**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» КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

***К КУРСОВОЙ РАБОТЕ НА ТЕМУ:***

«Компилятор языка Oberon»

Студент группы **ИУ7-21М**  **А.В. Борисов**

(Подпись, дата) (И.О.Фамилия)

Руководитель **\_А.А. Ступников**

(Подпись, дата) (И.О.Фамилия)

*2025 г.*

**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)**

УТВЕРЖДАЮ

Заведующий кафедрой ИУ7

(Индекс)

И.В. Рудаков\_

(И.О.Фамилия)

« » 2025 г.

**З А Д А Н И Е**

**на выполнение курсовой работы**

по дисциплине Конструирование компиляторов Студент группы ИУ7-21М

Борисов Артём Викторович

(Фамилия, имя, отчество)

Тема курсового проекта Компилятор языка Oberon.

Направленность КП (учебный, исследовательский, практический, производственный, др.)

учебный Источник тематики (кафедра, предприятие, НИР) кафедра График выполнения проекта: 25% к \_4\_ нед., 50% к \_7\_ нед., 75% к 11 нед., 100% к \_14\_ нед.

***Задание***\_Проанализировать грамматику языка Oberon, выделить её ключевые составляющие. Разработать прототип компилятора на основе скорректированной грамматики, использующий библиотеку ANTLR4 для синтаксического анализа входного потока данных и построения AST- дерева. Для последующих преобразований необходимо использовать LLVM, переводящий абстрактное дерево в IR (Intermediate Representation).

***Оформление курсового проекта:***

Расчетно-пояснительная записка на \_20-30 листах формата А4.

Расчетно-пояснительная записка должна содержать постановку задачи, введение, аналитическую, конструкторскую, технологическую части, заключение, список литературы.

Дата выдачи задания «14» февраля 2025 г.

**Руководитель курсового проекта**  А.А. Ступников

(Подпись, дата) (И.О.Фамилия)

**Студент**  А.В. Борисов

(Подпись, дата) (И.О.Фамилия)

**СОДЕРЖАНИЕ**

[ВВЕДЕНИЕ 5](#_Toc198904198)

[1. Аналитическая часть 6](#_Toc198904199)

[1.1. Составляющие компилятора 6](#_Toc198904200)

[1.1.1. Препроцессор 7](#_Toc198904201)

[1.1.2. Лексический анализатор 7](#_Toc198904202)

[1.1.3. Синтаксический анализатор 8](#_Toc198904203)

[1.1.4. Семантический анализатор 9](#_Toc198904204)

[1.1.5. Генерация кода 10](#_Toc198904205)

[1.2. Методы реализации лексического и синтаксического анализаторов 10](#_Toc198904206)

[1.2.1. Генераторы лексического анализатора 10](#_Toc198904207)

[1.2.2. Генераторы синтаксического анализатора 12](#_Toc198904208)

[1.3. LLVM 13](#_Toc198904209)

[2. Конструкторская часть 15](#_Toc198904210)

[2.1 IDEF0 15](#_Toc198904211)

[2.2. Язык Oberon 15](#_Toc198904212)

[2.3. Лексический и синтаксический анализаторы 18](#_Toc198904213)

[2.4. Семантический анализ 18](#_Toc198904214)

[3. Технологическая часть 19](#_Toc198904215)

[3.1. Выбор средств программной реализации 19](#_Toc198904216)

[3.2 Сгенерированные классы анализаторов 20](#_Toc198904217)

[3.3 Тестирование 21](#_Toc198904218)

[ЗАКЛЮЧЕНИЕ 24](#_Toc198904219)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 25](#_Toc198904220)

[ПРИЛОЖЕНИЕ A 26](#_Toc198904221)

[ПРИЛОЖЕНИЕ Б 28](#_Toc198904222)

# ВВЕДЕНИЕ

Компилятор – программное обеспечение, которое переводит поданный на вход текст программы, написанный на одном из языков программирования – исходном, в машинный код для исполнения на компьютере. В процессе преобразования команд выполняется оптимизация кода и анализ ошибок, что позволяет улучшить производительность и избежать некоторых сбоев при выполнении программы. [[1](#_bookmark26)]

Целью данной курсовой работы является разработка компилятора языка Oberon. Компилятор должен выполнять чтение текстового файла, содержащего код на языке Oberon и генерировать на выходе программу, пригодную для запуска.

Для достижения цели необходимо решить следующие задачи:

* проанализировать грамматику языка Oberon;
* изучить существующие средства для анализа исходных кодов программ, системы для генерации низкоуровневого кода, запуск которого возможен на большинстве из используемых платформ и операционных систем;
* реализовать прототип компилятора.

# Аналитическая часть

## Составляющие компилятора

Компилятор состоит из трёх частей.

1. *Frontend* преобразует текст программы на исходном языке во внутреннее представление, состоит из: препроцессора, лексического, синтаксического и семантического анализаторов, генератора промежуточного представления.
2. *Middle-end* занимается машинно-независимой оптимизацией полученного промежуточного представления.
3. *Backend* выполняет преобразование промежуточного представления в программу на языке целевой платформы (ассемблер или машинный код).

На рисунке 1.1 изображены основные фазы работы компилятора.

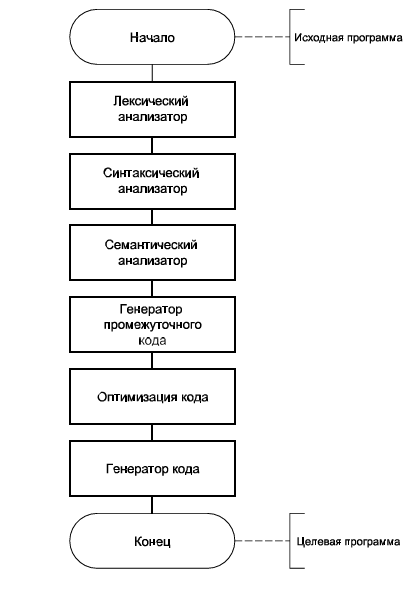


Рисунок 1.1 – Фазы компилятора.

### **Препроцессор**

Препроцессоры создают входной поток информации для компилятора, выполняют функции:

1. обработка макросов;
2. включение файлов;
3. обработка языковых расширений.

### **Лексический анализатор**

Цель – превратить поток символов в токены (этот процесс называется «токенизацией»). Выполняется группировка определённых терминальных символов в лексемы. Задаются конкретные правила в виде регулярных выражений, детерминированных конечных автоматов, грамматик.

Основные функции:

* удаление пробелов и комментариев,
* сборка последовательности цифр, формирующих константу;
* распознавание идентификаторов и ключевых слов.

Как правило, лексический анализатор находится между синтаксическим анализатором и входящим потоком и взаимодействует с ними, как показано на рисунке [1.2](#_bookmark6): анализатор считывает символы из входного потока, группирует их в лексемы и передаёт токены, образуемые этими лексемами, последующим стадиям компиляции.

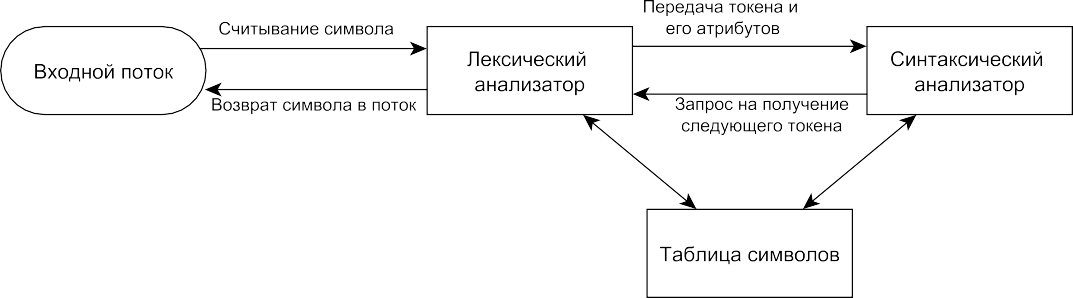


Рисунок 1.2 – Лексический анализатор.

Лексический анализ может представляться как один из этапов синтаксического анализа.

Обнаружение лексических ошибок, таких как недопустимые символы, ошибки идентификаторов или числовых констант, также является частью этого процесса.

### **Синтаксический анализатор**

Иерархический анализ называется разбором («parsing») или синтаксическим анализом, который включает группировку токенов исходной программы в грамматические фразы, используемые компилятором. Обычно они представляются в виде дерева разбора.

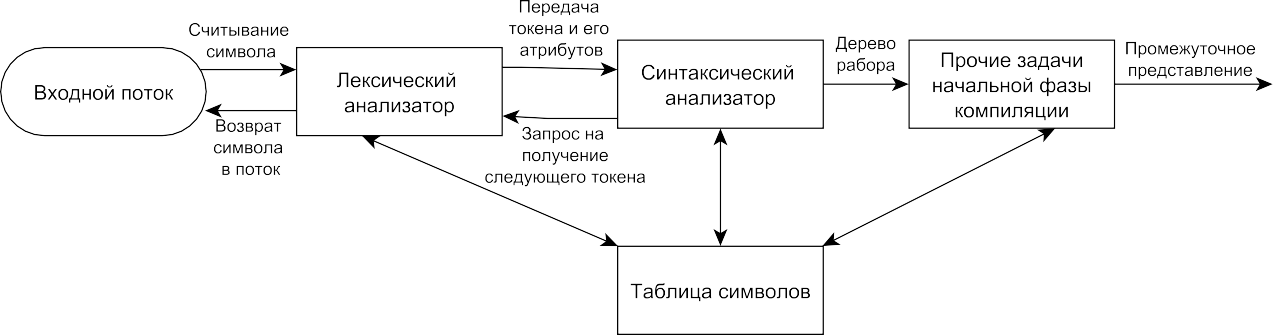


Рисунок 1.3 – Синтаксический анализатор.

Как правило, это представление выражается в виде абстрактного синтаксического дерева, где каждый внутренний узел является оператором, а дочерние – его аргументами. Среди них можно выделить несколько групп связанных объектов:

* элементы арифметических выражений: каждый узел представляет собой операцию и содержит её аргументы;
* элементы системы типов: базовые типы (числовые, строковые, структуры и т.п.), указатели, массивы и функции;
* выражения пяти типов: арифметические, блочные и управляющие выражения, условные конструкции, циклы.

Разбор начинается со стартового нетерминала.

Условные конструкции описывают конструкцию «if», включающую в себя арифметическое выражение условия, выражение, выполняемое в случае его истинности, и альтернативное опциональное выражение.

Конструкции циклов (включают в себя «while», «do while», «for») описывают арифметическое выражение условия и выражение, исполняемое в цикле.

Управляющие выражения – «break», «continue», «return» и т.д.

Блочные выражения – последовательность других выражений, преимущественно используются в качестве тел функций и условных выражений и циклов.

Полученная грамматическая структура используется в последующих этапах компиляции для анализа исходной программы и генерации кода для целевой платформы.

Синтаксический анализ выявляет синтаксические ошибки, относящиеся к нарушению структуры программы.

### **Семантический анализатор**

В процессе семантического анализа проверяется наличие семантических ошибок в исходной программе и накапливается информация о типах для следующей стадии – генерации кода. Используются иерархические структуры, полученные во время синтаксического анализа для идентификации операторов и операндов выражений и инструкций.

Как правило, семантический анализатор разделяется на ряд более мелких, каждый из которых предназначен для конкретной конструкции. Соответствующий семантический анализатор вызывается синтаксическим анализатором как только он распознает синтаксическую единицу, требующую обработки.

Семантические анализаторы взаимодействуют между собой посредством информации, хранящейся в структурах данных, например, в таблице символов.

### **Генерация кода**

Последний этап – генерация кода. Начинается тогда, когда во все системные таблицы занесена необходимая информация. В этом случае, компилятор переходит к построению соответствующей программы в машинном коде. Код генерируется при обходе дерева разбора, построенного на предыдущих этапах.

Для получения машинного кода требуется два отдельных прохода:

* генерация промежуточного кода;
* генерация собственно машинного кода.

Для каждого узла дерева генерируется соответствующий операции узла код на целевой платформы. В процессе анализа кода программы данные связываются с именами переменных. При выполнении генерации кода предполагается, что вход генератора не содержит ошибок. Результат – код, пригодный для исполнения на целевой платформе.

## Методы реализации лексического и синтаксического анализаторов

Существует два подхода для реализации лексического и синтаксического анализаторов:

* с использованием стандартных алгоритмов анализа;
* с привлечением готовых инструментов генерации.

### **Генераторы лексического анализатора**

Существует множество генераторов, наиболее популярные из них – Lex, Flex, ANTLR4 и другие.

**Lex** – стандартный инструмент для получения лексических анализаторов в операционных системах Unix, обычно используется совместно с генератором синтаксических анализаторов Yacc. В результате обработки входного потока получается исходный файл на языке Си. Lex-файл разделяется на три блока (блок определений, правил и кода на Си), разделённые строками, содержащими по два символа процента. [[2](#_bookmark27)]

В блоке определений задаются макросы и заголовочные файлы. Блок правил описывает шаблоны, представляющие собой регулярные выражения, и ассоциирует их с вызовами. Блок кода содержит операторы и функции на Си, которые копируются в генерируемый файл.

**Flex** (Fast Lexical Analyzer) заменяет Lex в системах на базе пакетов GNU и имеет аналогичную функциональность. [[3](#_bookmark28)]

**ANTLR** (ANother Tool for Language Recognition) – генератор лексических и синтаксических анализаторов, позволяет создавать анализаторы на таких язы- ках, как: Java, C#, Python 2, Python 3, JavaScript, Go, C++, Swift, PHP. [[4](#_bookmark29)]

ANTLR генерирует классы нисходящего рекурсивного синтаксического анализатора, на основе правил, заданных в виде РБНФ грамматики. Он также позволяет строить и обходить деревья синтаксического анализа с использованием паттернов посетитель или слушатель. Благодаря своей эффективности и простоте использования, ANTLR является одним из наиболее предпочтительных генераторов анализаторов при создании кода синтаксического анализатора. В текущей работе было решено использовать этот инструмент.

Решение использовать ANTLR было принято по ряду причин, которые выгодно отличают его от альтернативных решений, таких как **Flex/Bison**, **Lex/Yacc**, **Coco/R** и др.

Преимущества ANTLR:

1. **Унифицированный инструмент для лексического и синтаксического анализа**: В отличие от классических связок вроде Lex + Yacc или Flex + Bison, ANTLR позволяет описывать грамматику в одном файле и генерирует как лексер, так и парсер. Это уменьшает избыточность и повышает читаемость кода.
2. **Поддержка EBNF (расширенной формы Бэкуса-Наура)**: ANTLR позволяет использовать конструкции вроде \*, +, ?, альтернативы | и группировки (), что делает описание грамматик гораздо более интуитивным и лаконичным по сравнению с BNF, применяемой в более старых генераторах.
3. **Генерация абстрактного синтаксического дерева (AST)**: ANTLR по умолчанию строит дерево разбора, с которым удобно работать с помощью шаблона **Visitor** или **Listener**, позволяя легко реализовать семантический анализ или генерацию кода.
4. **Отладка и визуализация**: Существуют мощные средства отладки ANTLR-грамматик, включая плагины для IDE (например, плагин ANTLR для CLion) с возможностью визуализации дерева разбора.
5. **Кроссплатформенность и многозадачность**: ANTLR генерирует код для многих целевых языков (в том числе C++, что важно для проекта), и может быть встроен в любые сборочные цепочки (CMake, Make, Gradle и др.).
6. **Оптимизация производительности**: ANTLR использует алгоритм **LL(\*)**, что обеспечивает баланс между гибкостью грамматики и скоростью разбора. В отличие от LALR, используемого в Bison, LL(\*) не требует строгой левофакторизации или переписывания грамматик.

### **Генераторы синтаксического анализатора**

Для создания синтаксических анализаторов используются такие инструменты, как Yacc/Bison, Coco/R и описанный ранее ANTLR.

**Yacc** – стандартный генератор синтаксических парсеров в Unix системах,

**Bison** – аналогичный ему генератор для GNU систем. [[5](#_bookmark30)]

**Coco/R** – генератор лексических и синтаксических анализаторов. Лексические анализаторы работают по принципу конечных автоматов, а синтаксические используют рекурсивный спуск. Поддерживаются такие языки программирования, как C++, C#, Java и другие.

## LLVM

LLVM (Low Level Virtual Machine) – проект программной инфраструктуры для создания компиляторов и сопутствующих им утилит. В его основе лежит платформонезависимая система кодирования машинных инструкций – байткод LLVM IR (Intermediate Representation). LLVM может создавать байткод для множества платформ, включая ARM, x86, x86-64, GPU от AMD и Nvidia и другие. В проекте есть генераторы кода для множества языков, а для компиляции LLVM IR в код платформы используется clang. В состав LLVM входит также интерпретатор LLVM IR, способный исполнять код без компиляции в код платформы. [6]

Некоторые проекты имеют собственные LLVM-компиляторы, например, LLVM-версия GCC.

LLVM поддерживает целые числа произвольной разрядности, числа с плавающей точкой, массивы, структуры и функции. Большинство инструкций в LLVM принимает два аргумента (операнда) и возвращает одно значение (трёхадресный код).

Значения в LLVM определяются текстовым идентификатором. Локальные значения обозначаются префиксом %, а глобальные – @. Тип операндов всегда указывается явно и однозначно определяет тип результата. Операнды арифметических инструкций должны иметь одинаковый тип, но сами инструкции «перегружены» для любых числовых типов и векторов.

LLVM поддерживает полный набор арифметических операций, побитовых логических операций и операций сдвига. LLVM IR строго типизирован, поэтому существуют операции приведения типов, которые явно кодируются специальными инструкциями. Кроме того, существуют инструкции преобразования между целыми числами и указателями, а также универсальная инструкция для приведения типов bitcast.

Помимо значений регистров в LLVM есть работа с памятью. Значения в памяти адресуются типизированными указателями. Обратиться к ней можно с помощью двух инструкций: load и store. Инструкция alloca выделяет память на стеке. Она автоматически освобождается при выходе из функции при помощи инструкций ret или unwind.

Для вычисления адресов элементов массивов и структур с правильной типизацией используется инструкция getelementptr. Она только вычисляет адрес без обращения к памяти, принимает произвольное количество индексов и может разыменовывать структуры любой вложенности.

**Выводы**

В данном разделе приведён обзор основных фаз компиляции, описана каждая из них. Также был выбран генератор лексического и синтаксического анализаторов – ANTLR и LLVM в качестве генератора машинного кода.

# Конструкторская часть

## 2.1 IDEF0

Концептуальная модель разрабатываемого компилятора в нотации IDEF0 представлена на рисунке [2.1](#_bookmark16).

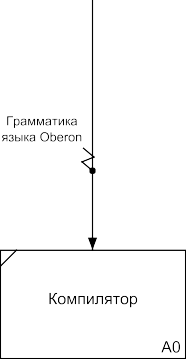


Рисунок 2.1 – Концептуальная модель в нотации IDEF0.

## 2.2. Язык Oberon

Язык программирования **Oberon** представляет собой модульный, строго типизированный процедурный язык, созданный Никлаусом Виртом как эволюционное продолжение языков Pascal и Modula-2. Oberon разрабатывался с акцентом на минимализм, понятность и формальную строгость, что делает его удобной средой для преподавания основ программирования, системной разработки и реализации компиляторов. Несмотря на внешнюю простоту, язык поддерживает полноценные абстракции, управление памятью и структурирование программ.

**1. Статическая и строгая типизация**

Ключевой особенностью языка Oberon является **строгая статическая типизация**, предполагающая, что все переменные и выражения имеют однозначно определённый тип во время компиляции. Типы указываются явно при объявлении переменной, например: *VAR x: INTEGER;.*

Подобная система исключает неявные преобразования типов и значительно упрощает реализацию семантического анализатора компилятора. Любые попытки выполнить операции над несовместимыми типами фиксируются как ошибки на этапе компиляции. Это позволяет на раннем этапе отсеивать недопустимые конструкции и облегчает генерацию корректного промежуточного представления в LLVM.

Кроме того, строгая типизация оказывает влияние на семантику выражений, особенно в операциях сравнения и присваивания: выражения, участвующие в таких операциях, должны быть одного и того же типа. В противном случае компилятор обязан выдать диагностическое сообщение.

**2. Ограниченный, но выразительный набор типов**

Язык Oberon поддерживает небольшой, но достаточный для большинства задач набор типов:

* Примитивные типы: *INTEGER, REAL*
* Массивы: ARRAY, с фиксированным размером, например: ARRAY 100 OF INTEGER

Такой минималистичный подход к типовой системе позволяет сосредоточиться на базовых принципах компиляции, таких как управление памятью, выравнивание и работа с указателями. При трансляции в LLVM компилятор должен корректно отображать эти типы в соответствующие конструкции IR, например, *i32* для *INTEGER*, массивы — в виде *getelementptr*, а записи — через агрегатные типы.

**3. Модульная структура программы**

Программа в Oberon структурируется как **модуль** — именованная единица компиляции, включающая в себя объявление переменных, процедур и основной блок. Подобная структура способствует упрощённому синтаксическому анализу: каждый модуль начинается с ключевого слова MODULE и завершается указанием его имени после END. Это позволяет организовать компиляцию как пофайловую трансляцию, что особенно удобно при разработке крупных проектов.

**4. Отсутствие динамической типизации и перегрузки**

В отличие от современных языков, таких как Python или C++, Oberon не поддерживает динамическую типизацию, перегрузку операторов и функций. Это упрощает реализацию компилятора, исключая необходимость в диспетчеризации по типам и сложных таблицах символов. Все вызовы и операции однозначно определяются на этапе анализа, а кодогенератор опирается на чёткие типы без дополнительных проверок во время выполнения. Грамматика приведена в Приложении А.

## 2.3. Лексический и синтаксический анализаторы

Лексический и синтаксический анализаторы в данной работе генерируются с помощью ANTLR. На вход поступает грамматика языка в формате ANTLR4 (файл с расширением .g4).

В результате работы создаются файлы, содержащие классы лексера и парсера, а также вспомогательные файлы и классы для их работы. Также генерируются шаблоны классов для обхода дерева разбора, которое получается в результате работы парсера.

На вход лексера подаётся текст программы, преобразованный в поток символов. На выходе получается поток токенов, который затем подаётся на вход парсера. Результатом его работы является дерево разбора.

Ошибки, возникающие в ходе работы лексера и парсера, выводятся в стандартный поток ввода-вывода.

## 2.4. Семантический анализ

Абстрактное синтаксическое дерево можно обойти двумя способами: применяя паттерн Listener или Visitor.

Listener позволяет обходить дерево в глубину и вызывает обработчики соответствующих событий при входе и выходе из узла дерева.

Visitor предоставляет возможность более гибко обходить построенное дерево и решить, какие узлы и в каком порядке нужно посетить. Таким образом, для каждого узла реализуется метод его посещения. Обход начинается с точки входа в программу (корневого узла).

**Выводы**

В текущем разделе была представлена концептуальная модель в нотации IDEF0, приведена грамматика языка Oberon, описаны принципы работы лексического и синтаксического анализаторов и идея семантического анализа.

# Технологическая часть

## Выбор средств программной реализации

В качестве основного языка программирования для разработки компилятора был выбран C++, что обусловлено как техническими, так и практическими причинами. C++ является одним из наиболее распространённых языков системного программирования, обладающим широкими возможностями как для низкоуровневой работы с памятью, так и для организации сложной объектно-ориентированной архитектуры компилятора.

Ключевые причины выбора C++ включают следующие аспекты:

* **Наличие развитой экосистемы для разработки компиляторов:** C++ активно используется в области трансляции и предоставляет доступ ко множеству высококачественных инструментов и библиотек, среди которых особо выделяются:
  + **ANTLR** — мощный генератор лексических и синтаксических анализаторов, позволяющий описывать грамматики с использованием расширенного EBNF-формата;
  + **LLVM** — современный инфраструктурный фреймворк для построения промежуточного представления (IR), генерации, оптимизации и компиляции низкоуровневого кода под различные архитектуры;
  + **STL** — обширная библиотека, предоставляющие структуры данных, алгоритмы, средства работы с файлами и строками, что значительно ускоряет разработку.
* **Высокая производительность и контроль над ресурсами:** Благодаря близости к аппаратному уровню C++ обеспечивает высокую производительность, что особенно важно при разработке компиляторов, где критичны скорость анализа, оптимизации и генерации кода.
* **Накопленный опыт и зрелость инструментов:** На момент начала разработки уже имелся опыт программирования на C++, включая владение современными стандартами языка (C++17/20), знание принципов объектно-ориентированного проектирования и опыт работы с внешними библиотеками. Это позволило значительно сократить время на освоение новых технологий и сосредоточиться непосредственно на реализации логики компиляции.
* **Поддержка кроссплатформенной сборки:** Язык C++ совместим с основными системами сборки (CMake, Make), что упрощает переносимость проекта на различные операционные системы. Это особенно удобно при работе с фреймворками вроде LLVM, которые активно используются в разных средах.
* **Гибкость архитектурных решений:** Благодаря возможности тонкой настройки архитектуры приложения, C++ позволил построить модульную структуру компилятора с раздельной реализацией лексического анализа, синтаксического анализа, семантической обработки и генерации кода. Использование таких паттернов, как Visitor, облегчает сопровождение и расширение проекта.

Таким образом, использование языка C++ обеспечило не только соответствие всем техническим требованиям, но и комфортную и продуктивную среду разработки. Это решение оказалось наиболее рациональным с точки зрения баланса между мощностью, гибкостью и доступностью средств разработки компилятора.

## Сгенерированные классы анализаторов

В результате работы ANTLR генерируются следующие классы.

* OberonLexer — лексический анализатор.
* OberonParser — синтаксический анализатор.
* OberonVisitor — абстрактный класс посетителя для обхода дерева.
* OberonListener — абстрактный класс слушателя, с пустыми методами.
* OberonBaseVisitor — реализация интерфейса OberonVisitor c пустыми переопределенными методами.
* OberonBaseListener — реализация интерфейса OberonListener с пустыми переопределенными методами.

Для обхода синтаксического дерева используется паттерн Visitor. Создаётся класс, унаследованный от OberonVisitor, в котором переопределяются методы visit, соответствующие различным правилам грамматики. Обход выполняется вручную: каждый метод вызывается явно, а для перехода к дочерним узлам используется visitChildren(ctx). Такой подход обеспечивает точный контроль над порядком обхода дерева и позволяет задавать необходимую логику обработки содержимого каждого узла.

## 3.3 Тестирование

Для проверки корректной работы компилятора был реализован модуль тестирования, основанный на классе OberonCompile. В рамках этого модуля осуществляется автоматическое сопоставление пар файлов: исходного файла на языке Oberon и соответствующего файла с ожидаемым результатом выполнения. Все тесты организованы в двух каталогах: tests/oberonPrograms и tests/expected. Каталог oberonPrograms содержит входные программы, а expected — корректный текстовый вывод, который должна сгенерировать каждая программа при запуске. После компиляции программа выполняется, и результат её выполнения сохраняется в директорию out, после чего автоматически сравнивается с соответствующим эталонным файлом из expected.

Сравнение осуществляется построчно, с выводом различий, если они имеются. Это позволяет эффективно проверять как корректность трансляции исходного кода, так и правильность поведения сгенерированной программы на выходе.

Для обеспечения покрытия разных сценариев были выделены и протестированы следующие **классы эквивалентности** входных данных:

* Объявление и использование переменных: присваивание значений, использование в выражениях;
* Арифметические операции: сложение, вычитание, умножение, деление, DIV, MOD;
* Логические и сравнительные выражения: <, >, =, #;
* Условные конструкции: IF, IF-ELSE;
* Циклы: WHILE, FOR с различными условиями;
* Обработка массивов фиксированной длины: чтение и запись элементов;

Кроме классов эквивалентности, проводилось тестирование на корректность поведения компилятора при следующих граничных и типичных сценариях:

* Использование переменных без предварительной инициализации;
* Вложенные условные блоки;
* Последовательность операций в теле модуля, проверка порядка вычислений.

Данный подход позволил убедиться в устойчивости работы компилятора, а также в корректности вывода результата для типичных программ на языке Oberon.

**Структура системы тестирования**

* **Каталоги тестовых данных.**  
  Все тестовые файлы хранятся в специально выделенных директориях:
  + oberonPrograms/ — содержит исходные программы на языке Oberon;
  + expected/ — содержит ожидаемые выходные данные программ;
  + LLVMIR/ — директория, куда записываются сгенерированные компилятором .ll файлы для последующего сравнения.
  + out/ - директория, где лежат файлы с выводом программы
* **Процесс тестирования.**  
  Класс OberonCompile реализует логику тестирования следующим образом:
  + Проходит по списку входных файлов в директории oberonPrograms/;
  + Для каждого файла запускается процесс компиляции;
  + Сгенерированный LLVM IR сохраняется в файл в каталоге LLVMIR/ с тем же именем, что и у исходной программы;
  + Путём построчного сравнения с соответствующим файлом из директории expected/ осуществляется проверка эквивалентности вывода;
  + В случае несовпадения результаты теста помечаются как ошибочные, при этом выводится информация о расхождении (например, с помощью GTest или логирования вручную).

**Используемые технологии**

Для написания тестов использовалась библиотека **Google Test (GTest)**, которая предоставляет удобный и выразительный интерфейс для написания модульных и интеграционных тестов. Каждый тест оформлен в виде TEST-кейса, в котором загружаются пары файлов и выполняется их сравнение. При несовпадении результат теста считается проваленным.

## 

# ЗАКЛЮЧЕНИЕ

Таким образом, в рамках текущей курсовой работы рассмотрены основные части компилятора, алгоритмы и способы их реализации. Также были рассмотрены инструменты генерации лексических и синтаксических анализаторов.

Был разработан прототип компилятора языка Oberon, использующий ANTLR для синтаксического анализа входного потока данных и построения AST-дерева, и LLVM для последующих преобразований, переводящих абстрактное дерево в IR.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. АХО А.В, ЛАМ М.С., СЕТИ Р., УЛЬМАН Дж.Д. Компиляторы: принципы, технологии и инструменты. – М.: Вильямс, 2008.
2. Lesk M. E., Schmidt E. Lex: A lexical analyzer generator. – Murray Hill, NJ : Bell Laboratories, 1975. – С. 1-13.
3. Sampath P. et al. How to test program generators? A case study using flex //Fifth IEEE International Conference on Software Engineering and Formal Methods (SEFM 2007). – IEEE, 2007. – С. 80-92.
4. What is ANTLR? [Электронный ресурс]. – Режим доступа: [https://www.antlr.org/](http://www.antlr.org/) (Дата обращения: 25.04.2025).
5. Donnelly C. BISON the YACC-compatible parser generator //Technical report, Free Software Foundation. – 1988.
6. The LLVM Compiler Infrastructure Project [Электронный ресурс]. – Режим до- ступа: https://llvm.org/ (Дата обращения: 27.04.2025).

# ПРИЛОЖЕНИЕ A

1 grammar Oberon;

2

3 ident: IDENT;

4 qualident: ident;

5 identdef: ident '\*'?;

6

7 integer: (DIGIT+);

8 real: DIGIT+ '.' DIGIT\*;

9 number: integer | real;

10

11 constDeclaration: identdef '=' constExpression;

12 constExpression: expression;

13

14 typeDeclaration: identdef '=' type\_;

15 type\_: qualident | arrayType;

16 arrayType: ARRAY length OF type\_;

17

18 length: constExpression;

19

20 identList: identdef (',' identdef)\*;

21 variableDeclaration: identList ':' type\_;

22

23 expression: simpleExpression (relation simpleExpression)?;

24 relation: '=' | '#' | '<' | '<=' | '>' | '>=';

25 simpleExpression: ('+' | '-')? term (addOperator term)\*;

26 addOperator: '+' | '-' | OR;

27 term: factor (mulOperator factor)\*;

28 mulOperator: '\*' | '/' | DIV | MOD | '&';

29 factor: number | STRING | designator (actualParameters)? | '(' expression ')'

| '~' factor;

30 designator: qualident selector\*;

31 selector: '[' expList ']';

32 expList: expression (',' expression)\*;

33 actualParameters: '(' expList? ')';

34 statement: (assignment | ifStatement | whileStatement | forStatement)?;

35 assignment: designator ':=' expression;

36 statementSequence: statement (';' statement)\*;

37 ifStatement: IF expression THEN statementSequence (ELSIF expression THEN statementSequence)\* (ELSE statementSequence)? END;

38 whileStatement: WHILE expression DO statementSequence (ELSIF expression DO statementSequence)\* END;

39 forStatement: FOR ident ':=' expression TO expression (BY constExpression)? DO statementSequence END;

40 declarationSequence: (CONST (constDeclaration ';')\*)? (TYPE (typeDeclaration ';')\*)? (VAR (variableDeclaration ';')\*)?;

41

42 module: MODULE ident ';' declarationSequence (BEGIN statementSequence)? RETURN factor ';' END ident '.' EOF;

43

44 ARRAY: 'ARRAY';

45 OF: 'OF';

46 END: 'END';

47 TO: 'TO';

48 OR: 'OR';

49 DIV: 'DIV';

50 MOD: 'MOD';

51 IF: 'IF';

52 THEN: 'THEN';

53 ELSIF: 'ELSIF';

54 ELSE: 'ELSE';

55 WHILE: 'WHILE';

56 DO: 'DO';

57 FOR: 'FOR';

58 BY: 'BY';

59 BEGIN: 'BEGIN';

60 RETURN: 'RETURN';

61 TYPE: 'TYPE';

62 VAR: 'VAR';

63 MODULE: 'MODULE';

64 STRING: ('"' .\*? '"');

65 IDENT: LETTER (LETTER | DIGIT)\*;

66 LETTER: [a-zA-Z];

67 DIGIT: [0-9];

68 COMMENT: '(\*' .\*? '\*)' -> skip;

69 WS: [ \t\r\n] -> skip;

Листинг 1: Грамматика языка Oberon

# ПРИЛОЖЕНИЕ Б

На листингах 1-2 ниже приведены примеры кода на языке Oberon для нахождения пятого числа Фибоначчи и соответствующие файлы (листинг 3-4) промежуточного представления.

|  |  |  |
| --- | --- | --- |
| 1 | MODULE Fib; |  |
| 2 | VAR |
| 3 | n, x1, x2, tmp, i: | INTEGER; |
| 4 |  |  |
| 5 | BEGIN |  |
| 6 | n := 5; |  |
| 7 | i := 1; |  |
| 8 |  |  |
| 9 | x1 := 0; |  |
| 10 | x2 := 1; |  |
| 11 |  |  |
| 12 | WHILE i < n DO |  |
| 13 | tmp := x1 + x2; |  |
| 14 | x1 := x2; |  |
| 15 | x2 := tmp; |  |
| 16 |  |  |

17

i := i + 1;

18 END;

19 RETURN x1;

20 END Fib.

Листинг 2: Программа для нахождения числа Фибоначчи

1 ; ModuleID = 'module'

2 source\_filename = "module"

3

4 @n = common global i32 0

5 @x1 = common global i32 0

6 @x2 = common global i32 0

7 @tmp = common global i32 0

8 @i = common global i32 0

9

10 define i32 @main() {

11 main\_entry:

12 store i32 5, ptr @n, align 4

13 store i32 1, ptr @i, align 4

14 store i32 0, ptr @x1, align 4

15 store i32 1, ptr @x2, align 4

16 br label %while

17

18 while: ; preds = %do, %main\_entry

19 %value = load i32, ptr @i, align 4

20 %value1 = load i32, ptr @n, align 4

21 %cmp\_smaller = icmp slt i32 %value, %value1

22 br i1 %cmp\_smaller, label %do, label %end

23

24 do: ; preds = %while

25 %value2 = load i32, ptr @x1, align 4

26 %value3 = load i32, ptr @x2, align 4

27 %math\_operation\_int = add i32 %value2, %value3

28 store i32 %math\_operation\_int, ptr @tmp, align 4

29 %value4 = load i32, ptr @x2, align 4

30 store i32 %value4, ptr @x1, align 4

31 %value5 = load i32, ptr @tmp, align 4

32 store i32 %value5, ptr @x2, align 4

33 %value6 = load i32, ptr @i, align 4

34 %math\_operation\_int7 = add i32 %value6, 1

35 store i32 %math\_operation\_int7, ptr @i, align 4

36 br label %while

37

38 end: ; preds = %while

39 %value8 = load i32, ptr @x1, align 4

40 ret i32 %value8

41 }

Листинг 3: Файл промежуточного представления для программы нахождения чисел Фибоначчи

|  |  |  |
| --- | --- | --- |
| 1 | MODULE FibArray; |  |
| 2 | VAR |
| 3 | n, i: INTEGER; |
| 4 | arr: ARRAY 100 | OF INTEGER; |
| 5 |  |  |
| 6 | BEGIN |  |
| 7 | n := 5; |  |
| 8 | i := 2; |  |
| 9 |  |  |
| 10 | arr[0] := 0; |  |
| 11 | arr[1] := 1; |  |
| 12 |  |  |
| 13 | WHILE i < n DO |  |

Листинг 4: Программа для нахождения числа Фибоначчи на массиве

14

arr[i] := arr[i - 1] + arr[i - 2];

15

16

i := i + 1;

17 END;

18 RETURN arr[n - 1];

19 END FibArray.

1 ; ModuleID = 'module'

2 source\_filename = "module"

3

4 @n = common global i32 0

5 @i = common global i32 0

6 @arr = common global [100 x i32] zeroinitializer

7

8 define i32 @main() {

9 main\_entry:

10 store i32 5, ptr @n, align 4

11 store i32 2, ptr @i, align 4

12 store i32 0, ptr @arr, align 4

13 store i32 1, ptr getelementptr inbounds ([100 x i32], ptr @arr, i32 0, i32 1), align 4

14 br label %while

15

16 while: ; preds = %do, %main\_entry

17 %value = load i32, ptr @i, align 4

18 %value1 = load i32, ptr @n, align 4

19 %cmp\_smaller = icmp slt i32 %value, %value1

20 br i1 %cmp\_smaller, label %do, label %end

21

22 do: ; preds = %while

23 %value2 = load i32, ptr @i, align 4

24 %arr\_element\_ptr = getelementptr [100 x i32], ptr @arr, i32 0, i32 %value2

25 %value3 = load i32, ptr @i, align 4

26 %math\_operation\_int = sub i32 %value3, 1

27 %arr\_element\_ptr4 = getelementptr [100 x i32], ptr @arr, i32 0, i32 % math\_operation\_int

28 %value5 = load i32, ptr %arr\_element\_ptr4, align 4

29 %value6 = load i32, ptr @i, align 4

30 %math\_operation\_int7 = sub i32 %value6, 2

31 %arr\_element\_ptr8 = getelementptr [100 x i32], ptr @arr, i32 0, i32 % math\_operation\_int7

32 %value9 = load i32, ptr %arr\_element\_ptr8, align 4

33 %math\_operation\_int10 = add i32 %value5, %value9

34 store i32 %math\_operation\_int10, ptr %arr\_element\_ptr, align 4

35 %value11 = load i32, ptr @i, align 4

36 %math\_operation\_int12 = add i32 %value11, 1

37 store i32 %math\_operation\_int12, ptr @i, align 4

38 br label %while

39

40 end: ; preds = %while

41 %value13 = load i32, ptr @n, align 4

42 %math\_operation\_int14 = sub i32 %value13, 1

43 %arr\_element\_ptr15 = getelementptr [100 x i32], ptr @arr, i32 0, i32 % math\_operation\_int14

44 %value16 = load i32, ptr %arr\_element\_ptr15, align 4

45 ret i32 %value16

46 }

Листинг 5: Файл промежуточного представления для программы нахождения чисел Фибоначчи на массиве