Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет
имени Н. Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н. Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления (ИУ)»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии (ИУ7)»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К КУРСОВОЙ РАБОТЕ

НА ТЕМУ:

«Разработка компилятора языка программирования Pascal»

Студент ИУ7-22М (Группа) _____ (Подпись, дата) И. А. Цветков (И. О. Фамилия)

Руководитель курсовой работы _____ (Подпись, дата) А. А. Ступников (И. О. Фамилия)

2024 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В современном мире программирование является важнейшим двигателем научно-технического прогресса и инноваций в самых различных областях: от разработки программного обеспечения для мобильных устройств и веб-приложений до создания сложных систем искусственного интеллекта и работы с большими данными. Программирование лежит в основе практически всех технологических достижений, обеспечивая автоматизацию процессов, повышение производительности и создание новых возможностей для решения задач, которые ранее считались невозможными.

Компиляторы играют ключевую роль в этой экосистеме, так как они обеспечивают связь между языками программирования высокого уровня и аппаратными ресурсами компьютеров. Языки программирования высокого уровня, такие как C, Java, Python, позволяют программистам разрабатывать программы, используя более абстрактные понятия и конструкции, которые проще для восприятия человека. Эти языки предоставляют возможность работать с переменными, функциями, объектами, структурами данных и другими высокоуровневыми концепциями, не требуя от разработчика знаний о том, как именно компьютер обрабатывает данные на уровне процессора.

Однако, несмотря на удобство работы с высокоуровневыми языками программирования, компьютерные процессоры не могут непосредственно исполнять код, написанный на этих языках. Процессоры работают только с машинным кодом — набором низкоуровневых инструкций, каждая из которых соответствует конкретной операции, такой как арифметические вычисления, загрузка данных в память или управление потоком исполнения программы. Компилятор берет на себя задачу трансляции высокоуровневого исходного кода в машинный код, который понятен и исполним процессором.

Целью работы является разработка компилятора языка программирования Pascal на языке Python для целевой платформы LLVM(IR) сс использованием генератора синтаксических анализаторов ANTLR4. Для ее достижения необходимо выполнить следующие задачи.

1. Проанализировать предметную область.

2. Определить грамматику языка Pascal.

3. Сгенерировать синтаксическое дерево с помощью ANTL4.

4. Сгенерировать выходной код с помощью LLVM(IR).

1 Аналитическая часть

1.1 Компилятор

Компилятор [1] – это специальная программа, которая выполняет преобразование исходного кода, написанного на высокоуровневом языке программирования, в машинный код или другой низкоуровневый формат, понятный компьютеру. Высокоуровневые языки программирования, такие как C, C++, Java, Python, и другие, позволяют разработчикам описывать логику программ, используя удобные для человека конструкции: переменные, циклы, функции, классы и другие абстрактные структуры. Однако компьютеры не могут напрямую понимать эти конструкции, так как процессоры работают исключительно с низкоуровневыми инструкциями – машинным кодом.

Машинный код [2] – это набор двоичных инструкций, которые выполняются процессором компьютера. Каждая команда машинного кода задает конкретную операцию, такую как сложение двух чисел, перемещение данных между памятью и регистрами или условный переход. Например, операция сложения двух чисел в машинном коде может выглядеть как последовательность битов «`10001001`», где каждая часть инструкции определяет операнд, команду и другие параметры.

Компилятор выполняет важнейшую роль в этом процессе, преобразуя исходный код программы (который понятен программисту) в инструкции, понятные процессору. Таким образом, он решает задачу связи между уровнем абстракции, на котором работает разработчик, и физической машиной, которая исполняет программу. Компилятор обеспечивает возможность программировать на высокоуровневом языке, не заботясь о конкретной архитектуре процессора, операционной системе или особенностях работы с памятью.

Одним из ключевых преимуществ компиляторов является то, что они обеспечивают возможность писать программы, которые могут выполняться на различных аппаратных платформах, не требуя от программиста детального знания о том, как работает каждая из этих платформ. Например, программа, написанная на C, может быть скомпилирована для процессора Intel, ARM или любой другой архитектуры, если для этих архитектур существуют соответствующие компиляторы. Этот процесс называется кросс-компиляцией — компилятор может производить код для целевой архитектуры, отличной от

той, на которой он исполняется.

1.2 Компиляция

Компиляция состоит из нескольких этапов, каждый из которых выполняет важную функцию в процессе преобразования исходного кода в машинный код. Основные этапы содержат следующее.

1. Лексический анализ.
2. Синтаксический анализ.
3. Семантический анализ.
4. Генерация машинного кода.

1.2.1 Лексический анализ

Лексический анализ [3] – это первый этап компиляции, на котором исходный код программы разбивается на минимальные значимые элементы, называемые лексемами.

Важным понятием является понятие токена – это объект, состоящий из двух частей: типа токена, который описывает, что это за лексема (например, "ключевое слово "оператор "идентификатор"), и значения токена, которое представляет саму лексему (например, "int"или "10").

Основная задача этого этапа – преобразовать текст программы в структурированную последовательность токенов, которые затем будут переданы на следующие этапы компиляции (синтаксический и семантический анализы). Лексический анализатор устраняет ненужные элементы, такие как пробелы и комментарии, и выделяет базовые компоненты программы, такие как ключевые слова, операторы, идентификаторы и литералы.

Лексический анализатор обрабатывает исходный код программы как поток символов. Он читает исходный текст по символам, начиная с первого, и создает токены, объединяя символы в осмысленные группы. Этот процесс продолжается до тех пор, пока весь исходный код не будет обработан.

Рассмотрим основные задачи лексического анализа.

1. **азбиение исходного кода на лексемы** – исходный код анализируется и делится на последовательности символов, которые представляют собой токены.

2. **Идентификация типов лексем** -— лексер классифицирует найденные лексемы на основе правил грамматики языка. Например, определяет, является ли последовательность символов ключевым словом, идентификатором или оператором.

3. **Удаление незначащих символов** -— такие символы, как пробелы, табуляции и комментарии, игнорируются, поскольку они не влияют на выполнение программы, но могут присутствовать для улучшения читаемости кода.

4. **Обработка ошибок на уровне лексем** -— лексер может обнаруживать ошибки, связанные с неправильным использованием символов или идентификаторов. Например, использование символов, которые недопустимы в идентификаторах, или отсутствие закрывающей кавычки в строковом литерале.

1.2.2 Синтаксический анализ

Синтаксический анализ [4] – это второй этап компиляции, следующий за лексическим анализом. Его основная цель – проверить грамматическую правильность исходного кода и построить синтаксическое дерево (дерево разбора), представляющее структуру программы. В отличие от лексического анализа, который работает с отдельными лексемами, синтаксический анализ рассматривает отношения между ними и проверяет, соответствуют ли они правилам грамматики языка программирования.

Синтаксический анализатор можно представить как машину, которая берет на вход последовательность токенов и проверяет их на соответствие грамматическим правилам языка программирования. Если код соответствует правилам, то строится синтаксическое дерево. Если обнаруживается ошибка, то синтаксический анализатор сообщает об ошибке.

Рассмотрим основные задачи синтаксического анализа.

1. **Проверка соответствия грамматике** – синтаксический анализатор

проверяет, соответствует ли последовательность токенов правилам грамматики языка программирования. Это позволяет обнаружить синтаксические ошибки, такие как некорректные выражения или отсутствие обязательных символов (например, закрывающих скобок или точек с запятой).

2. **Построение синтаксического дерева** – синтаксический анализатор строит древовидную структуру, которая отражает иерархические отношения между элементами программы. Узлы дерева представляют синтаксические конструкции, такие как выражения, операторы, блоки кода и т. д.

3. **Подготовка к семантическому анализу** – синтаксическое дерево служит основой для следующего этапа компиляции, на котором проверяется семантическая корректность программы.

Синтаксический анализатор основывается на формальной грамматике, которая описывает правила структуры языка программирования. В теории формальных языков часто используется классификация грамматик по иерархии Хомского, которая включает следующие виды.

1. **Контекстно-свободные грамматики** – они определяют правила, по которым можно составлять выражения и операторы. Такие грамматики описываются в виде множества правил, где каждая левая часть правила может быть заменена на правую часть. Контекстно-свободные грамматики часто используются для описания синтаксиса языков программирования.

2. **Контекстно-зависимые грамматики** – в этих грамматиках правила зависят от контекста, в котором они применяются. Такие грамматики редко используются для описания синтаксиса языков программирования, но могут быть полезны для семантического анализа.

1.2.3 Семантический анализ

Семантический анализ [5] – это третий этап компиляции, следующий за синтаксическим анализом. Его основная цель — проверить, правильно ли

программа использует смысловые (семантические) правила языка программирования. В то время как синтаксический анализ проверяет правильность структуры программы (правильность расстановки ключевых слов, операторов и т.д.), семантический анализ проверяет логику и осмысленность программы.

Рассмотрим основные задачи семантического анализа.

1. **Проверка типов данных** – определение того, соответствуют ли операнды и выражения типам, которые допустимы в данном контексте. Например, нельзя складывать строку и число, или присваивать дробное значение переменной целого типа.

2. **Проверка объявлений переменных и функций** – семантический анализатор проверяет, были ли все переменные и функции объявлены перед их использованием. Если переменная используется до объявления или вовсе не объявлена, будет выдана ошибка.

3. **Проверка области видимости (скопа)** – важно убедиться, что переменные и функции используются в правильной области видимости. Переменная, объявленная внутри функции, не должна быть доступна за её пределами, если она не была передана явно.

4. **Проверка логики программы** – например, семантический анализатор может обнаружить недопустимые конструкции, такие как деление на ноль, использование недоступных или освобождённых ресурсов, бесконечные циклы без выхода.

5. **Вывод типов (type inference)** -— в некоторых языках, таких как Haskell или Python, компилятор может автоматически выводить типы переменных на основе контекста, в котором они используются.

6. **Проверка корректности вызовов функций** -— например, семантический анализатор проверяет, что функции вызываются с правильным числом и типом аргументов.

1.2.4 Генерация машинного кода

Генерация машинного кода [6] – это один из заключительных этапов компиляции, на котором исходный код преобразуется в машинный код или

промежуточный код, который может быть выполнен на целевой аппаратной платформе. Этот этап следуют после семантического анализа, где уже проверено, что программа логически и синтаксически корректна. Генерация кода включает в себя преобразование высокоуровневых абстракций в инструкции, которые могут быть непосредственно исполнены процессором или виртуальной машиной.

Рассмотрим основные задачи генерации кода.

1. Преобразование промежуточного представления в машинный код – на этом этапе промежуточное представление программы (например, в виде трехадресных инструкций или промежуточного языка) преобразуется в машинный код, который может быть исполнен процессором.

2. Оптимизация кода – генерируемый код может быть дополнительно оптимизирован для повышения его эффективности. Оптимизация может включать улучшение скорости выполнения, уменьшение размера кода или снижение потребления ресурсов.

3. Генерация кода для различных платформ – если компилятор поддерживает кросс-компиляцию, он должен сгенерировать код, который будет работать на различных аппаратных платформах.

1.3 ANTLR4

ANTLR (ANother Tool for Language Recognition) [7] – это мощный инструмент для создания парсеров и лексеров, используемый для работы с языками программирования, скриптовыми языками и любыми другими текстовыми форматами. ANTLR4 — это четвёртая версия ANTLR, которая улучшает и расширяет возможности предыдущих версий. Он используется для генерации парсеров, которые могут обрабатывать текстовые данные и преобразовывать их в структуры, понятные компьютеру.

Основные компоненты ANTLR4.

1. Грамматика (Grammar):

   (a) Лексическая грамматика (Lexer grammar): определяет, как исходный текст разбивается на токены. Токены — это основные единицы

текста, такие как ключевые слова, идентификаторы, операторы и литералы.

   (b) Синтаксическая грамматика (Parser grammar): определяет, как токены должны быть организованы в более сложные структуры, такие как выражения и операторы. Это описывает структуру языка.

2. Лексер (Lexer): преобразует входной текст в поток токенов. Он использует правила лексической грамматики для определения типа каждого токена и его значения.

3. Парсер (Parser): принимает поток токенов от лексера и преобразует его в синтаксическое дерево или абстрактное синтаксическое дерево (AST) на основе синтаксической грамматики.

4. Синтаксическое дерево (Parse Tree): полное дерево, отражающее все синтаксические детали входного текста. Оно включает в себя все токены и их отношения.

5. Абстрактное синтаксическое дерево (AST): упрощённое представление синтаксического дерева, которое сохраняет только те детали, которые важны для дальнейшей обработки.

Возможности ANTLR4.

1. Поддержка нескольких языков: ANTLR4 поддерживает генерацию парсеров на различных языках, таких как Java, C#, Python, JavaScript, C++, и других.

2. Поддержка различных типов грамматик: ANTLR4 позволяет работать с контекстно-свободными грамматиками и расширениями, такими как контекстно-зависимые грамматики.

3. Простота интеграции: Сгенерированные лексеры и парсеры можно легко интегрировать в существующие проекты, обеспечивая гибкость и масштабируемость.

4. Оптимизация и отладка: ANTLR4 предоставляет инструменты для отладки и оптимизации грамматик и парсеров, что упрощает разработку и устранение ошибок.

5. Гибкость в настройке: Пользователи могут настраивать лексеры и парсеры, изменяя поведение генератора кода и обработчиков ошибок.

1.4 LLVM(IR)

LLVM (Low-Level Virtual Machine) [8] – это инфраструктура для разработки компиляторов, которая предоставляет мощные инструменты для создания, оптимизации и генерации машинного кода. Основным компонентом LLVM является его промежуточное представление (Intermediate Representation, IR), которое служит мостом между высокоуровневыми языками программирования и машинным кодом.

Основные компоненты LLVM.

1. LLVM IR (Intermediate Representation) – это промежуточный язык, который компиляторы используют для описания программного кода в унифицированном формате. Он находится между высокоуровневыми языками программирования и низкоуровневыми машинными инструкциями.

2. LLVM Core Libraries – библиотеки предоставляют основные функции для работы с LLVM IR, включая анализ, оптимизацию и генерацию кода.

3. LLVM Backend – компонент отвечает за преобразование LLVM IR в машинный код для конкретной архитектуры. Он включает в себя наборы инструментов для генерации кода, оптимизации и выработки целевого кода.

4. LLVM Frontend – инструменты и библиотеки, которые преобразуют исходный код высокоуровневых языков в LLVM IR. Это включает в себя парсеры, лексеры и синтаксические анализаторы.

Основные особенности LLVM IR.

1. **Многоуровневая абстракция**: LLVM IR предоставляет абстракцию, которая находится между высоким уровнем языков программирования и машинным кодом. Это позволяет писать компиляторы, которые работают с одной и той же промежуточной репрезентацией, независимо от целевой платформы.

2. **Типизация**: LLVM IR является строго типизированным. Он поддерживает различные типы данных, такие как целые числа, вещественные числа, указатели и структуры.

3. **Модульность**: LLVM IR поддерживает модульную организацию кода, что упрощает управление зависимостями и оптимизацию.

4. **Глобальные** и локальные переменные: Переменные в LLVM IR могут быть глобальными или локальными и могут использоваться для хранения промежуточных данных.

5. **Функции и вызовы функций**: Функции в LLVM IR могут быть объявлены и определены, и их можно вызывать из других функций.

6. **Управляющие конструкции**: LLVM IR поддерживает условные операторы (br для ветвления), циклы и другие управляющие конструкции.

7. **Целевые архитектуры**: LLVM IR может быть преобразован в машинный код для различных целевых архитектур, таких как x86, ARM, MIPS и других.

2 Конструкторская часть

2.1 Концептуальная модель компилятора в нотации IDEF0

Для создания функциональной модели компилятора, отражающей его основные этапы, наиболее наглядно использовать нотацию IDEF0. На рисунке 2.1 приведена концептуальная модель компилятора. На рисунке 2.2 представлена детализированная концептуальная модель компилятора в нотации IDEF0.

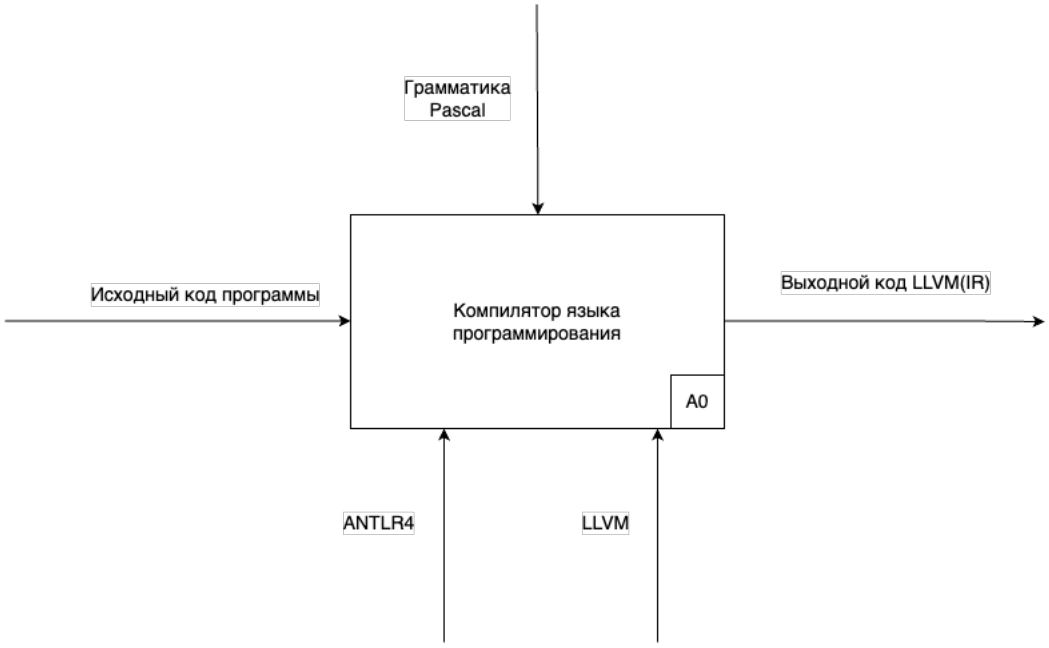

Рисунок 2.1 – Концептуальная модель компилятора в нотации IDEF0

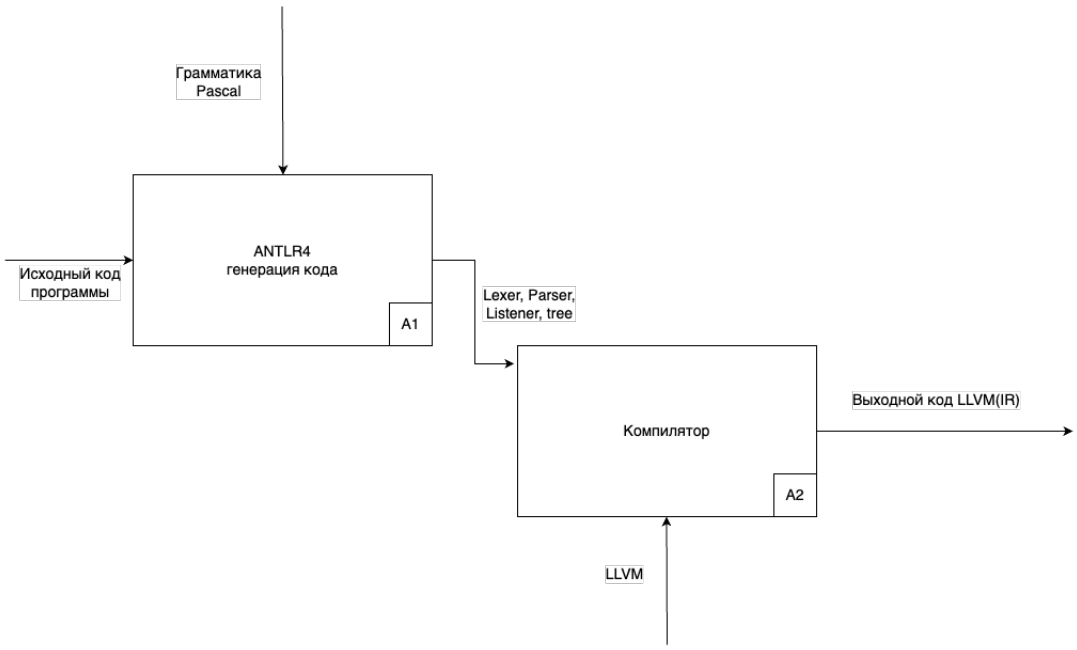

Рисунок 2.2 – Детализированная концептуальная модель компилятора в нотации IDEF0

2.2 Обход AST

Обход абстрактного синтаксического дерева (AST) – это ключевая операция в различных задачах компиляции и анализа программ, таких как интерпретация, оптимизация и трансформация кода. AST представляет собой иерархическую структуру, где узлы дерева соответствуют конструкциям исходного кода, а его ветви отражают синтаксическую структуру программы.

Для обхода абстрактного синтаксического дерева, созданного с помощью ANTLR, применяется паттерн слушателя. ANTLR автоматически генерирует интерфейс слушателя, который связан с встроенным классом, осуществляющим обход дерева. Слушатель предоставляет методы «`enter`» и «`exit`», которые вызываются при входе и выходе из узлов дерева соответственно.

Интерфейс слушателя, создаваемый ANTLR, зависит от грамматики, заданной в проекте. Он включает методы «`enter`» и «`exit`» для каждого правила грамматики, которые необходимо реализовать в соответствии с требованиями проекта.

2.3 Генерация выходного кода LLVM(IR)

Чтобы создать код LLVM IR, нужно разработать алгоритм, который обходит каждую вершину дерева разбора, полученного с помощью парсера ANTLR, и генерирует соответствующий LLVM IR код в зависимости от типа этой вершины.

Общая схема алгоритма для генерации кода LLVM(IR) для компилятора, который умеет работать с цикалми, инициализацией и условными выражениями, приведена на рисунках 2.3-2.5.

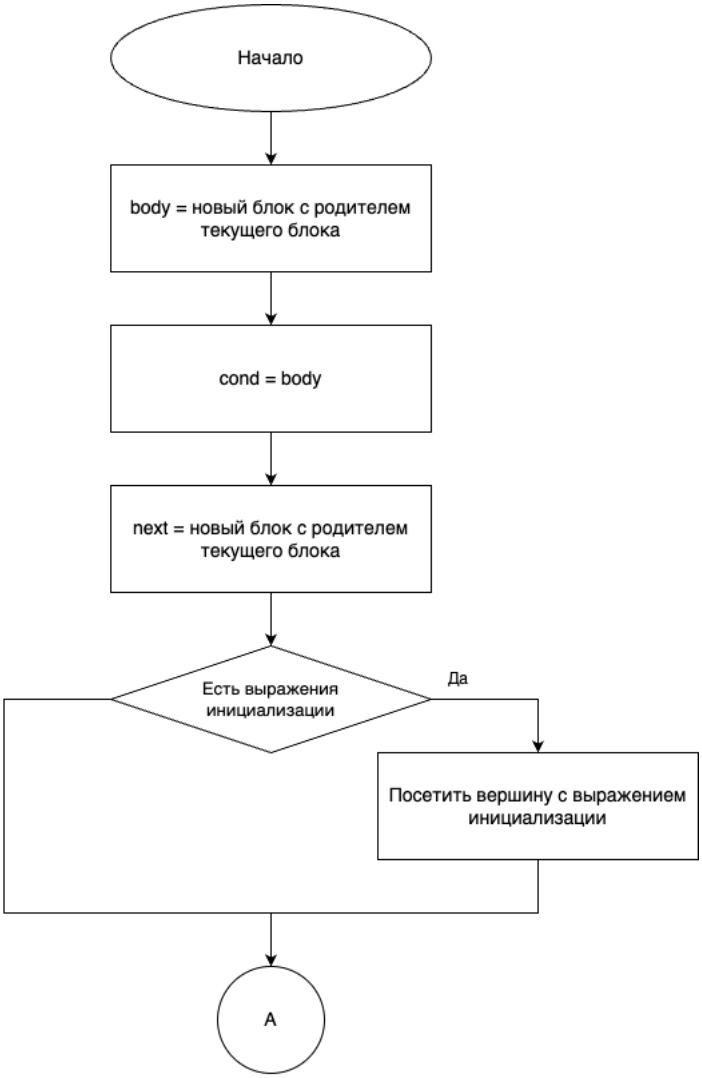

Рисунок 2.3 – Схема алгоритма генерации LLVM(IR) (1)

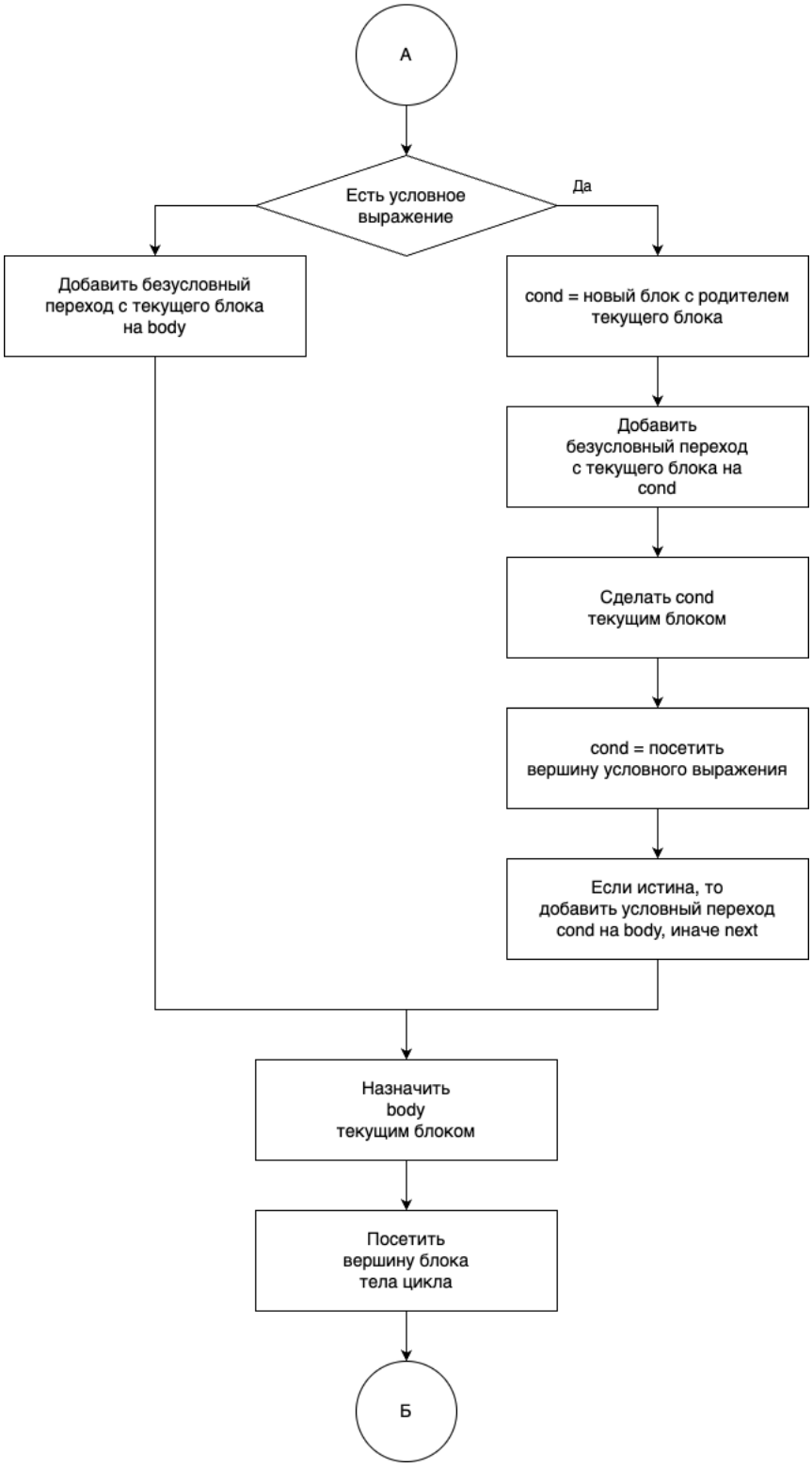

Рисунок 2.4 – Схема алгоритма генерации LLVM(IR) (2)

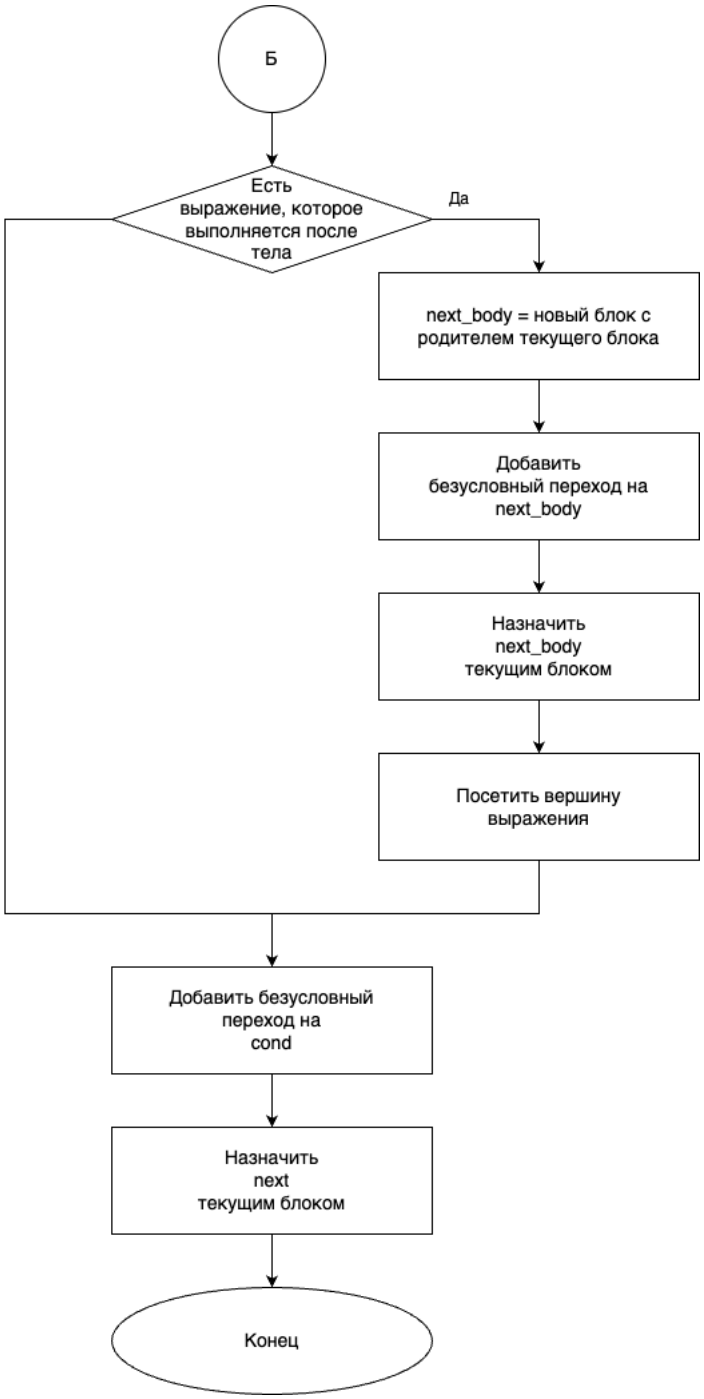

Рисунок 2.5 – Схема алгоритма генерации LLVM(IR) (3)

3 Технологическая часть

3.1 Ошибки компилирования

ANTLR4 обрабатыват все ошибки, которые возникают на этапах лексического анализа и синтаксического анализа. При это все возникающие исключения перехватываются и выводятся текстом в консоль.

3.2 Генерация ANTLR4

По грамматике языка программирования Pascal были сгенерированы следующие файлы с классами языка Python.

1. PascalLexer – лексический анализатор.
2. PascalParser – синтаксический анализатор.
3. PascalListener – абстрактный класс слушателя, с пустыми методами.

3.3 Обход сгенированного AST

Для обхода дерева был создан класс-наследник базового класса слушателя (PascalListener) для переопределения «`enter`» и «`exi`» методов, которые соответствуют праивлам исходной гнрамматики.

3.4 Пример компиляции программы

В качестве примера приводится программа на языке Pascal, которая выводит фразу «Hello World» в стандартный поток вывода. Она представлена в листинге 3.1 и сгенерированный LLVM(IR) код представлен в листинге 3.2.

Листинг 3.1 — Пример программы на Pascal

```
1  program HelloWorld;
2
3  begin
4    writeln("Hello, world!");
5  end.
```

Листинг 3.2 — Пример вывода LLVM(IR)

```
@.str_1 = private unnamed_addr constant [12 x i8] c"Hello World\00", align 1
declare i32 @printf(i8*, ...)
declare i32 @scanf(i8*, ...)
define i32 @main() {
%1 = getelementptr inbounds [12 x i8], [12 x i8]* @.str_1, i64 0, i64 0
%2 = ptrtoint i8* %1 to i99
%3 = inttoptr i99 %2 to i8*
%4 = call i32 (i8*, ...) @printf(i8* %3)
ret i32 0
ret i32 zeroinitializer
}
```

ЗАКЛЮЧЕНИЕ

В ходе выполнения данного курсового проекта была осуществлена разработка компилятора с использованием ANTLR и LLVM. Проект включал в себя создание грамматики для определения структуры языка, разработку парсера для преобразования исходного кода в абстрактное синтаксическое дерево (AST), а также реализацию генерации кода на основе LLVM IR.

Таким образом, цель достигнута, а все задачи выполнены.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. What is a compiler? [Электронный ресурс]. — Режим доступа URL: https://www.techtarget.com/whatis/definition/compiler (Дата обращения: 16.04.2024).
2. What is machine code (machine language)? [Электронный ресурс]. — Режим доступа URL: https://www.techtarget.com/whatis/definition/machine-code-machine-language (Дата обращения: 16.04.2024).
3. Introduction of Lexical Analysis [Электронный ресурс]. — Режим доступа URL: https://www.geeksforgeeks.org/introduction-of-lexical-analysis/ (Дата обращения: 16.04.2024).
4. Introduction to Syntax Analysis in Compiler Design [Электронный ресурс]. — Режим доступа URL: https://www.geeksforgeeks.org/introduction-to-syntax-analysis-in-compiler-design/ (Дата обращения: 16.04.2024).
5. Semantic Analysis in Compiler Design [Электронный ресурс]. — Режим доступа URL: https://www.geeksforgeeks.org/semantic-analysis-in-compiler-design/ (Дата обращения: 16.04.2024).
6. Compiler Design - Code Generation [Электронный ресурс]. — Режим доступа URL: https://www.tutorialspoint.com/compiler_design/compiler_design_code_generation.htm (Дата обращения: 16.04.2024).
7. What is ANTLR? [Электронный ресурс]. — Режим доступа URL: https://www.antlr.org (Дата обращения: 16.04.2024).
8. LLVM(IR) [Электронный ресурс]. — Режим доступа URL: https://llvm.org/docs/LangRef.html (Дата обращения: 16.04.2024).