2733629750_Copy_3)

[3.2.3. Обработка циклов и операторов перехода. 48](#__RefHeading___Toc4051_2733629750_Copy_5)

[3.2.4. Оптимизация исполнительного кода. 51](#__RefHeading___Toc4051_2733629750_Copy_6)

[ЗАКЛЮЧЕНИЕ 52](#__RefHeading___Toc4292_3138745730)

[СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ 53](#__RefHeading___Toc4347_3138745730)

[ПРИЛОЖЕНИЯ. 54](#__RefHeading___Toc4347_3138745730_Copy_1)

**ВВЕДЕНИЕ**

Исследование, проведенное в ходе написания дипломной работы посвящено разработке нового языка программирования JadeLoom, который обладает обратной совместимостью с языком Java. Обратная совместимость позволяет использовать существующий код, написанный на языке Java, без необходимости его изменения, при разработке с использованием нового языка.

Целью данного исследования является создание нового языка программирования, который наследует преимущества языка Java и предоставляет дополнительные возможности и улучшения. Однако, при этом, новый язык должен быть совместимым с существующим кодом на языке Java, чтобы минимизировать усилия по переносу и адаптации уже существующих проектов.

Обратная совместимость с языком Java означает, что программы, написанные на языке Java, могут быть выполнены без изменений на новом языке программирования. Это включает возможность использования существующих библиотек, классов и функциональности языка Java в новом языке.

В ходе данного исследования будет проведен анализ языка Java и его особенностей, а также определены ключевые аспекты, которые должны быть учтены при разработке нового языка с обратной совместимостью. Результаты исследования будут использованы для создания прототипа нового языка программирования, который будет проверен на примерах существующего кода на языке Java.

Разработка нового языка программирования с обратной совместимостью с языком Java имеет потенциал значительно упростить процесс разработки программного обеспечения, улучшить его производительность и повысить эффективность использования существующего кода

**1. Обзор структуры JVM**

**1.1 Принцип работы JVM, основные компоненты JVM**

JVM является основной платформой для исполнения программ, написаных и скомпилированых на языке программирования JadeLoom и отвечает за исполнение байткода, сгенерированого в ходе компиляции программ JadeLoom. Основные компоненты JVM включают в себя, но не ограничиваются ими:

Загрузчик (Class Loader): Объект класса загрузчика отвечает за загрузку и инициализацию классов, которые представляют собой байткод, по мере необходимости во время выполнения программы. Класс-загрузчик загружает байт-код из файловой системы или сети и преобразует его в внутреннее представление JVM.

Существуют различные интерпретации загрузчиков, как системные интерпретации, так и пользовательские, нацеленые на выполнение задач по загрузке классов из файлов различного формата, например, позволяют загрузить класс из машинного кода или из программного кода, выполняя роль интерпретатора.

1. Системные загрузчики:

- Bootstrap ClassLoader: Это базовый низкоуровневый загрузчик, который отвечает за загрузку основных Java-классов, необходимых для запуска и работы функционала JVM.

- Extension ClassLoader: Oтвечает за загрузку расширения JVM, которые находятся в директории $JAVA\_HOME/jre/lib/ext. Данные расширения позволяет добавлять дополнительные функциональные возможности в стандартную Java-платформу.

- Application ClassLoader: Отвечает за загрузку классов из classpath приложения. Classpath может включать локальные директории, JAR-файлы и даже удаленные URL-адреса.

2. Пользовательские ClassLoaders:

- Разработчики могут создавать собственные реализации ClassLoader, расширяя базовый класс `ClassLoader`.

- Это позволяет загружать классы из альтернативных источников, таких как:

- Файловая система: Загрузка классов из произвольной локации на диске

- Сетевые ресурсы: Загрузка классов по HTTP, FTP или другим протоколам

- Базы данных: Извлечение классов из хранилищ данных

- Другие форматы: Загрузка классов, представленных в виде машинного кода, байт-кода для других VM, интерпретируемого программного кода и т.д.

- Пользовательские загрузчики часто используются в следующих сценариях:

- Создание модульных приложений

- Динамическая загрузка плагинов или компонентов

- Реализация специализированных сред выполнения (например, для языков, компилируемых в Java-байт-код)

3. OSGi ClassLoaders:

- Используются в модульных системах OSGi для обеспечения изолированной загрузки классов в каждом модуле.

- Каждый модуль OSGi имеет свой собственный загрузчик, который загружает только те классы, которые явно экспортируются другими модулями.

- Это позволяет избежать конфликтов между зависимостями классов в разных модулях.

- OSGi ClassLoaders обеспечивают более контролируемую и управляемую среду выполнения для модульных JVM приложений.

Runtime Data Areas - это набор логических областей памяти, используемых JVM для хранения различных типов данных во время выполнения Java-программы. Основные области входящие в Runtime Data Areas включают в себя:

Heap (Куча): Это основная область памяти, где хранятся объекты, создаваемые во время выполнения программы.

Heap управляется Garbage Collector, который автоматически освобождает память от неиспользуемых объектов.

Heap разделен на несколько поколений (new, old, permanent) для более эффективного управления памятью.

Method Area: Здесь хранится структурная информация о загруженных классах, такая как определение классов, методов, полей и констант.

Method Area является общей для всех потоков выполнения в JVM.

Java Stack: Каждый поток выполнения в JVM имеет свой собственный стек вызовов методов.

Стек хранит фреймы вызовов методов, включая локальные переменные и промежуточные результаты.

При вызове метода создается новый фрейм на верхушке стека, при возврате из метода - фрейм удаляется.

Program Counter Register: Хранит адрес (указатель) текущей выполняемой инструкции для каждого потока выполнения.

Native Method Stacks: Хранит стек вызовов для нативных (написанных на других языках и скомпилированных в машинный код) методов, используемых Java-программой.

Сборщик мусора (Garbage Collector) - это один из ключевых компонентов JVM, отвечающий за автоматическое управление памятью, используемой объектами JadeLoom. Он выполняет следующие основные функции:

1. Отслеживание ссылок: Сборщик мусора отслеживает, какие объекты в памяти JVM все еще имеют действительные ссылки на них (то есть, на них ссылаются другие объекты или части кода). Объекты без действительных ссылок считаются "мусором" и подлежат удалению.

2. Освобождение памяти: Когда сборщик мусора определяет, что объект больше не используется, он автоматически освобождает память, занимаемую этим объектом, делая ее доступной для повторного использования.

3. Алгоритмы сбора мусора: JVM может использовать различные алгоритмы сбора мусора, такие как "Поколенческий" (Generational), "Параллельный" (Parallel) или "Инкрементальный" (Incremental). Эти алгоритмы оптимизируют производительность очистки памяти с учетом характеристик приложения JadeLoom.

4. Планирование сбора мусора: Сборщик мусора определяет, когда необходимо проводить очистку памяти, чтобы избежать дефицита свободной памяти. Он может делать это автоматически или по запросу программы.

5. Сохранение целостности данных: Сборщик мусора обеспечивает, чтобы во время сбора мусора не было потери или повреждения данных, используемых программой JadeLoom.

Благодаря этим функциям, сборщик мусора JVM значительно упрощает разработку программ на JadeLoom, избавляя разработчиков от необходимости явного управления памятью. Это повышает производительность и надежность приложений. Необходимо также рассмотреть основные достоинства и недостатки сборщика мусора в JadeLoom.

Основные достоинства сборщика мусора:

1. Автоматизация управления памятью: Сборщик мусора автоматически отслеживает и освобождает неиспользуемую память, избавляя разработчиков от необходимости выполнять это вручную. Это значительно упрощает разработку приложений на JadeLoom.

2. Повышение надежности: Автоматическое управление памятью сводит к минимуму риск возникновения ошибок, связанных с утечками памяти или повреждением данных.

3. Увеличение производительности: Эффективные алгоритмы сбора мусора могут значительно повысить производительность приложений JadeLoom за счет оптимизации использования памяти.

4. Масштабируемость: Сборщик мусора позволяет JVM поддерживать приложения JadeLoom, использующие большие объемы данных и память.

Основные недостатки сборщика мусора:

1. Накладные расходы: Работа сборщика мусора сама по себе требует вычислительных ресурсов, что может привести к временным задержкам в работе приложения.

2. Непредсказуемость производительности: Запуск сборки мусора в неподходящие моменты может привести к периодам снижения производительности приложения.

3. Сложность оптимизации: Настройка параметров сборщика мусора для оптимизации производительности может быть непростой задачей, особенно для крупных и сложных приложений.

4. Ограничения обнаружения утечек памяти: Сборщик мусора не может обнаружить все возможные ситуации утечек памяти, особенно связанные с циклическими ссылками.

Несмотря на перечисленные выще недостатки, сборщик мусора в целом значительно упрощает разработку и эксплуатацию приложений на JadeLoom, предоставляя разработчикам инструмент автоматического управления памятью.

Также необходимо рассмотреть ключевой компонент JVM - Execution Engine - отвечающий за непосредственное исполнение байт-кода, сгенерированного в ходе компиляции программ, написаных на языке программирования JadeLoom. Его основные функции включают в себя:

1. Интерпретацию байт-кода: Execution Engine интерпретирует и выполняет инструкции байт-кода, генерируемого компилятором JadeLoom.

2. Оптимизация производительности: Одна из ключевых задач Execution Engine - повышение производительности исполнения кода JadeLoom. Для этого используются несколько основных оптимизирующих техник:

3. Execution Engine может применять динамическую (Just-In-Time) компиляцию, преобразуя некоторые, наиболее часто используемые участки байт-кода в оптимизированный машинный код для повышения производительности.

Инлайнинг методов:

- Execution Engine анализирует вызовы методов и может "встраивать" код небольших, часто используемых методов прямо в место вызова.

- Данный метод устраняет накладные расходы на вызов методов и позволяет применять дополнительные оптимизации.

Удаление общих подвыражений:

- Execution Engine обнаруживает и устраняет повторяющиеся вычисления одних и тех же выражений. Данная техника оптимизации помогает избежать выполнения некоторых вычислений, ускоряет и оптимизирует выполнение программы.

Развертывание циклов

- Для коротких, часто выполняемых циклов Execution Engine может развернуть их тело, заменив цикл прямым кодом. Данная техника оптимизации позволяет избежать накладных расходов на управление циклом.

Профилирование и адаптивная оптимизация:

- Execution Engine собирает профилирующую информацию о выполнении программы и динамически применяет оптимизации к наиболее часто исполняемым участкам кода, что позволяет непрерывно совершенствовать производительность в процессе работы приложения.

3. Управление стеком вызовов: Execution Engine отвечает за организацию и управление стеком вызовов методов, обеспечивая правильность и целостность выполнения программы.

4. Встроенные библиотеки: Execution Engine предоставляет доступ к большому набору встроенных библиотек JadeLoom, реализующих стандартные функциональные возможности.

5. Взаимодействие с другими компонентами JVM: Execution Engine тесно взаимодействует с другими основными компонентами JVM, такими как сборщик мусора, загрузчик классов и системой безопасности.

**1.2 Структура файла, содержащий байткод JVM.**

Во время компиляции программ, написаных на языке программирования JadeLoom, исходный код генерируется в файл, имеющий бинарный формат, содержащий байткод для загрузки и исполнения в JVM. Информация в нем записывается без отступов между последовательными частями информации. 16-битные и 32-битные значения записываются с помощью двух или четырех последовательных 8-битных байтов.

Файл с байткодом содержит следующую информацию.

Магическое число, сигнатура. Первые четыре байта каждого файла, сгенерированного после компиляции программ JadeLoom всегда равны значениям 0xCA 0хFE 0хBA 0хBE. Данные четыре байта идентифицируют наиболее часто используемый формат файла, содержащий байткод, для выполнения на JVM.

Версия файла. Следующие четыре байта содержат мажорную и минорную версию JVM. Номера определяют совместимую версию формата файла. Если файл имеет основной мажорную версию M и минорную m, то мы обозначаем эту версию как M.m.

У каждой JVM есть ограничения по поддерживаемым версиям class-файлов. Например, Java 11 поддерживает major версию с 45 до 55, Java 12 — с 45 по 56.

Пул констант. Данный участок представляют из себя таблицу структур, представляющих строковые константы, имена классов, интерфейсов, полей, методов и другие константы, которые есть в структуре ClassFile и ее подструктурах. Каждый элемент пула констант начинается с однобайтового тега, определяющего тип константы. В зависимости от типа константы следующие байты могут быть непосредственным значением константы или ссылкой на другой элемент в пуле. Основными типами элементов пула констант являются:

1. Константы примитивных типов:

`CONSTANT\_Integer\_info` - целочисленные константы, 4 байта

`CONSTANT\_Float\_info` - константы с плавающей точкой, 4 байта

`CONSTANT\_Long\_info` - длинные целочисленные константы, 8 байт

`CONSTANT\_Double\_info` - константы с двойной точностью, 8 байт

2. Ссылки на классы, интерфейсы и пакеты:

`CONSTANT\_Class\_info` - ссылка на класс или интерфейс, 2 байта

`CONSTANT\_Package\_info` - ссылка на пакет, 2 байта

3. Ссылки на поля и методы:

`CONSTANT\_Fieldref\_info` - ссылка на поле класса

`CONSTANT\_Methodref\_info` - ссылка на метод класса

`CONSTANT\_InterfaceMethodref\_info` - ссылка на метод интерфейса

4. Строковые константы:

`CONSTANT\_String\_info` - строковые литералы

5. Имена и дескрипторы:

`CONSTANT\_NameAndType\_info` - имя и дескриптор (сигнатура) поля или метода

`CONSTANT\_Utf8\_info` - строковые представления имен и дескрипторов

Пул констант организован в виде массива записей переменной длины, где каждая запись представляет одну константу. Он используется для эффективного доступа к различным сущностям, таким как классы, методы, поля и литералы, во время выполнения программы.

Структура и содержимое пула констант являются ключевыми аспектами, определяющими внутреннее представление Java-классов в байт-коде JVM.

Флаги доступа. — Это поле занимает 2 байта и содержит следующие флаги:

ACC\_PRIVATE - класс доступен только внутри определяющего пакета

ACC\_PUBLIC - класс доступен из любой зоны видимости

ACC\_PROTECTED - класс доступен в пределах пакета и наследующих классов

ACC\_PACKAGE (отсутствие модификаторов) - класс доступен только внутри определяющего пакета

ACC\_FINAL - класс не может быть унаследован

ACC\_INTERFACE - класс является интерфейсом

ACC\_ABSTRACT - класс является абстрактным (не может быть напрямую инстанцирован)

ACC\_SYNTHETIC - класс сгенерирован компилятором.

ACC\_ANNOTATION - класс является аннотацией

ACC\_ENUM - класс является перечислением (enum)

This class. Ссылка на запись имени класса в пуле констант.

Super class. Ссылка на запись имени родительского класса в пуле констант.

Интерфейсы. Массив ссылок на интерфейсы, расширяющие представления класса.

Поля. Таблица структур переменной длины. По одной для каждого поля с описанием типа поля и названия (со ссылкой на пул констант).

Методы. Таблица структур методов переменной длины. Для каждого метода содержится следующая информация: дескриптор метода (тип возвращаемого значения и список аргументов), количество слов, необходимых для локальных переменных метода, максимальное количество слов стека, необходимых для стека операндов метода, таблицу исключений, перехватываемых методом, байт-коды метода и таблица номеров строк.

Определение методов в class-файле содержит следующую информацию:

1. Флаги доступа (2 байта):

- Флаги, определяющие модификаторы доступа:

ACC\_PUBLIC - метод является public (доступен из любого места)

ACC\_PRIVATE - метод является private (доступен только внутри определяющего класса)

ACC\_PROTECTED - метод является protected (доступен в пределах пакета и наследующих классов)

ACC\_PACKAGE (отсутствие модификаторов) - метод является package-private (доступен только внутри определяющего пакета)

ACC\_STATIC - метод является статическим

ACC\_FINAL - метод является final (не может быть переопределен)

ACC\_SYNCHRONIZED - метод является synchronized

ACC\_BRIDGE - метод является synthetic bridge-методом

ACC\_VARARGS - метод принимает переменное число аргументов

ACC\_NATIVE - метод является native

ACC\_ABSTRACT - метод является абстрактным

ACC\_STRICT - метод требует строгое соблюдение IEEE 754 (плавающая точка)

ACC\_SYNTHETIC - метод был сгенерирован компилятором, а не написан вручную

2. Индекс имени (2 байта):

- Индекс в пуле констант, который ссылается на запись, содержащую имя метода.

3. Индекс дескриптора (2 байта):

- Индекс в пуле констант, который ссылается на запись, содержащую дескриптор (сигнатуру) метода.

- Дескриптор описывает типы параметров и возвращаемый тип метода.

4. Аттрибуты:

- Список атрибутов метода, таких как:

- `Code`: содержит байт-код метода и информацию об обработке исключений.

- `Exceptions`: содержит список типов исключений, которые может выбрасывать метод.

- `Signature`: содержит обобщенную сигнатуру метода.

- `Annotations`: содержит аннотации, применяемые к методу.

- `MethodParameters`: содержит информацию о параметрах метода.

Атрибуты. Структуры переменной длины, описывающие каждый атрибут.

**1.3 Инструкции JVM, используемые для исполнения программ.**

Инструкции JVM состоят из опкода, определяющего выполняемую операцию, за которым следуют одно или более операндов, или не следует ни одного операнда, представляющих собой значения, с которыми нужно работать. Они представляют собой базовые операции, которые выполняет виртуальная машина для интерпретации и выполнения Java-программ. Инструкции JVM играют важную роль, которая заключается я в следующем:

1. Представление кода: Инструкции JVM являются низкоуровневым представлением Java-кода. Когда Java-код компилируется, он преобразуется в последовательность инструкций JVM, которые могут быть выполнены виртуальной машиной.

2. Обеспечение портативности: Инструкции JVM являются платформенно-независимым представлением кода. Это означает, что Java-программы, скомпилированные в инструкции JVM, могут быть выполнены на любой системе, где установлена Java Virtual Machine, независимо от аппаратной платформы.

3. Реализация функциональности: Инструкции JVM определяют основные операции, которые могут выполняться виртуальной машиной, такие как загрузка и сохранение данных, арифметические и логические операции, управление потоком выполнения, работа с объектами и массивами, обработка исключений и т.д.

4. Оптимизация производительности: Понимание инструкций JVM помогает разработчикам анализировать и оптимизировать производительность Java-приложений, поскольку они могут видеть, какие именно операции выполняет виртуальная машина.

5. Отладка и анализ кода: Знание инструкций JVM важно для отладки и анализа Java-приложений, поскольку позволяет изучать и понимать внутреннее представление кода на низком уровне.

**2. Oбзор конструкций языка программирования.**

**2****.1 Введение в конструкции языка программирования JadeLoom.**

Разрабатываемый язык программирования должен стать важным и значимым инструментом, с помощью которого разработчики могут в краткие сроки создать программное обеспечение для виртуальной машины JVM, что, в свою очередь, гипотетически может применяться при написании кода для андроид. Несмотря на то, что андроид использует ART для исполнения программ, специально разработанных для платформы андроид, который, в свою очередь, использует DEX байткод (Dalvik executable), мы можем скомпилировать байткод JVM в DEX байткод, но возможности байткода JVM не ограничиваются компиляцией в DEX байткод. Например, байткод JVM можно скомпилировать в машинный код, или код javascript, с помощью GraalVM. Рассмотрим основные конструкции языка программирования, которые широко применяются в различных языках, в том числе языке JadeLoom.

Конструкции языка программирования JadeLoom включают в себя условные операторы, операторы перехода, циклы, функции, объекты, методы, выьов и обработку исключений. Каждая из этих конструкций предоставляет различные возможности для разработчиков.

Циклы, условные операторы и операторы перехода позволяют выполнять блоки несколько раз в зависимости от условии. Они особенно полезны для обработки повторяющихся задач, таких как обход элементов массива или выполнение итераций по определенному диапазону значений.

Классы — это основные компоненты объектно ориентировочного языка программирования JadeLoom. Они представляют из себя шаблоны, формальные описания объектов, и определяют состояние и поведение объектов

Определение классов содержат:

- Поля, или переменные класса, которые представляют состояние объектов, созданных на основе класса. Они могут иметь различные типы данных и могут быть доступными только внутри класса или также доступными извне, в зависимости от модификатора доступа. Также поля, или переменные класса, могут быть статическими, или динамическими. Статические переменные класса (также известные как переменные класса или переменные уровня класса) относятся к переменным, которые разделяются между всеми объектами, созданными на основе этого класса. Они хранятся в единственном экземпляре и доступны для всех экземпляров класса. Динамические переменные класса (также известные как переменные экземпляра или переменные объекта) относятся к переменным, которые связаны с каждым отдельным объектом, созданным на основе класса. Каждый объект имеет свою собственную копию динамических переменных класса.

- Методы – это подпрограммы, определенные внутри класса, которые определяют поведение объектов этого класса, используемые для выполнения скомпилированного кода. Они позволяют группировать фрагменты кода в модули или подпрограммы. Mетоды могут принимать параметры, передаваемые им для обработки, и могут возвращать значения, полученые в ходе обработки параметров, включая как одно значение, так и несколько значений, или не возвращать значения. Это обеспечивает модульность, повторное использование кода и возможность работы с различными типами данных. Mетоды могут обрабатывать не только полученые параметры, но и глобальные параметры, параметры класса или параметры, заданные в области видимости (scope).

Описание метода класса включает в себя следующие аспекты:

1. Имя метода: Методы имеют уникальное имя, которое идентифицирует их в рамках класса.

2. Тип возвращаемого значения: Методы могут иметь тип возвращаемого значения или быть объявлены с типом void, если они не возвращают значения. Тип возвращаемого значения определяет тип данных, который будет возвращен методом при его выполнении.

3. Параметры: Методы могут принимать параметры, которые представляют собой значения, передаваемые в метод для его выполнения. Параметры определяются в сигнатуре метода и используются внутри метода для доступа к переданным значениям.

4. Тело метода: Тело метода содержит набор инструкций, которые определяют действия, выполняемые методом. Оно определяет, что будет происходить при вызове метода и как будет обрабатываться переданные параметры.

5. Уровень доступа: Методы могут иметь различные уровни доступа, такие как public, private, protected или без указания уровня доступа (пакетный уровень доступа). Уровень доступа определяет, где и как можно вызывать методы класса.

Методы класса позволяют инкапсулировать поведение объектов и предоставлять интерфейс для взаимодействия с ними. Они могут быть использованы для выполнения различных операций, обработки данных, изменения состояния объектов и возврата результатов. Методы являются основным строительным блоком объектно-ориентированного программирования и позволяют разделить код на более мелкие и управляемые части. Существуют статические методы класса и динамические методы класса, и отличаются они следующим образом:

Статические методы:

• Статические методы принадлежат классу, а не конкретному экземпляру класса. Они могут быть вызваны без создания объекта класса.

• Статические методы не имеют доступа к нестатическим полям и методам класса, так как они не привязаны к конкретному экземпляру объекта.

Динамические методы:

• Динамические методы (или нестатические методы) привязаны к конкретному экземпляру класса. Они требуют создания объекта класса для вызова метода.

• Динамические методы имеют доступ к нестатическим полям и методам класса, а также к другим динамическим методам.

• Динамические методы могут быть переопределены в подклассах при наличии наследования.

- Конструкторы и блоки кода: Конструкторы, и блоки кода - это специальные методы, которые вызываются при создании объектов класса. Они инициализируют начальные значения полей объекта и могут выполнять другие необходимые действия при создании объекта. Конструкторы в JadeLoom могут быть определены только как динамические методы, в то время, как блоки кода могут быть определены как динамические методы, так и статические, но не могут принимать параметры. Конструкторы и блоки кода не возвращают значение.

- Вложенные классы: В языке программирования JadeLoom классы могут быть вложены в другие классы. Вложенные классы могут использоваться для организации и структурирования кода.

- Имя класса: Имя класса является идентификатором, который идентифицирует класс в программе. Имя класса должно быть уникальным в пределах пакета.

Объекты и классы являются ключевыми концепциями объектно-ориентированного программирования. Они позволяют организовывать код в виде объектов определенного класса, которые имеют свою структуру. ООП обеспечивает принципы инкапсуляции, наследования и полиморфизма.

Полиморфизм - это концепция объектно-ориентированного программирования, которая позволяет объектам одного типа проявлять различное поведение в зависимости от контекста использования. Это означает, что объекты могут быть рассмотрены и использованы как экземпляры своего собственного типа, а также как экземпляры типа, от которого они наследуются или реализуют интерфейс.

Полиморфизм позволяет разрабатывать гибкий и расширяемый код, так как он способствует созданию общих интерфейсов и абстракций. Он облегчает работу с коллекциями объектов, позволяя одновременно обрабатывать объекты разных типов через общий интерфейс.

Существуют два основных типа полиморфизма:

1. Полиморфизм подтипов (Subtype Polymorphism): Этот тип полиморфизма основан на наследовании и возможности объектов-потомков быть использованными вместо объектов-родителей. При полиморфном использовании объектов-потомков они могут вызывать свои собственные реализации методов, переопределяя поведение, определенное в родительском классе.

2. Параметрический полиморфизм (Parametric Polymorphism): Этот тип полиморфизма позволяет писать обобщенный код, который может работать с различными типами данных, не зависимо от их конкретного типа. В языках программирования, поддерживающих параметрический полиморфизм, можно создавать шаблонные функции или классы, которые могут быть параметризованы различными типами данных.

Полиморфизм помогает создавать более гибкие и модульные программы, упрощает повторное использование кода и способствует разделению ответственности между объектами и классами. Он является одним из ключевых принципов объектно-ориентированного программирования.

Наследование - это концепция объектно-ориентированного программирования, которая позволяет создавать новые классы (потомки) на основе уже существующих классов (родителей). При наследовании потомок наследует свойства и методы родителя, что позволяет расширять и переиспользовать код.

В наследовании класс-потомок может наследовать все публичные и защищенные свойства и методы класса-родителя. Это означает, что потомок может использовать и изменять унаследованные свойства и методы, а также добавлять свои собственные.

Инкапсуляция является одним из основных принципов объектно-ориентированного программирования (ООП). Этот принцип заключается в объединении данных и методов, работающих с этими данными, внутри классов и скрытии их реализации от внешнего кода.

Инкапсуляция позволяет скрыть данные от прямого доступа извне класса. Таким образом, данные не могут быть изменены или получены напрямую. Вместо этого, доступ к данным осуществляется через специальные методы класса, называемые геттерами и сеттерами. Геттеры позволяют получить значения данных, а сеттеры - устанавливать новые значения. Это обеспечивает контролируемый доступ к данным и позволяет классу установить правила и проверки для доступа к своим данным.

Инкапсуляция также позволяет скрыть внутреннюю реализацию класса. Внешний код может использовать только публичные методы класса, которые определены интерфейсом класса. Внутренние детали реализации класса, такие как приватные методы или переменные, остаются скрытыми и недоступными для внешних объектов или классов. Это позволяет изменять внутреннюю реализацию класса без влияния на внешний код, что упрощает поддержку и развитие программных систем.

JadeLoom поддерживает различные типы данных, которые могут быть использованы для объявления переменных, параметров методов, возвращаемых значений и других элементов программы. Oсновные типы данных в JadeLoom включают в себя:

1. Примитивные типы данных:

- boolean: логический тип данных, который может принимать значения true или false.

- byte: 8-битное целое число со знаком, диапазон значений от -128 до 127.

- short: 16-битное целое число со знаком, диапазон значений от -32,768 до 32,767.

- int: 32-битное целое число со знаком, диапазон значений от -2,147,483,648 до 2,147,483,647.

- long: 64-битное целое число со знаком, диапазон значений от -9,223,372,036,854,775,808 до 9,223,372,036,854,775,807.

- float: 32-битное число с плавающей точкой, представленное в формате IEEE 754.

- double: 64-битное число с плавающей точкой, представленное в формате IEEE 754.

- char: 16-битный символ Unicode.

2. Ссылочные типы данных:

- Object: тип данных, являющийся суперклассом для всех классов в JadeLoom.

3. Специальные типы данных:

- Массивы: тип данных, являющееся упорядоченной коллекцией элементов указаного типа (кроме void).

- void: тип данных, используемый для методов, которые не возвращают значения.

Обработка исключений позволяет программе обрабатывать ошибки и исключительные ситуации. Она помогает предотвратить аварийное завершение программы и предоставляет механизмы для управления и восстановления от ошибок.

**2****.2 Синтаксис и семантика языка программирования JadeLoom.**

Исходный код JadeLoom состоит из следующих основных компонентов:

Объявление пакета: В языке программирования JadeLoom пакет, к которому принадлежат классы, статические методы и поля класса, обычно варьируется от названия относительного пути, из которого импортируется класс. Пакет может быть объявлен единижды, с помощью ключевого слова package, например:

package пакет.имя;

Импорты: Объявления, позволяющие использовать классы и пакеты из внешних библиотек. Позволяют сократить необходимость полного указания имен классов. Импорт выполняется следующим образом:

import пакет.имя.имякласса;

import пакет.имя.имякласса as новоеимя;

import пакет.имя.имякласса.\*;

Комментарии: Текстовые пояснения, которые не влияют на выполнение программы. Помогают разработчикам понять логику и назначение различных частей кода. Коментарии в языке программирования JadeLoom объявляются следующим образом:

// однострочный коментарии

/\* многострочный коментарии \*/

Объявления классов:

- Определение классов, которые являются основными строительными блоками программы.

- В каждом классе содержатся поля (переменные) и методы.

Эти основные компоненты - классы, методы, переменные, операторы, ключевые слова, комментарии и импорты - образуют структуру исходного кода JadeLoom-программы. Kлассы в JadeLoom определяются с использованием синтаксиса, который описывает их структуру и поведение, а также семантику, которая определяет их значение и смысл.

Основные определения класса JadeLoom:

- Класс является основным строительным блоком объектно-ориентированного программирования в JadeLoom.

- Класс может содержать поля (переменные класса), которые определяют состояние объектов этого класса.

- Класс также может иметь конструкторы, которые используются для инициализации объектов этого класса.

- Класс может определять методы, которые определяют поведение объектов этого класса. Методы могут возвращать значения и принимать параметры.

- Класс может наследоваться от другого класса с использованием ключевого слова `extends`. Это позволяет классу наследовать поля и методы родительского класса.

- Класс может реализовывать один или несколько интерфейсов с использованием ключевого слова `implements`. Интерфейсы определяют контракты, которые класс обязан выполнить.

- В JadeLoom классы могут быть вложенными, то есть один класс может быть определен внутри другого класса.

В этом примере у класса `Main` есть приватное поле `поле`, конструктор, метод `метод`.

Также синтаксис и семантика методов в языке программирования Java включают в себя определение и использование методов.

Синтаксис определения метода в языке программирования JadeLoom выглядит следующим образом:

<<обобщения>> <возвращаемый\_тип> <имя\_метода>(<параметры>) [throws <исключения>]{

// Тело метода

}

- <возвращаемый\_тип>: Возвращаемый тип указывает тип данных.

- <имя\_метода>: Имя метода должно быть уникальным в пределах класса и должно быть осмысленным, отражающим его назначение.

- <обобщения>: Обобщенные типы позволяют указывать тип или типы данных, с которыми будет работать метод, путем добавления типизированных параметров в определение класса, метода или интерфейса. Типизированный параметр представляет собой параметр для фактического типа данных, который будет использоваться при вызове метода.

- <параметры>: Параметры метода являются значениями, которые передаются в метод для выполнения операций. Они указываются внутри круглых скобок и могут быть необязательными. Если метод не принимает параметры, скобки остаются пустыми.

- `// Тело метода`: Внутри фигурных скобок находится код, который будет выполняться при вызове метода.

Чтобы вызвать метод в JadeLoom, используется следующий синтаксис:

<выражение>.<имя\_метода>(<аргументы>);

<выражение>.<<обобщения>> <имя\_метода>(<аргументы>);

<имя\_метода>(<аргументы>);

<<обобщения>> <имя\_метода>(<аргументы>);

- `<имя\_метода>` (Method Name): Указывается имя вызываемого метода.

- `<аргументы>` (Arguments): Аргументы представляют выражения, которые передаются в метод при его вызове. Они указываются внутри круглых скобок и разделяются запятыми. Аргументы должны соответствовать типам и порядку параметров, объявленным в определении метода.

Семантика методов включает их поведение и воздействие на программу. Методы могут выполнять различные операции, обрабатывать данные, изменять состояние объектов и возвращать результаты. Они позволяют организовать код в более модульную и повторно используемую структуру, улучшают читабельность и облегчают сопровождение программы.

Объявление переменных, и присвоение значения переменным является одной из основных и наиболее используемых операции в JadeLoom. Переменные используются для хранения данных.

Объявление переменных:

В языке программирования JadeLoom переменные объявляются явно, с указанием типа и имени, или неявно, с помощью ключевого слова var. Синтаксис объявления переменной выглядит следующим образом:

тип\_переменной имя\_переменной;

тип\_переменной имя\_переменной = выражение;

var имя\_переменной = выражение;

new тип\_переменной имя\_переменной (выражениe, ...)

new тип\_переменной имя\_переменной [выражение] ...

Например, для объявления переменной целочисленного типа с именем "число", необходимо использовать следующую конструкцию:

int число;

Объявление переменных также может быть сопровождено их инициализацией, то есть присвоением начального значения в момент объявления.

**3. Реализация языка программирования JadeLoom.**

**3.1. Разработка лексического и синтаксического анализатора кода.**

Язык программирования состоит из синтаксиса, семантики, стандартной библиотеки, инструментов и документации. Данный язык программирования содержит стандартные библиотеки, предоставляемые виртуальной машиной JVM.

Синтаксис определяет, как должны быть структурированы и записаны программы на языке программирования языке, включая ключевые слова, операторы, переменные, функции и классы. Семантика задает значение и смысл этих синтаксических элементов, определяя, что должно происходить при их выполнении.

Стандартная библиотека предоставляет набор предопределенных функций и модулей, упрощающих разработку приложений. Инструменты, такие как компиляторы, интерпретаторы и отладчики, помогают писать, тестировать и оптимизировать программы. Документация описывает синтаксис, семантику, стандартную библиотеку и рекомендации по использованию языка.

Среди инструментов языка программирования наиболее важнейшим инструментом является компилятор. В ходе выполнения данной работы был написан компилятор, способный перевести код, написаный на языке программирования JadeLoom, на байткод, исполняемый виртуальной машиной JVM. Реализация компилятора начинается с разработки лексического анализатора и синтаксического анализатора.

Лексический анализатор отвечает за разбиение входного исходного кода на лексемы - базовые элементы языка, такие как ключевые слова, идентификаторы, числа, операторы и т.д. Это делается путем чтения входного потока символов и сопоставления их с шаблонами, определенными в спецификации языка.

Синтаксический анализатор затем использует эти лексемы для построения абстрактного синтаксического дерева (AST) - иерархической структуры, которая представляет синтаксическую структуру программы в соответствии с грамматикой языка. Этот процесс называется синтаксическим анализом.

После получения AST компилятор может выполнять различные оптимизации и преобразования для улучшения качества генерируемого кода. Это может включать:

- Семантический анализ - проверка на соответствие типов, область видимости переменных и т.д.

- Генерация промежуточного представления (IR) - более низкоуровневая форма, оптимизированная для дальнейшей обработки.

- Оптимизации кода - применение алгоритмов для уменьшения размера и повышения производительности.

Заключительным этапом является генерация целевого кода - перевод оптимизированного промежуточного представления в машинные инструкции, понятные целевой аппаратной платформе.

Синтаксический анализатор был реализован на языке C++ методом рекурсивного спуска, Рекурсивный спуск - это подход к реализации синтаксического анализатора, при котором синтаксический анализатор представляет собой набор взаимосвязанных функций, каждая из которых соответствует одному из правил грамматики языка. Данные функции рекурсивно вызывают друг друга, анализируя входной поток символов и распознавая соответствующие синтаксические конструкции.

Процесс синтаксического анализа организован в виде "спуска" по структуре синтаксических правил языка.

Данный подход отличается простотой реализации и наглядностью кода парсера, но может быть менее эффективным для языков с левой рекурсией в правилах грамматики.

К основным преимуществам рекурсивного спуска относятся:

Простота реализации кода

Возможность ручной доработки кода

К основным ограничениям рекурсивного спуска относятся:

Большие трудозатраты по сравнению с генераторами кода.

В ходе реализации лексического анализатора кода методом рекурсивного спуска были выделены следующие правила грамматики:

1. Программа:

- `Программа ::= [Пакет] {Импорт} {Тип} {Поле} {Метод} `

- Программа, написанная на языке программирования JadeLoom, состоит из объявления пакета, объявлений импорта, типов, полей и функции

2. Объявление пакета:

- `Пакет ::= "package" имя ";"`

- Объявление пакета начинается с ключевого слова "package" и указывает полное имя пакета.

3. Объявление импорта:

- `Импорт ::= "import" ["static"] идентификатор ["." "\*"] ["as" ] ";"`имя

- Объявление импорта начинается с ключевого слова "import". Оно может быть обычным импортом класса или статическим импортом. Также возможен импорт всех классов, статических методов и полей пакета с помощью "\*", и ипморт класса под другим названием, отличным от оригинального названия импортируемого класса.

4. Объявление типа:

- `Тип ::= Класс | Интерфейс | Перечисление`

- Объявление типа может быть объявлением класса, интерфейса или перечисления.

5. Объявление класса:

- `Класс ::= {Модификатор} "class" имя ["<" СсылочныйТип имя {"," СсылочныйТип имя} ">"] [" extends" тип] [" implements" {тип}] "{" { [{Модификатор} ":{" ] { Поле | Метод | Класс | Интерфейс | Перечисление | БлокИнициализации | Конструктор } ["}" ] } "}"

- Объявление класса начинается с ключевого слова "class", за которым следует идентификатор класса..

6. Объявление интерфейса:

- `Интерфейс ::= {Модификатор} "interface" имя ["<" СсылочныйТип имя {"," СсылочныйТип имя} ">"] [" extends" {тип}] "{" { [{Модификатор} ":{" ] { Поле | Метод | Класс | Интерфейс | Перечисление } ["}" ] } "}"

- Объявление интерфейса начинается с ключевого слова "interface", за которым следует идентификатор интерфейса. Интерфейс может иметь типовые параметры и может наследовать от одного или более других интерфейсов. Примечательно, что, в отличии от обычных классов, интерфейсы не содержат конструкторы или блоки инициализации

7. Объявление перечисления:

- `Перечисление ::= {Модификатор} "enum" имя [" implements" {тип}] "{" имя [{"," имя}] { [{Модификатор} ":{" ] { Поле | Метод | Класс | Интерфейс | Перечисление | БлокИнициализации | Конструктор } ["}" ] } "}" `

- Объявление перечисления начинается с ключевого слова "enum", за которым следует идентификатор перечисления. Перечисление может реализовывать интерфейсы. Перечисление содержит список констант перечисления, разделенных запятыми, и представленных в виде имен.

8. Объявление полей:

- `Поле ::= {Модификатор} ИдентификаторТипа { "[]" } имя [ "=" выражение ] {"," имя [ "=" выражение ] } ; `

- Объявление поля начинается с объявления идентификатора типа, за которым следует имя, и, необязательно, присвоение выражения.

9. Объявление идентификатора типа:

- `ИдентификаторТипа ::= „byte“ | „short“ | „int“ | „long“ | „char“ | „float“ | „double“ | „boolean“ | СсылочныйТип `

- .Идентификатор типа используется для определения типа данных, и может быть либо примитивным типом ( „byte“ | „short“ | „int“ | „long“ | „char“ | „float“ | „double“ | „boolean“ ), либо ссылочным типом.

10. Объявление ссылочного типа:

- СсылочныйТип: имя { '.' имя } ['<' {',' СсылочныйТип } '>'] { '.' имя ['<' {',' СсылочныйТип } '>'] }

- Ссылочные типы используются для определения классов, интерфейсов и перечислении. Синтаксис объявления ссылочного типа состоит из следующих элементов:

Имя - это название класса, перечисления или интерфейса, определяющего тип объекта.

'.' - оператор доступа, используется для обращения к вложенным типам.

'<' и '>' - угловые скобки, используемые для указания типовых параметров.

',' - разделитель, используемый для перечисления нескольких типовых параметров.

11. Объявление метода:

- Метод: {Модификатор} ["<" СсылочныйТип имя {"," СсылочныйТип имя} ">"] ИдентификаторТипа имя "(" [ ИдентификаторТипа имя {"," } ИдентификаторТипа имя ] ")" "{" { выражение } "}"

- Объявление метода определяет реализацию функционала, доступной через вызов данного метода. Оно включает в себя различные элементы, такие как модификаторы доступа, типовые параметры, возвращаемый тип, имя метода, параметры метода и инструкции метода. Эти элементы определяют поведение и характеристики метода

**3.1.1 Лексический анализатор кода.**

Процесс лексического анализа подразумевает собой процесс аналитического разбора входного текста на последовательный набор лексем. Лексема — это минимальная значимая последовательность символов. Чаще всего лексемы в языке программировании именуется токенами. Токен содержит такую информацию, как:

- Позиция токена — информация о местоположении токена в исходном коде, включая строку, столбец, смещение от начала файла.

- Тип токена — тип токена определяет, к какому синтаксическому элементу языка относится данный токен.

- Значение токена — представляет собой актуальное значение или выражение, представленное токеном, например, имя переменной, число, строка

В ходе реализации лексического анализатора кода были выделены следующие типы токенов:

Ключевые слова: Зарезервированные слова языка, которые представляют собой основными структурыми элементами синтаксиса.

Идентификаторы: Имена переменных, функций, классов и других сущностей, определенных в коде. Важно также отметить, что идентификаторы содержат буквы латинского алфавита, буквы национальных алфавитов, символ «$», подчеркивание и цифры. Идентификаторы различаются регистром и не могут начинаться с цифры. Идентификатором также не может быть зарезервированое ключевое слово.

Числовые литералы: Реализованы способы непосредственного представления числового значения в коде, такие, как целые числа, числа с плавающей точкой, шестнадцатеричные числа, восьмеричные числа, бинарные числа. В числовых литералах используется арабские символы, точка, подчеркивание и символы латинского алфавита. Для обозначения бинарных чисел используется префикс «0b», для обозначения шестнадцатеричных чисел используется префикс «0x», для обозначения восьмеричных чисел используется префикс «0». Существуют десятеричные и шестнадцатеричные числа с плавающей точкой, десятеричные числа с плавающей точкой могут иметь префикс «0.», «.» или содержать точку и десятеричные цифры, шестнадцатеричные числа имеют префикс «0x» и могут иметь целую часть, разделитель «.», дробную часть.

Строковые литералы, такие как многострочные литералы и однострочные литералы. Многострочные строковые литералы заключаются в три двойные кавычки, в то время как однострочные строковые литералы заключаются в двойные кавычки. Строковые литералы содержат обычные символы, символы юникода (\uXXXX, где XXXX — четырехзначный шестнадцатеричный код unicode — символа), ASCII - символы (\0OOO, где ООО — трехзначный восьмеричный код ASCII — символа), управляющие символы (\n - новая строка, \t - горизонтальная табуляция, \r - возврат каретки, \f - перевод страницы, \b - забой), символы (\" - ", \' - ', \\ - \).

Логические литералы: true, false.

Математические операторы, такие как «+», «-», «/», «\*», «\*\*», «%»

Битовые операции, такие как «!», «&», «|», «^».

Логические операции, такие как «&&», «^^», «||»

Oператоры сравнения, такие как «>», «<», «>=», «<=», «==», «!=»

Oператоры присваивания, такие как «=», «+=», «-=», «/=», «\*=», «\*\*=», «%=», «&=», «|=», «^=», «!=».

Разделители: Символы, используемые для разделения элементов кода, такие как, «,», «;», «:», «(«, «)», «{» , «}», «[», «]», «.», «?».

Комментарии: Текст, игнорируемый компилятором/интерпретатором и используемый для документации кода. Существуют такие типы коментарии, как однострочные коментарии и многострочные коментарии. Однострочные коментарии начинаются с преикса «//» и заканчиваются новой строкой, а многострочные коментарии начинаются с префикса «/\*» и заканчиваются суффиксом «\*/»

Токен с пробелами: Содержит в себе символы новой строки, табуляции, пробела, возврата каретки, перевода таблицы и забоя.

Pассмотрим устройство и работу лексического анализатора. Лексический анализатор содержит несколько основных компонентов, среди которых наиболее значимыми являются:

- Сканер: Отвечает за считывание и обработку входного потока символов, а также отвечает за отслеживание информации о позиции токена, выполняет игнорирование незначительных токенов, таких как коментарии и токены с пробелами.

- Таблица токенов: Содержит определения некоторых типов допустимых лексических конструкций языка программирования, таких как ключевые слова, математические операторы, операторы сравнения операторы присваивания, логические операторы, побитовые операторы и логические литералы.

- Конечные автоматы: Используются для описания и распознавания шаблонов лексем.

Рассмотрим процесс лексического анализа, который включает себя несколько этапов, таких как:

- Считывание входного потока: Сканер последовательно считывает входной поток символов, анализирует и запоминает позицию текущего потока, включая позицию символа в потоке, позицию символа в строке и номер линии

- Распознавание лексем: Сканер сопоставляет считанную последовательность символов с определениями лексем в таблице токенов, используя при этом регулярные выражения для ключевых слов, математических операторов, побитовых операторов, логических операторов, операторов сравнения и операторов присвоения и конечные автоматы.

- Формирование токенов: В том случае, если лексема распознана успешно, сканер создает соответствующий токен, с необходимой информацией о типе распознаной лексемы, с содержанием лексемы, с позицией лексемы в коде, и возвращает лексемы

- Обработка ошибок: В том случае, если сканер не может распознать входную последовательность символов как корректную лексему, то сканер вызывает соответствующее исключение.

Также следует отметить, что сканер игнорирует незначимые символы. Сканер может игнорировать пробелы, табуляцию, комментарии и символы управления, которые не несут смысловой нагрузки для обработки генератором байткода.

- Отслеживание позиции в исходном коде: Лексический анализатор может хранить информацию о строке, столбце и позиции в исходном коде, где был обнаружен каждый токен.

- Отображение лексем на токены: Лексический анализатор выполняет преобразование лексем (например, идентификаторов) в соответствующие токены, например, в случае с ключевыми словами или с логическими лексемами. В данном случае при успешном распознавании токена как идентификатора, сканер проверяет на наличие данного идентификатора в таблице токенов, и, в случае если был найден токен, соответствующий идентификатору, то лексеме присваивается тип найденого токена.

Лексический анализатор был реализован на C++ вручную с использованием конструкций языка программирования, операторов перехода, switch case методом рекурсивного спуска, но, лексический анализатор можно реализовать также с помощью инструментов автоматической генерации программного кода, таких как Flex (для C/C++), ANTLR (для Java) и другие, которые позволяют автоматически генерировать лексический анализатор на основе описания лексических правил.

Информация о позиции токенов в исходном коде программы , такая как местоположение токена в исходном коде, включая строку, столбец, смещение от начала файла, необходима для нескольких важных целей, таких как:

Генерация отладочной информации при возникновении ошибок во время компиляции или выполнения программы. Данная информация позволяет построить отладочные сообщения с указанием строки и столбца, где произошла ошибка. Генерация отладочной информации с указанием строки и столбца, а также смещения от начала файла или потока помогает разработчикам быстро находить и устранять проблемы в коде, указывая на строку, столбец и контекст, где была обнаружена ошибка. Также данная информация полезна для среды разработки и систем машинного обучения.

Улучшение сообщений об ошибках. Точная информация о позиции токенов позволяет генерировать более интуитивно распознаваемые сообщения об лексических и синтаксических ошибках, и позволяет получать подробную информацию о том, где именно в коде произошла ошибка, что облегчает процесс устранения проблем.

Информация о позиции токенов также активно используется редакторами кода для подсветки синтаксиса и форматирования программ, что улучшает читаемость и понимание кода разработчиками.

Сведения о позиции токенов позволяют реализовывать функции навигации по коду, такие как переход к определению переменной, функции или класса, что повышает производительность разработчиков при работе с большими кодовыми базами.

- Информация о позиции токенов используется в интерактивных средах разработки (IDE) для предоставления интеллектуальных возможностей, таких как автодополнение, подсказки типов и рефакторинг кода.

Можно вывести мнение о том, что точная информация о позиции токенов в исходном коде является критически важной для построения различных инструментов разработки программного обеспечения. Она позволяет улучшить процесс отладки, повысить читаемость кода и облегчить навигацию по нему.

В данном случае, информация о позиции токенов в исходном коде также активно используется синтаксическим анализатором для генерации дополнительной информации для каждой ветви абстрактного синтаксического дерева. Данная информация, содержащаяся в каждой ветви абстрактного синтаксического дерева является востребованой во время процесса генерации байткода из абстрактного синтаксического дерева, что позволяет дополнить информацию об ошибках как во время компиляции программы, так и во время его исполнения.

Сканер лексического анализатора получает информацию о позиции токена в исходном коде программы следующим образом:

Объект сканера поддерживает внутреннее состояние, хранящее текущую позицию (строку, столбец) в исходном файле, и при анализе входного потока символов сканер последовательно обновляет эту позицию, отслеживая продвижение по строкам и столбцам. Сканер также хранит вектор, содержащий количество символов для каждой обработаной строки для того, чтобы корректно выполнять откат чтения буфера потока или файла.

Объект сканера также активно использует маркеры позиции. При обнаружении начала нового токена, сканер сохраняет текущую позицию в виде специального маркера позиции. Данный маркер ассоциируется с токеном и передается в дальнейший этап лексического анализа.

Когда лексический анализатор идентифицирует и возвращает токен, он также предоставляет информацию о его позиции в исходном коде. Эта информация может включать номер строки, номер столбца, а также смещение токена относительно начала файла.

На основе данной приведенной информации можно сделать вывод о том, что сканер лексического анализатора последовательно отслеживает позицию в исходном коде, сохраняя маркеры позиции для каждого обнаруженного токена. Эта информация о позиции впоследствии используется компилятором или интерпретатором для генерации более точных сообщений об ошибках, отладки и других целей.

**3.1.2 Синтаксический анализатор кода.**

Синтаксический анализатор кода представляет собой компонент компилятора, отвечающий за построение внутреннего представления программы, или абстрактного синтаксического дерева, на основе синтаксической структуры определенного языка программирования.

Перед выполнением работы синтаксическим анализатором, выполняется работа по лексическому анализу кода, для получения последовательного набора лексем, или токенов, опираясь на которых синтаксический анализатор кода будет генерировать абстрактное синтаксическое дерево. Процесс генерации абстрактного синтаксического дерева является важным этапом при компиляции исходного кода программы в байткод, или интерпретации программы.

Абстрактное синтаксическое дерево (сокращенно AST) - это древовидная внутренняя структура данных, в котором внутренние вершины сопоставимы с операторами языка программирования, и каждая ветвь, которая исходит от каждой внутренней вершины абстрактного синтаксического дерева, представляет собой операторы. Листья синтаксического дерева сопоставимы с пустыми операторами, и представляют только переменные и константы. Абстрактное синтаксическое дерево является синтаксической структурой исходного кода программы. Она создается на этапе синтаксического анализа и используется на последующих этапах компиляции.

Крайне важно выделить несколько основных характеристик AST для того, чтобы получить полную картину представления AST, такие как:

Древовидная структура — абстрактное синтаксическое дерево состоит из узлов, которые соответствуют синтаксическим конструкциям языка программирования, такими как описание классов, методов, полей, описание выражения. Связи между узлами отражают иерархию и взаимосвязи между элементами кода.

Абстрактность — AST является обобщеным представлением синтаксической структуры, и не содержит в себе детали, которые не существены при генерации байткода. Таковыми деталями в нашем случае являются, например, коментарии в программном коде. Данная характеристика позволяет работать с текстом программы на более высоком уровне абстракции, что улучшает производительность компилятора.

Также важно отметить, что AST не отражает все особенности конкретного синтаксиса языка программирования. Например, в языке программирования JadeLoom можно представить числовые значения как шестнадцатеричные, восьмеричные, бинарные и десятеричные. В абстрактном синтаксическом дереве не будет учитываться система исчисления числовых значении, так как синтаксический смысл не будет зависеть от системы исчисления, и генератор байткода не оперирует данной информацией, поэтому абстрактное синтаксическое дерево будет хранить только операнд с числовой константой. Или, например, строки в языке программирования JadeLoom могут быть однострочными или многострочными, иметь обозначения специальных символов, такие как обозначения символов Unicode, ASCII, управляющих символов, но, при этом, абстрактное синтаксическое дерево будет содержать информацию только о количестве символов и сам массив символов, отбрасывая информацию о том, каким образом в коде программы был представлен каждый из символов.

Синтаксический анализатор использует грамматику языка программирования JadeLoom, чтобы распознавать правильные синтаксические конструкции и построить соответствующую структуру дерева. Грамматика определяет правила, по которым могут быть построены правильные предложения на языке программирования. Она состоит из терминальных символов (лексемы) и нетерминальных символов (синтаксические конструкции), а также правил вывода, которые определяют, как эти символы могут быть комбинированы.

Процесс генерации абстрактного синтаксического дерева является обработкой входной последовательности токенов и создание каждого соотвествующего узла дерева, в процессе распознавания той или иной синтаксической конструкции в входной последовательности токенов. При генерации узлов абстрактного синтаксического дерева также учитывается информация, содержащаяся в токенах, как позиция токена в исходном коде, тип токена, содержание токена соответствующего типа.

Тип узла, сгенерированого в процессе синтаксического анализа кода, зависит от типа синтаксической конструкции: выражение, оператор, объявление, управляющая конструкция.

Для создания узла анализатор использует специальные методы, определенные для каждого типа узла.

Каждый узел абстрактного синтаксического дерева содержит в себе информацию, необходимую для дальнейшей обработки и интерпретации данной синтаксической конструкции, а также отладки и нахождения ошибок во время генерации байткода и во время исполнения

Информация, содержащееся в каждом узле абстрактного синтаксического дерева, включает в себя тип узла, значение, связанное с узлом, представленым соответствующим токеном.

Каждый узел также содержит ссылки на дочерние узлы, которые могут быть операндами выражения, параметрами функции, методами, конструкциями и полями классов, выражениями для методов и конструкции.

Узел также содержит метаданные, необходимые в процессе выявления ошибок, такие как положение в исходном коде, номер строки и номер столбца

Данная информация позволяет последующим этапам компиляции правильно обработать и проанализировать каждый узел абстрактного синтаксического дерева

Синтаксический анализатор связывает узлы дерева иерархически, отражая структуру входной программы.

**3.2 Разработка генератора байткода для виртуальной машины JVM.**

После завершения разработки лексического и синтаксического анализатора кода, финальным этапом разработки компилятора является разработка генератора байткода на основе абстрактного синтаксического дерева, сгенерированого синтаксическим анализатором кода. Данный генератор байткода был написан на языке программирования Java в следствии необходимости обращаться напрямую к библиотекам, предоставляемым JVM для поиска информации о доступных библиотеках для использования в ходе разработки на языке программирования JadeLoom. Генератор байткода обращается к синтаксическому анализатору через нативный метод, предварительно загрузив необходимую библиотеку с нативным кодом, написанным на языке программирования C++.

Разработка генератора байткода для компилятора - это наиболее значительный и важный этап в создании компилятора языка программирования JadeLoom, и включает в себя несколько подэтапов:

1. Определение промежуточного представления: Необходимо определить формат промежуточного представления байткода, которое будет генерироваться. В данном случае промежуточным представлением является структура файла байткода JVM.

2. Разработка генератора промежуточного кода: Этот компонент генерирует промежуточное представление на основе абстрактного синтаксического дерева, сгенерированного с помощью синтаксического анализатора кода . Каждому элементу абстрактного синтаксического дерева соответствуют одна или несколько инструкций промежуточного кода.

3. Реализация инструкций промежуточного кода: Для каждой инструкции промежуточного кода необходимо определить ее формат и семантику. В данном этапе используется библиотека org.objectweb.asm которая, в свою очередь, является популярным инструментом для работы с байткодом JVM на уровне низкоуровневого программирования. Она предоставляет классы и функции для анализа, создания и преобразования объектов, наследуемых от ClassWriter, напрямую в байткод.

4. Генерация промежуточного кода: Основная логика генератора промежуточного кода заключается в обходе AST и генерации соответствующих инструкций промежуточного представления. Это может включать в себя оптимизации для повышения эффективности.

5. Интеграция с другими компонентами компилятора: Генератор промежуточного кода интегрирован с другими представлениями компилятора, такими как лексический анализатор и синтаксический анализатор.

Промежуточное представление файла байткода JVM включает в себя определения классов, методов, полей, инструкции методов. В первую очередь следует обратить внимание на то, каким образом в программном коде мы можем получить доступ к тому или иному полю, методу или классу. Методы и поля могут быть объявлены в классе в произвольном порядке.

Допустим, что метод „пример\_метода1“ содержит код, в котором был объявлен вызов метода „пример\_метода2“, описание которого находится после описания метода „пример\_метода1“. Генератор байткода JadeLoom должен вызвать ошибку, в котором описано, что генератор не может найти объявленый метод „пример\_метода2“, или должен завершить работу досрочно, если будет обрабатывать и компилировать абстрактное синтаксическое дерево по мере итерации. То же самое можно допустить и при работе с полями, вложеными классами и конструкциями. По этой причине необходимо, чтобы генератор байткода выполнил первый проход по абстрактному синтаксическому дереву для сбора определений всех классов, методов и полей, независимо от порядка их объявления.

Далее необходимо построить внутреннее представление структур классов, в котором все ссылки на классы, методы и поля были бы разрешены. Отследить и разрешить все ссылки на методы, поля и вложенные классы, даже если они объявлены после их использования в коде. Обработать ошибки, связанные с неразрешенными ссылками, и выводить информацию об ошибках любым удобным способом. В нашем случае генератор байткода будет выводить информацию об неразрешенных ссылках в консоль, а также будет выводить информацию о том, где произошла та или иная ошибка с неразрешенными ссылками.

Важно также учесть, что программный код JadeLoom может содержать функции и поля вне класса, но генератор байткода должен выводить только файлы классов, если следовать специфике работы JVM, поэтому необходимо определить стандартный класс, который будет содержать в себе те поля и методы, которые находятся вне классов. Если определение стандартного класса не содержится в программном коде JadeLoom, то генератор байткода должен сгенерировать необходимый класс.

**3.2.1. Поиск перегруженого метода.**

В языке программирования JadeLoom реализуется механизм перегрузки методов, что позволяет создавать несколько методов в одном классе с одинаковыми именами и разными параметрами. Данный механизм создает возможность использовать одно название для разных описании методов, при условии, что методы содержат различное количество параметров или различные типы. Генератор байткода JVM должен уметь работать с перегружеными методами.

Когда компилятор генерирует вызов перегруженного метода в байт-коде, виртуальная машина во время выполнения должна определить, какая именно версия метода должна быть вызвана. Данный процесс поиска состоит из следующих основных шагов:

1. Анализ аргументов вызова:

- Генератор байткода анализирует количество аргументов, переданных в вызов перегруженного метода, и выбирает те методы, количество аргументов которых равна количеству аргументов, переданных в вызов перегруженного метода. Если найдено 0 таких методов, то генератор байткода вызывает исключение, сообщающее программисту о том, что соответствующий метод с соответствующим количеством аргументов не найден

- Необходимо также определить типы аргументов, переданных в вызов перегруженого метода, и их соответствующие сигнатуры и типовые параметры

2. Поиск информации о сигнатурах в метаданных класса:

- Генератор обращается к метаданным класса, где хранится информация о сигнатурах перегруженных методов, сравнивая сигнатуру вызова с доступными сигнатурами методов.

3. Выбор подходящей версии:

- Если генератор находит в метаданных точно совпадающую сигнатуру, то выбирает соответствующую версию метода.

- Если в метаданных нет точного совпадения, генератор применяет правила приведения типов, чтобы найти наиболее подходящую версию. Если подходящих версии более чем 1, или равно 0, то генератор вызывает соответствующее исключение.

Важно отметить, что применяя правила приведения типов, генератор байткода JVM создает матрицу с уровнями совпадения, примерно так же, как показано на таблице 2, где по горизонтали даны названия типов переменных, а по вертикали — перегруженые методы. Уровень совпадения 1 означает, что приведенные типы совпадают полностью, уровень совпадения -1 означает, что приведеные типы не совпадают:

В примере, приведенной на таблице 2, тип А напрямую наследуется от типа Д, поэтому уровень совпадения А к Д равна 2. В данном примере наиболее подходящим методом для переменных с типами (A, Б, В) является метод «Метод(А,Б,В)» потому, что среди представленых методов данных метод содержит параметры, все уровни совпадения которых являются минимальными. Также очень важно выделить, что у подходящего метода все уровни совпадения должны быть минимальными, и хотя бы один уровень совпадения должен быть наименьшим из соответствующего столбца, иначе генератор должен вызвать соответствующее исключение

Таким образом, можно сделать вывод, что процесс поиска правильной версии перегруженного метода во время генерации байткода включает в себя анализ аргументов вызова, поиск подходящей сигнатуры в метаданных класса и применение правил приведения типов для выбора наиболее подходящей версии метода. Примечательно, что, данную задачу поиска подходящего перегруженого метода выполняет именно компилятор, во время генерации байткода на основе абстрактного синтаксического дерева, но не выполняет интерпретатор байткода JVM.

**3.2.2. Приведения типов, уровни наследования.**

В языке программирования JadeLoom реализуются несколько способов приведения типов, которые используются для преобразования переменной или объекта из одного типа данных в другой. Это необходимо, когда тип данных, полученный в результате операции, не соответствует ожидаемому типу, например, при поиске перегруженого класса.

Существует два основных типа приведения типов в Java:

Неявное приведение типов - автоматическое преобразование, выполняемое генератором байткода, когда тип данных меньшего уровня преобразуется в тип данных большего уровня. Генератор байткода реализует следующие приведения для примитивных типов:

((byte → short) | char) → int → long → float → double

Явное приведение типов - преобразование, реализуемое выполняемое программистом, когда тип данных большего уровня преобразуется в тип данных меньшего уровня.

В JadeLoom все объекты ссылочного типа наследуются от java.lang.Object. Любой объект ссылочного типа, а также объект массива, можно преобразовать в java.lang.Object. Поэтому, при вычислении уровня наследования класса от java.lang.Object, в качестве уровня наследования берется число 2^31

Класс в JVM наследуется от единственного класса и множества интерфейсов, в то время, как интерфейс наследуется только от java.lang.Object и множества интерфейсов.

Уровень наследования класса определяется количеством предков , или родительских классов и интерфейсов, в иерархии наследования. В случае, если класс наследуется от интерфейсов, которые, в свою очередь, имеют общих предков, то применяется метод поиска кратчайщего пути для того, чтобы вычислить наименьшее количество предков в иерархии наследования.

Интерфейсы также могут участвовать в наследовании. Интерфейс может наследовать другие интерфейсы, но не может наследовать классы. Уровень наследования интерфейса определяется аналогично классам - количеством интерфейсов-предков в иерархии.

Также важно отметить, что класс не может напрямую наследовать от массива, поэтому при вычислении уровня наследования типа массива от другого типа выполняется проверка на равенство размерности типов. Если типы массивов имеют одинаковые размерности, то вычисляется уровень наследования классов, представляющих массивы.

Примитивные типы данных, такие как byte, short, char, int, long, double, float, boolean, не поддерживают наследование напрямую, и связано это с тем, что примитивные типы являются неизменяемыми, и хранятся непосредственно в памяти, в отличие от объектных типов данных. Для того, чтобы представить примитивные типы в виде ссылочных типов, генератор байткода использует методы классов, так называемых «oболочек», которые, на основе имеющейся информации о переменной примитивного типа, которым является значение, создают объекты ссылочного типа, с соответствующим типом для каждого примитивного типа. Так как все ссылочные типы являются потомками класса java.lang.Object, данные объекты можно назвать объектами ссылочных типов, и для них доступны методы для работы с соответствующими примитивными типами.

Существуют такие ссылочные типы, соответствующее для каждого из примитивных типов, как:

- Integer - для типа int

- Long - для типа long

- Boolean - для типа boolean

- Character - для типа char

- Float - для типа float

- Double - для типа double

- Byte - для типа byte

- Short - для типа short

Оболочечные классы позволяют использовать примитивные типы в ситуациях, где требуются использование объектов ссылочного типа, например, при использовании коллекций или передаче в качестве типизированых аргументов.

Несмотря на то, что оболочечные классы наследуют от Object, они не поддерживают наследование, так как являются финальными классами. Каждый оболочечный класс представляет только один примитивный тип.

Важно отметить, что операции с примитивными типами обычно более эффективны, чем операции с их оболочечными классами, поэтому рекомендуется использовать примитивные типы там, где это возможно, и переходить к оболочечным классам только при необходимости.

**3.2.3. Обработка циклов и операторов перехода.**

При генерации исполнительного кода для методов из абстрактного синтаксического дерева, генератору байткода необходимо корректно обрабатывать операторы перехода и циклы, а также операторы управления потоков. В языке программирования JadeLoom были реализованы такие циклы, как циклы с предусловием for, while, циклы с постусловием do while операторы перехода с условием if, else операторы управления циклом continue, break, и безусловный оператор goto. В то время, как операторы перехода с условием и циклы не обращаются к именованым меткам (меткам, объявленым в блоке кода), операторы управления циклом могут обращаться к именованым меткам, а оператор безусловного перехода обращается только к именованой метке, для нее не генерируется анонимная метка.

При обработке генератором операторов условного и безусловного переходов, циклов, операторов управления циклом, генерируется соответствующий байткод, с начальным адресом метки, равную нулю. Также генератор байткода запоминает адреса, помеченые метками, для дальнейшего заполнения адресами.

Генератор байткода обязательно присваивает каждой метке (именованой или анонимной) соответствующий адрес в блоке кода. Присвоение метки происходит в финальной стадии обработки блока кода, когда все необходимые метки были обнаружены. Не допускается повторное переопределение метки для именованых меток, то есть, в блоке кода не могут присуствовать метки с одинаковыми именами. В то же время, анонимные метки и именованые метки могут иметь одинаковый соответствующий адрес в блоке кода.

После присвоения меткам адресов, генератор байткода редактирует сгенерированый байткод, заменяя поля с соответствующими метками соответствующими адресами.

Существуют такие типы меток, как именованые метки, и анонимные метки. Именованые метки следует объявить в блоке кода, в то время как анонимные метки генерируются автоматически. Метка содержит уникальный идентификатор и имя, если метка является именованой. В том случае, если для операторов перехода goto и операторов управления continue, break не найдена соответствующая метка или если именованая метка была объявлена не единижды, то генератор байткода вызывает соответствующее исключение. Если метка была объявлена единижды, но при этом она не была использована, то генератор байткода вызывает предупреждение о том, что метка не была использована. Именованая метка может быть использована единижды, несколько раз или может быть не использована, в то время, как анонимная метка используется единижды или несколько раз.

Для обработки циклов while и for из абстрактного синтаксического дерева генератор байткода генерирует две анонимные метки: метка выхода из цикла и метка начала цикла.

Метка начала цикла используется для инструкции перехода, которая реализует повторение цикла for или while. Также оно обозначает начало цикла while или for. Метка начала цикла может использоваться оператором continue в том случае, если continue был определен без названия именованой метки и был определен в цикле, для безусловного продолжения цикла.

Meтка выхода из цикла обозначает адрес, где цикл должен завершится, и используется для инструкции перехода, которые выводят из цикла, когда условие цикла не является истиной. Метка выхода может использоваться оператором break в том случае, если break был определен без названия именованой метки и был определен в цикле, для выхода из цикла.

Для обработки циклов с постусловием do из абстрактного синтаксического дерева генератор байткода генерирует только одну анонимную метку, означающее начало цикла, и использующаяся для инструкции перехода, которая выполняет переход в случае, если условие цикла является истиной. Метка начала цикла с постусловием может использоваться оператором continue в том случае, если continue был определен без названия именованой метки и был определен в цикле, для безусловного продолжения цикла.

В том случае, если внутри цикла do используется оператор break без названия именованой метки, генерируется еще одна анонимная метка, означающая адрес, где цикл должен завершится.

Для обработки операторов с условием if генератор байткода генерирует одну метку, означающая конец выполнения кода с условием, и использующаяся для инструкции перехода, которая выполняет переход в случае, если условие оператора является ложной. В случае, если имеется конструкция if else, то для данной конструкции генерируется две метки, одна метка которая означает конец выполнения блока кода else, и переход к которой выполняется после завершения блока кода if, и метка начала блока кода else, переход к которой выполняется в том случае, если условие оператора if является ложной.

**3.2.4. Оптимизация исполнительного кода.**

При необходимости генератор байткода сокращает инструкции для того, чтобы оптимизировать выходной байткод и улучшить производительность и эффективность программ, написаных на языке программирования JadeLoom.

В ходе реализации генератора байткода был реализован функционал, который заменяет выражение, не изменяющее своего значения во время выполнения программы, на из непосредственное значение. Арифметические или строковые операции с константами заменяются на их результат

В языке программирования JadeLoom используются финальные переменные в блоке кода, которые объявляются ключевым словом final и обязательно должны быть инициализованы. Если для инициализации финальной переменной используется константное выражение, или выражение, которое можно заменить константой, то генератор байткода не генерирует инструкцию, которая должна сохранять значение финальной переменной в определенной ячейке памяти, а использует ее как константу. Данный подход позволяет значительно оптимизировать производительность программ и сократить набор генерируемых инструкции. Также, при возможности, генератор байткода заменяет сложные инструкции на эквивалентные, но более простыми инструкциями. Например, при умножении целочисленого числа на число, равному 2^x, генератор байткода заменяет выражение на эквивалентное, побитовое смещение.

Генератор байткода может пропустить участки кода, которые никогда не выполняются, но при этом вызывает соответствующее предупреждение. Генератор анализирует код, чтобы определить, какие участки никогда не будут выполнены, что особенно актуально при наличии условных операторов и операторов перехода где некоторые участки кода могут быть неиспользуемыми.

Также, при необходимости, генератор устраняет виртуальные вызовы, заменяя их на прямые. Виртуальные вызовы методов связаны с дополнительными накладными расходами, так как они требуют поиска правильной реализации метода в иерархии классов. Компилятор может заменять виртуальные вызовы на прямые, если тип объекта известен на этапе компиляции.

Генератор анализирует код, чтобы найти повторяющиеся вычисления одних и тех же выражений. Он заменяет такие повторяющиеся выражения на ссылки на единственное вычисленное значение, что позволяет избежать некоторые вычисления и улучшает эффективность выполнения кода.

ЗАКЛЮЧЕНИЕ

В ходе выполнения данной выпускной квалификационной работы был разработан новый высокоуровневый типизированный язык программирования JadeLoom, построенный на основе виртуальной машины JVM. Основными целями разработки языка JadeLoom являлись упрощение разработки программного обеспечения и улучшение производительности за счет использования преимуществ статической типизации и JVM.

В рамках выполнения выпускной работы были реализованы основные компоненты языка JadeLoom, включая, но не ограничиваясь ими, синтаксический анализатор, систему приведения типов, средства компиляции в байт-код JVM. Было проведено сравнение с аналогичными языками, работающими на JVM.

Результаты показали, что JadeLoom обеспечивает более простой и понятный синтаксис по сравнению с Java, при этом демонстрируя сравнимую или даже лучшую производительность за счет эффективной компиляции в байт-код JVM. Это достигается благодаря использованию статической типизации, которая позволяет выполнять более агрессивную оптимизацию кода на этапе компиляции.

Разработанный язык JadeLoom может быть использован в качестве альтернативы Java или Kotlin для реализации широкого спектра приложений, работающих на JVM. Он обеспечивает упрощенный синтаксис, строгую типизацию и высокую производительность, что делает его привлекательным выбором для разработчиков, стремящихся к повышению эффективности и качества своего программного обеспечения.

Дальнейшее развитие языка JadeLoom может включать в себя расширение его функциональности, улучшение интеграции с существующими библиотеками и фреймворками Java, а также оптимизацию компилятора для достижения еще более высокой производительности.

СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ

1. JSR-000924 Java® Virtual Machine Specification ("Specification"): [Электронный ресурс]: Oracle Inc. and/or its affiliates, 2001, Режим доступа: [https://docs.oracle.com](https://docs.oracle.com/)[/javase/specs/jvms/](https://docs.oracle.com/javase/specs/jvms/)[se7/html/jvms-6.html](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-6.html). (Дата обращения: 21.03.2024).
2. JSR-000924 Java® Virtual Machine Specification ("Specification"): [Электронный ресурс]: Oracle Inc. and/or its affiliates, 2001, Режим доступа: [https://docs.oracle.com](https://docs.oracle.com/)[/javase/specs/jvms/](https://docs.oracle.com/javase/specs/jvms/)[se7/html/jvms-](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-6.html)4.html. (Дата обращения: 21.03.2024).
3. Список инструкции JVM [Электронный ресурс]: Википедия – Режим доступа: <https://en.wikipedia.org/wiki/List_of_Java_bytecode_instructions> (Дата обращения: 23.03.2024).
4. Ажмухамедов И. М., Кузнецова В. Ю., Зорин К. А. Структура программного продукта для семантического анализа текстовой информации [Электронный ресурс]: CyberLeninka — Режим доступа: <https://cyberleninka.ru/article/n/struktura-programmnogo-produkta-dlya-semanticheskogo-analiza-tekstovoy-informatsii/viewer> (Дата обращения: 11.04.2024).
5. Вартанов С. П. Ермаков М. К. Применение статической инструментации байт-кода языка Java для динамического анализа программ [Электронный ресурс]: CyberLeninka — Режим доступа: <https://cyberleninka.ru/article/n/primenenie-staticheskoy-instrumentatsii-bayt-koda-yazyka-java-dlya-dinamicheskogo-analiza-programm/viewer> (Дата обращения: 27.04.2024).

ПРИЛОЖЕНИЯ.

Таблица 1. Имена, назначения и байтовые выражения опкодов

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Mнемоника | Стек [до] -> [после] | Хеш-представление | Операнды  [число байтов]: [метки] | Описание |
| aaload | массив, индекс → значение | 32 |  | Загрузить в стек ссылку из массива |
| aastore | массив, индекс, значение → | 53 |  | Хранить ссылку в массиве |
| aconst\_null | → ноль | 1 |  | Загрузить нулевую ссылку на стек |
| aload | → объект | 19 | 1:индекс | Загрузить ссылку на стек из локальной переменной #индекс |
| aload\_0 | → объект | 2a |  | Загрузить ссылку на стек из локальной переменной 0 |
| aload\_1 | → объект | 2b |  | Загрузить ссылку на стек из локальной переменной 1 |
| aload\_2 | → объект | 2c |  | Загрузить ссылку на стек из локальной переменной 2 |
| aload\_3 | → объект | 2d |  | Загрузить ссылку на стек из локальной переменной 3 |
| anewarray | количество →массив | bd | 2:индекс | Создать новый массив ссылок на счет длины и типа компонента, идентифицированные по эталонному индексу класса (indexbyte1 << 8 | indexbyte2) в постоянном пуле |
| areturn | объект → [пусто] | b0 |  | Вернуть ссылку из метода |
| arraylength | массив → length | be |  | Получить длину массива |
| astore | объект → | 3a | 1:индекс | Сохранить ссылку в локальную переменную #iиндекс |
| astore\_0 | объект → | 4b |  | Сохранить ссылку в локальную переменную 0 |
| astore\_1 | объект → | 4c |  | Сохранить ссылку в локальную переменную 1 |
| astore\_2 | объект → | 4d |  | Сохранить ссылку в локальную переменную 2 |
| astore\_3 | объект → | 4e |  | Сохранить ссылку в локальную переменную 3 |
| athrow | объект → [пусто], объект | bf |  | Вызвать исключение (следует обратить внимание, что остальная часть стека очищена, оставляя лишь ссылку на бросок) |
| baload | массив, индекс→ значение | 33 |  | Загрузить байт или логическое значение из массива |
| bastore | массив, индекс, значение → | 54 |  | Сохранить байт или логическое значение в массиве |
| bipush | → значение | 10 | 1:байт | Загрузить байт на стек в качестве целочисленного значения |
| breakpoint |  | ca |  | Зарезервировано |
| caload | массив, индекс → значение | 34 |  | Загрузить символ из массива |
| castore | массив, индекс, значение → | 55 |  | Сохранить символ в массив |
| checkcast | объект → объект | c0 | 2:индекс | Проверяет, является ли objectref определенного типа, ссылка на класс которого находится в постоянном пуле в индексе (indexbyte1 << 8 | indexbyte2) |
| d2f | значение → результат | 90 |  | Преобразовать double в float |
| d2i | значение → результат | 8e |  | Преобразовать double в int |
| d2l | значение → результат | 8f |  | Преобразовать double в long |
| dadd | значение1, значение2 → результат | 63 |  | Сложить double |
| daload | массив, индекс → значение | 31 |  | Загрузить double из массива |
| dastore | массив, индекс, значение → | 52 |  | Сохранить double в массив |
| dcmpg | значение1, значение2 → результат | 98 |  | Сравните два double, 1 на NAN |
| dcmpl | значение1, значение2 → результат | 97 |  | Сравнить два double, -1 на NAN |
| dconst\_0 | → 0.0 | 0e |  | Загрузить постоянный 0,0 в стек |
| dconst\_1 | → 1.0 | 0f |  | Загрузить постоянный 1.0 в стек |
| ddiv | значение1, значение2 → результат | 6f |  | Разделить два doulbe |
| dload | → значение | 18 | 1:индекс | Загрузить double из локальной переменной #index |
| dload\_0 | → значение | 26 |  | Загрузить double из локальной переменной 0 |
| dload\_1 | → значение | 27 |  | Загрузить double из локальной переменной 1 |
| dload\_2 | → значение | 28 |  | Загрузите double из локальной переменной 2 |
| dload\_3 | → значение | 29 |  | Загрузить double из локальной переменной 3 |
| dmul | значение1, значение2 → результат | 6b |  | Умножить два double |
| dneg | значение → результат | 77 |  | Инвертировать double |
| drem | значение1, значение2 → результат | 73 |  | Получить остаток от деления между двумя double |
| dreturn | значение → [пусто] | af |  | Вернуть double |
| dstore | значение → | 39 | 1:индекс | Хранить double в локальной переменной #индекс |
| dstore\_0 | значение → | 47 |  | Хранить double в локальную переменную 0 |
| dstore\_1 | значение → | 48 |  | Хранить double в локальную переменную 1 |
| dstore\_2 | значение → | 49 |  | Хранить double в локальную переменную 2 |
| dstore\_3 | значение → | 4a |  | Хранить double в локальную переменную 3 |
| dsub | значение1, значение2 → результат | 67 |  | Вычесть double из другого double |
| dup | значение → значение, значение | 59 |  | Дублировать значение в верхней части стека |
| dup\_x1 | значение2, значение1 → значение1, значение2, значение1 | 5a |  | Вставьте копию верхнего значения в стек два значения сверху.Значение1 и значение2 не должны быть типа double или long. |
| dup\_x2 | значение3, значение2, значение1 → значение1, значение3, значение2, значение1 | 5b |  | Вставьте копию верхнего значения в стек два (если значение2 double или long, он также занимает запись значения3) или три значения (если значение2 не является ни double, ни longg) сверху |
| dup2 | {значение2, значение1} → {значение2, значение1}, {значение2, значение1} | 5c |  | Дублировать два верхних слова стека (два значения, если значение1 не является double, ни long; одно значение, если значение1 - double или long) |
| dup2\_x1 | значение3, {значение2, значение1} → {значение2, значение1}, значение3, {значение2, значение1} | 5d |  | Дублировать два слова и вставить под третьим словом (см. Объяснение выше) |
| dup2\_x2 | {значение4, значение3}, {значение2, значение1} → {значение2, значение1}, {значение4, значение3}, {значение2, значение1} | 5e |  | Дублировать два слова и вставить под четвертое слово |
| f2d | значение → результат | 8d |  | Преобразовать float в double |
| f2i | значение → результат | 8b |  | Преобразовать float в int |
| f2l | значение → результат | 8c |  | Преобразовать float в long |
| fadd | значение1, значение2 → результат | 62 |  | Сложить два float |
| faload | массив, индекс → значение | 30 |  | Загрузить float из массива |
| fastore | массив, индекс, значение → | 51 |  | Сохранить float в массиве |
| fcmpg | значение1, значение2 → результат | 96 |  | Сравнить два float, 1 на NAN |
| fcmpl | значение1, значение2 → результат | 95 |  | Сравнить два float, -1 на NAN |
| fconst\_0 | → 0.0f | 0b |  | Загрузить 0,0F в стек |
| fconst\_1 | → 1.0f | 0c |  | Загрузить 1,0F в стек |
| fconst\_2 | → 2.0f | 0d |  | Загрузить 2,0F в стек |
| fdiv | значение1, значение2 → результат | 6e |  | Разделить два float |
| fload | → значение | 17 | 1:индекс | Загрузить значение float из локальной переменной #индекс |
| fload\_0 | → значение | 22 |  | Загрузите значение float из локальной переменной 0 |
| fload\_1 | → значение | 23 |  | Загрузите значение float из локальной переменной 1 |
| fload\_2 | → значение | 24 |  | Загрузите значение float из локальной переменной 2 |
| fload\_3 | → значение | 25 |  | Загрузите значение float из локальной переменной 3 |
| fmul | значение1, значение2 → результат | 6a |  | Умножьте два float |
| fneg | значение → результат | 76 |  | Инвертировать float |
| frem | значение1, значение2 → результат | 72 |  | Получить остаток от деления между двумя float |
| freturn | значение → [пусто] | ae |  | Вернуть float |
| fstore | значение → | 38 | 1:индекс | Сохранить значение float в локальной переменной #iиндекс |
| fstore\_0 | значение → | 43 |  | Сохранить значение float в локальную переменную 0 |
| fstore\_1 | значение → | 44 |  | Сохранить значение float в локальную переменную 1 |
| fstore\_2 | значение → | 45 |  | Сохранить значение float в локальную переменную 2 |
| fstore\_3 | значение → | 46 |  | Сохранить значение float в локальную переменную 3 |
| fsub | значение1, значение2 → результат | 66 |  | Вычесть два float |
| getfield | объект → значение | b4 | 2:индеск | Получить значение поля объекта objectref, где поле идентифицируется по полевым ссылкам в индексе постоянного пула (Indexbyte1 << 8 | indexbyte2) |
| getstatic | → значение | b2 | 2:индекс | Получить статическое значение поля класса, где поле идентифицируется по полевым ссылкам в индексе постоянного пула (indexbyte1 << 8 | indexbyte2) |
| goto | [не изменено] | a7 | 2:адрес | Сместить курсор инструкции на указаный адрес |
| goto\_w | [не изменено] | c8 | 4:адрес | Сместить курсор инструкции на указаный адрес |
| i2b | значение → результат | 91 |  | Преобразовать int в byte |
| i2c | значение → результат | 92 |  | Преобразовать int в char |
| i2d | значение → результат | 87 |  | Преобразовать int в double |
| i2f | значение → результат | 86 |  | Преобразовать int в float |
| i2l | значение → результат | 85 |  | Преобразовать int в long |
| i2s | значение → результат | 93 |  | Преобразовать int в short |
| iadd | значение1, значение2 → результат | 60 |  | Сложить два int |
| iaload | массив, индекс → значение | 2e |  | Загрузить int из массива |
| iand | значение1, значение2 → результат | 7e |  | Bыполнить бинарную операцию and над двумя int |
| iastore | массив, индекс, значение → | 4f |  | Соранить Int в массив |
| iconst\_m1 | → -1 | 2 |  | Загрузить значение int -1 в стек |
| iconst\_0 | → 0 | 3 |  | Загрузить значение int 0 в стек |
| iconst\_1 | → 1 | 4 |  | Загрузить значение int 1 в стек |
| iconst\_2 | → 2 | 5 |  | Загрузить значение int 2 в стек |
| iconst\_3 | → 3 | 6 |  | Загрузить значение int 3 в стек |
| iconst\_4 | → 4 | 7 |  | Загрузить значение int 4 в стек |
| iconst\_5 | → 5 | 8 |  | Загрузить значение int 5 в стек |
| idiv | значение1, значение2 → результат | 6c |  | Разделить два целых числа |
| if\_acmpeq | значение1, значение2 → | a5 | 2:адрес | Если две ссылки равны, выполнить переход по адресу |
| if\_acmpne | значение1, значение2 → | a6 | 2:адрес | Если две ссылки не равны, выполнить переход по адресу |
| if\_icmpeq | значение1, значение2 → | 9f | 2:адрес | Если два значения int равны, выполнить переход по адресу |
| if\_icmpge | значение1, значение2 → | a2 | 2:адрес | Если int значение1 больше или равное int значению2, выполнить переход по адресу |
| if\_icmpgt | значение1, значение2 → | a3 | 2:адрес | Если int значение1 больше, чем значение2, выполнить переход по адресу |
| if\_icmple | значение1, значение2 → | a4 | 2:адрес | Если int значение1 меньше или равное значению2, выполнить переход по адресу |
| if\_icmplt | значение1, значение2 → | a1 | 2:адрес | Если int значение1 меньше, чем значение2, выполнить переход по адресу |
| if\_icmpne | значение1, значение2 → | a0 | 2:адрес | Если два значения int не равны, выполнить переход по адресу |
| ifeq | значение → | 99 | 2:адрес | Если int значение равно 0, выполнить переход по адресу |
| ifge | значение → | 9c | 2:адрес | Если int значение больше или равное 0, выполнить переход по адресу |
| ifgt | значение → | 9d | 2:адрес | Если int значение больше 0, выполнить переход по адресу |
| ifle | значение → | 9e | 2:адрес | Если int значение меньше или равное 0, выполнить переход по адресу |
| iflt | значение → | 9b | 2:адрес | Если int значение меньше 0, выполнить переход по адресу |
| ifne | значение → | 9a | 2:адрес | Если значение не 0, выполнить переход по адресу |
| ifnonnull | значение → | c7 | 2:адрес | Если ссылка не является нулевым, выполнить переход по адресу |
| ifnull | значение → | c6 | 2:адрес | Если ссылка является нулевым, выполнить переход по адресу |
| iinc | [не изменено] | 84 | 2:адрес | Увеличить локальную переменную #индекс на 1 |
| iload | → значение | 15 | 1:индек | Загрузить значение int из локальной переменной #индекс |
| iload\_0 | → значение | 1a |  | Загрузить значение int из локальной переменной 0 |
| iload\_1 | → значение | 1b |  | Загрузить значение int из локальной переменной 1 |
| iload\_2 | → значение | 1c |  | Загрузить значение int из локальной переменной 2 |
| iload\_3 | → значение | 1d |  | Загрузить значение int из локальной переменной 3 |
| impdep1 |  | fe |  | Зарезервировано для зависимых от реализации операций в рамках отладчиков |
| impdep2 |  | ff |  | Зарезервировано для зависимых от реализации операций в рамках отладчиков |
| imul | значение1, значение2 → результат | 68 |  | Умножить два числа int |
| ineg | значение → результат | 74 |  | Инвертировать int |
| instanceof | объект → результат | c1 | 2:индекс | Определяет, имеет ли объект objectref определенный тип, идентифицированный по индексу класса в пуле констант |
| invokedynamic | [аргумент1, аргумент2, ...] → результат | ba | 4:индекс,индекс | Вызывает динамический метод и помещает результат в стек; |
| invokeinterface | объект, [аргумент1, аргумент2, ...] → результат | b9 | 4:индекс,индекс | Вызывает метод интерфейса на объекте objectref и помещает результат в стек |
| invokespecial | объект, [аргумент1, аргумент2, ...] → результат | b7 | 2:индекс,индекс | Вызывает метод экземпляра на объекте objectref и помещает результат в стек |
| invokestatic | [аргумент1, аргумент2, ...] → результат | b8 | 2:индекс | Вызывает статический метод и помещает результат в стек |
| invokevirtual | объект, [аргумент1, аргумент2, ...] → результат | b6 | 2:индекс | Вызывает виртуальный метод на объекте objectref и размещает результат в стек |
| ior | значение1, значение2 → результат | 80 |  | Выполнить бинарную операцию or над двумя int |
| irem | значение1, значение2 → результат | 70 |  | Вычислить остаток от деления двyх int |
| ireturn | значение → [пусто] | ac |  | Вернуть целое число |
| ishl | значение1, значение2 → результат | 78 |  | Арифметический сдвиг слева для int |
| ishr | значение1, значение2 → результат | 7a |  | Арифметический сдвиг справа для int |
| istore | значение → | 36 | 1:индекс | Сохранить значение int в переменную #индекс |
| istore\_0 | значение → | 3b |  | Сохранить значение int в переменную 0 |
| istore\_1 | значение → | 3c |  | Сохранить значение int в переменную 1 |
| istore\_2 | значение → | 3d |  | Сохранить значение int в переменную 2 |
| istore\_3 | значение → | 3e |  | Сохранить значение int в переменную 3 |
| isub | значение1, значение2 → результат | 64 |  | Выполнить вычитание с двумя int |
| iushr | значение1, значение2 → результат | 7c |  | Логический сдвиг справа для int |
| ixor | значение1, значение2 → результат | 82 |  | Выполнить бинарную операцию xor над двумя int |
| jsr\_t | → address | a8 | 2:адрес | Выполнить переход по адресу из переменной индекс и поместите адрес в стек. |
| jsr\_wt | → address | c9 | 4:адрес | Выполнить переход по адресу из переменной индекс и поместите адрес в стек. |
| l2d | значение → результат | 8a |  | Преобразовать long в double |
| l2f | значение → результат | 89 |  | Преобразовать long в float |
| l2i | значение → результат | 88 |  | Преобразовать long в int |
| ladd | значение1, значение2 → результат | 61 |  | Выполнить сложение между двумя long |
| laload | массив, индекс → значение | 2f |  | Загрузить long из массива |
| land | значение1, значение2 → результат | 7f |  | Bыполнить бинарную операцию and над двумя int |
| lastore | массив, индекс, значение → | 50 |  | Сохранить long в массив |
| lcmp | значение1, значение2 → результат | 94 |  | Вернуть 0 если два long одинаковы, 1 eсли первое больше второго и -1 в противном случае |
| lconst\_0 | → 0l | 9 |  | Загрузить 0L в стек |
| lconst\_1 | → 1l | 0a |  | Загрузить 1L в стек |
| ldc | → значение | 12 | 1:индекс | Загрузить long в стек из пула констант |
| ldc\_w | → значение | 13 | 2:индекс | Загрузить long в стек из пула констант |
| ldiv | → значение | 6d |  | Разделить две long |
| lload | значение1, значение2 → результат | 16 | 1:индекс | Загрузить long значение из локальной переменной #индекс |
| lload\_0 | → значение | 1e |  | Загрузить long из локальной переменной 0 |
| lload\_1 | → значение | 1f |  | Загрузить long из локальной переменной 1 |
| lload\_2 | → значение | 20 |  | Загрузить long из локальной переменной 2 |
| lload\_3 | → значение | 21 |  | Загрузить long из локальной переменной 3 |
| lmul | значение1, значение2 → результат | 69 |  | Умножить два значения long |
| lneg | значение → результат | 75 |  | Инвентировать long |
| lookupswitch | ключ → | ab | 8 |  |
| lor | значение1, значение2 → результат | 81 |  | Выполнить бинарную операцию or над двумя long |
| lrem | значение1, значение2 → результат | 71 |  | Вычислить остаток от деления |
| lreturn | значение → [пусто] | ad |  | Вернуть значение long |
| lshl | значение1, значение2 → результат | 79 |  | Арифметический сдвиг слева для long |
| lshr | значение1, значение2 → результат | 7b |  | Арифметический сдвиг справа для long |
| lstore | значение → | 37 | 1:индекс | Coxpанить значение long в локальной переменной #индекс |
| lstore\_0 | значение → | 3f |  | Coxpанить значение long в локальной переменной 0 |
| lstore\_1 | значение → | 40 |  | Coxpанить значение long в локальной переменной 1 |
| lstore\_2 | значение → | 41 |  | Coxpанить значение long в локальной переменной 2 |
| lstore\_3 | значение → | 42 |  | Coxpанить значение long в локальной переменной 3 |
| lsub | значение1, значение2 → результат | 65 |  | Выполнить вычитание для двyх long |
| lushr | значение1, значение2 → результат | 7d |  | Логический сдвиг справа для long |
| lxor | значение1, значение2 → результат | 83 |  | Выполнить бинарную операцию xor над двумя long |
| monitorenter | объект → | c2 |  | Начать блокировку объекта |
| monitorexit | объект → | c3 |  | Завершить блокировку объекта |
| multianewarray | количество1, [количество2,...] → массив | c5 | 3:индекс,размерность | Создать новый массив заданной размерности и заданного измерения для заданного типа |
| new | → объект | bb | 2:индекс | Создать новый объект заданного типа |
| newarray | количество → массив | bc | 1:тип | Создайте новый массив с элементами заданного примитивного типа |
| nop | [не изменено] | 0 |  | Не выполнять операцию |
| pop | значение → | 57 |  | Отбросить верхнее значение в стеке |
| pop2 | {значение2, значение1} → | 58 |  | Отбросьте два верхних значения в стеке. |
| putfield | объект, значение → | b5 | 2:индекс | Загрузить значение в заданное поле объекта |
| putstatic | значение → | b3 | 2:индекс | Загрузить значение в заданное статическое поле объекта |
| ret | [не изменено] | a9 | 1:индекс | Продолжить выполнение из адреса, взятой из локальной переменной #индекс. |
| return | → [пусто] | b1 |  | Завершить исполнение метода |
| saload | массив, индекс → значение | 35 |  | Загрузить short из массива |
| sastore | массив, индекс, значение → | 56 |  | Сохранить short в массив |
| sipush | → значение | 11 | 2:байт | Сохранить int в стек как значение short |
| swap | значение2, значение1 → значение1, значение2 | 5f |  | Поменять местами два слова в стеке |
| tableswitch |  | aa | 16 | Продолжить выполнение по адресу в таблице при индексе смещения |
| wide | индекс →[то же, что и для соответствующей инструкции] | c4 | 3/5:опкод,индекс | Выполнить OpCode, где OpCode находится либо Iload, Fload, Aload, Lload, Dload, Istore, FSTORE, ASTORE, LSTORE, DSTORE или RET, но индекс составляет 16 бит |

Таблица 2: Пример генерации таблицы совпадении.

|  |  |  |  |
| --- | --- | --- | --- |
|  | А | Б | В |
| Метод(А,Б,В) | 1 | 1 | 1 |
| Метод(Д,Б,В) | 2 | 1 | 1 |
| Метод(А,К,О) | 1 | -1 | -1 |