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 **ИУ7-22М** _____ **А.А. Андреев**
(Подпись, дата) (И.О.Фамилия)

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

2024 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)
« _____» _____2024 г.

З А Д А Н И Е
на выполнение курсовой работы

по дисциплине Конструирование компиляторов

Студент группы ИУ7-22М

Андреев Александр Алексеевич
(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 «16» марта 2024 г.

Руководитель курсового проекта _____ А.А. Ступников
(Подпись, дата) (И.О.Фамилия)

Студент _____ А.А. Андреев
(Подпись, дата) (И.О.Фамилия)

СОДЕРЖАНИЕ

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]

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.

1 Аналитическая часть

1.1 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.

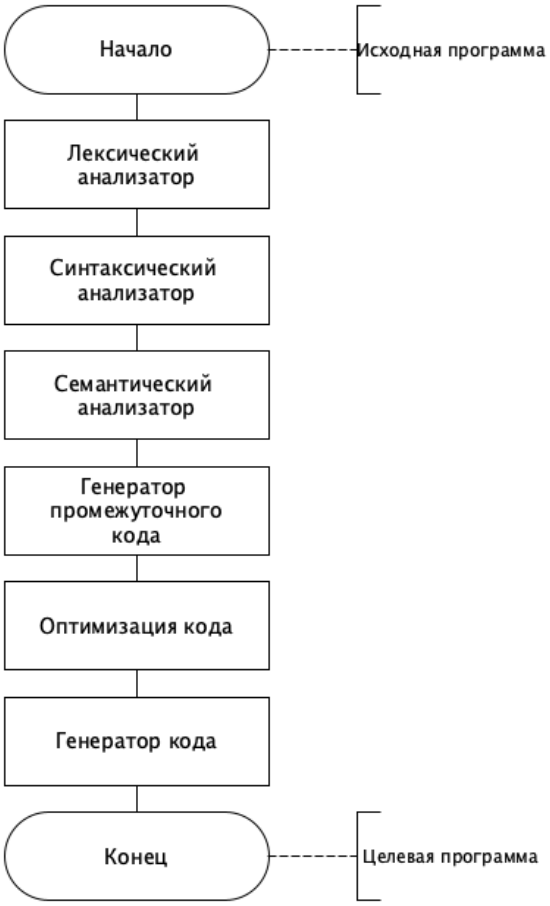

Рисунок 1.1 – Фазы компилятора.

1.1.1 Препроцессор

Препроцессоры создают входной поток информации для компилятора, выполняют функции:

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

1.1.2 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: анализатор считывает символы из входного потока, группирует их в лексемы и передаёт токены, образуемые этими лексемами, последующим стадиям компиляции.

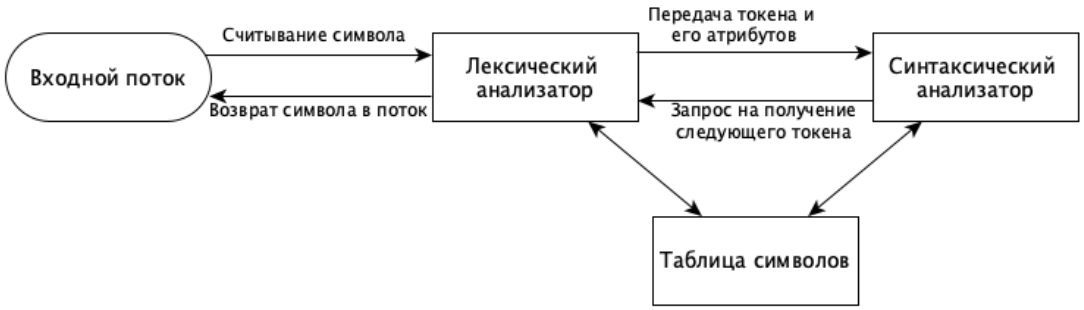

Рисунок 1.2 – Лексический анализатор.

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.

1.1.3 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.

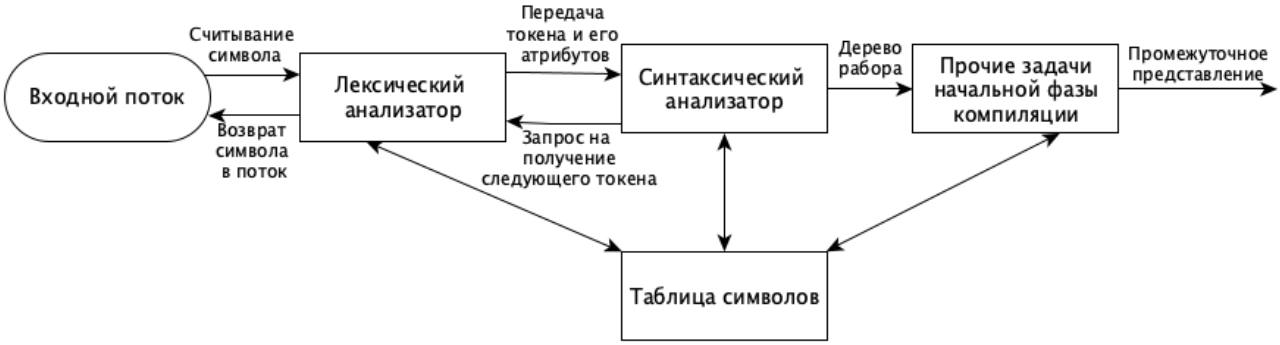

Рисунок 1.3 – Синтаксический анализатор.

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.

1.1.4 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.

1.1.5 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 предпола-

гается, что вход генератора не содержит ошибок. Результат – код, пригодный для исполнения на целевой платформе.

1.2 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.

1.2.1 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]

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]

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

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.

1.2.2 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]

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.

1.3 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.

2 Конструкторская часть

2.1 IDEF0

Концептуальная модель разрабатываемого компилятора в нотации IDEF0 представлена на рисунке 2.1.

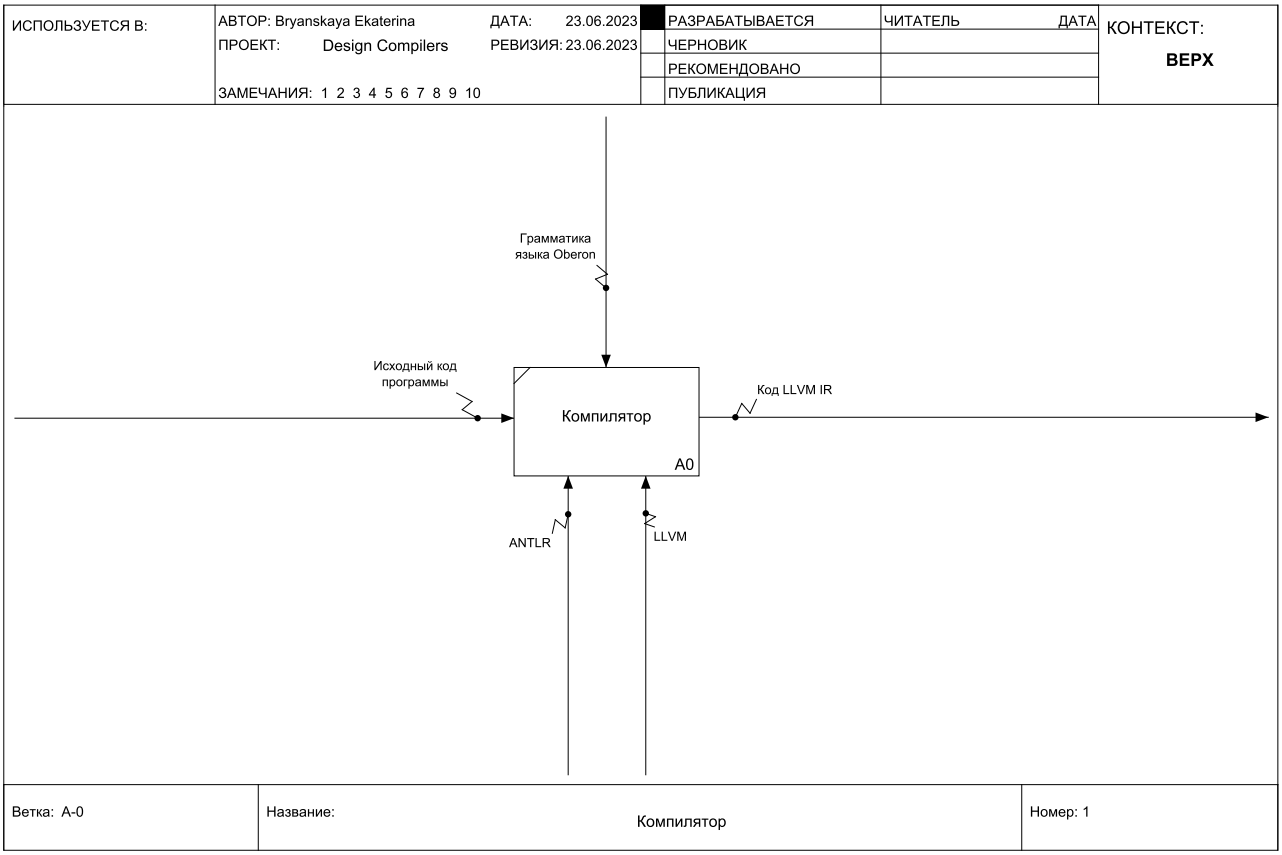

Рисунок 2.1 – Концептуальная модель в нотации IDEF0.

2.2 Язык Oberon

Oberon – язык программирования высокого уровня, предназначенный для исполнения программ на одноимённой операционной системе и основанный на таких языках, как Modula-2, Pascal.

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.

3 Технологическая часть

3.1 Выбор средств программной реализации

В качестве языка программирования была выбрана Java 17, ввиду нескольких причин.

- Компилятор, написанный на Java, может быть запущен на различных платформах, поддерживающих виртуальную машину JVM.
- Предоставляется много библиотек и инструментов для разработки компиляторов, включающих в себя инструменты для анализа синтаксиса, генерации кода и оптимизации.
- В дополнение, на момент реализации уже был накоплен существенный опыт в использовании этого языка программирования.

3.2 Сгенерированные классы анализаторов

В результате работы ANTLR генерируются следующие файлы.

1) Oberon.interp и OberonLexer.interp содержат данные (таблицы предсказания, множества следования, информация о правилах грамматики и т.д.) для интерпретатора ANTLR, используются для ускорения работы сгенерированного парсера для принятия решений о разборе входного потока.
2) Oberon.tokens и OberonLexer.tokens перечислены символические имена токенов, каждому из которых сопоставлено числовое значение типа токена. ANTLR4 использует их для создания отображения между символическими именами токенов и их числовыми значениями.
3) Интерфейсы OberonListener.java и OberonVisitor.java.
4) OberonBaseListener.java и OberonBaseVisitor.java – реализации соответствующих интерфейсов паттернов Listener и Visitor.
5) OberonParser.java – синтаксический анализатор.
6) OberonLexer.java – лексический анализатор.

3.3 Тестирование

Для проверки корректной работы программы был написан класс OberonCompil в котором указаны пары файлов (исходный файл программы на языке Oberon и файл с ожидаемым результатом). При запуске тестирования последовательно обрабатывается каждый исходный файл и результат сравнивается с тем, что написан в соответствующем парном файле.

3.4 Пример работы программы

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

Листинг 1: Программа для нахождения числа Фибоначчи

```
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: Файл промежуточного представления для программы нахождения чисел Фибоначчи

```
; ModuleID = 'module'
source_filename = "module"

@n = common global i32 0
@x1 = common global i32 0
@x2 = common global i32 0
@tmp = common global i32 0
@i = common global i32 0

define i32 @main() {
main_entry:
  store i32 5, ptr @n, align 4
  store i32 1, ptr @i, align 4
  store i32 0, ptr @x1, align 4
  store i32 1, ptr @x2, align 4
  br label %while

while:                                            ; preds = %do, %main_entry
  %value = load i32, ptr @i, align 4
  %value1 = load i32, ptr @n, align 4
  %cmp_smaller = icmp slt i32 %value, %value1
  br i1 %cmp_smaller, label %do, label %end

do:                                               ; preds = %while
  %value2 = load i32, ptr @x1, align 4
  %value3 = load i32, ptr @x2, align 4
  %math_operation_int = add i32 %value2, %value3
  store i32 %math_operation_int, ptr @tmp, align 4
  %value4 = load i32, ptr @x2, align 4
  store i32 %value4, ptr @x1, align 4
  %value5 = load i32, ptr @tmp, align 4
  store i32 %value5, ptr @x2, align 4
  %value6 = load i32, ptr @i, align 4
  %math_operation_int7 = add i32 %value6, 1
  store i32 %math_operation_int7, ptr @i, align 4
  br label %while

end:                                              ; preds = %while
```

```
39   %value8 = load i32, ptr @x1, align 4
40   ret i32 %value8
41 }
```

Листинг 3: Программа для нахождения числа Фибоначчи на массиве

```
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
14     arr[i] := arr[i - 1] + arr[i - 2];
15
16     i := i + 1;
17   END;
18   RETURN arr[n - 1];
19 END FibArray.
```

Листинг 4: Файл промежуточного представления для программы нахождения чисел Фибоначчи на массиве

```
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 }
```

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/ (Дата обращения: 25.04.2023).

5. Donnelly C. BISON the YACC-compatible parser generator //Technical report, Free Software Foundation. – 1988.

6. The LLVM Compiler Infrastructure Project [Электронный ресурс]. – Режим доступа: https://llvm.org/ (Дата обращения: 27.04.2023).

ПРИЛОЖЕНИЕ А

Листинг 5: Грамматика языка Oberon

```
grammar Oberon;

ident: IDENT;
qualident: ident;
identdef: ident '*'?;

integer: (DIGIT+);
real: DIGIT+ '.' DIGIT*;
number: integer | real;

constDeclaration: identdef '=' constExpression;
constExpression: expression;

typeDeclaration: identdef '=' type_;
type_: qualident | arrayType;
arrayType: ARRAY length OF type_;

length: constExpression;

identList: identdef (',' identdef)*;
variableDeclaration: identList ':' type_;

expression: simpleExpression (relation simpleExpression)?;
relation: '=' | '#' | '<' | '<=' | '>' | '>=';
simpleExpression: ('+' | '-')? term (addOperator term)*;
addOperator: '+' | '-' | OR;
term: factor (mulOperator factor)*;
mulOperator: '*' | '/' | DIV | MOD | '&';
factor: number | STRING | designator (actualParameters)? | '(' expression ')'
      | '~' factor;
designator: qualident selector*;
selector: '[' expList ']';
expList: expression (',' expression)*;
actualParameters: '(' expList? ')';
statement: (assignment | ifStatement | whileStatement | forStatement)?;
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;
```