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:

«Разработка компилятора языка Lua»

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

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

2025 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Транслятор — это программа перевода текста программы с исходного языка на объектный [1]. Если исходный язык является языком программирования высокого уровня, и, если объектный язык — язык ассемблера или машинный язык, то транслятор называют **компилятором**. Трансляция исходной программы в объектную выполняется во время компиляции, а фактическое выполнение объектной программы во время выполнения готовой программы.

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

В ходе работы необходимо решить следующие задачи:

1) Проанализировать грамматику языка Lua и выделить ее ключевые составляющие.

2) Изучить существующие средства для анализа исходных кодов программ, системы для генерации низкоуровневого кода, запуск которого возможен на большинстве из используемых платформ и операционных систем.

3) Разработать прототип компилятора на языке Go, выполняющий синтаксический анализ исходного текста программы и построение абстрактного синтаксического дерева.

4) Провести преобразование абстрактного синтаксического дерева в IR с использованием LLVM.

1 Аналитический раздел

1.1 Устройство компилятора

Процесс компиляции состоит из следующих этапов [2].

1) Лексический анализ.

2) Синтаксический анализ.

3) Семантический анализ.

4) Генерация кода на языке целевой платформы.

1.2 Лексический анализатор

Лексический анализ — распознавание базовых элементов языка, перевод исходной программы в поток лексем и передача токенов, образуемых этими лексемами, последующим стадиям компиляции.

Основные функции лексического анализатора:

— удаление пробелов и комментариев, обнаружение лексических ошибок;

— сборка последовательности цифр, формирующих константу;

— распознавание идентификаторов и ключевых слов.

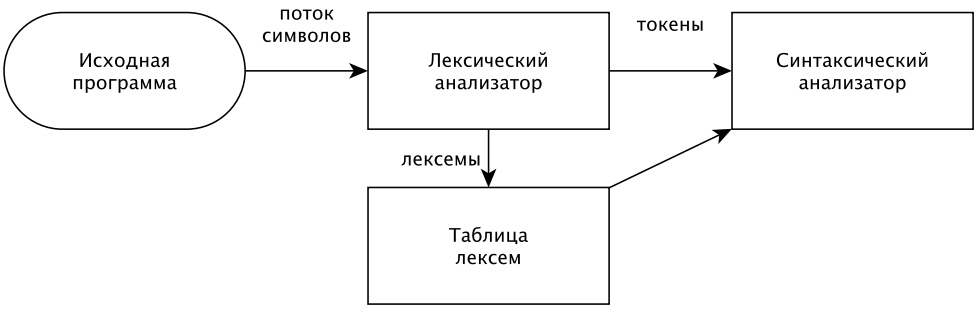

Рисунок 1.1 – Лексический анализатор

1.3 Синтаксический анализатор

Вторая фаза компилятора — синтаксический анализ или разбор. Анализатор использует первые компоненты токенов, полученных при лексическом анализе, для создания древовидного промежуточного представления, которое описывает грамматическую структуру потока токенов. Типичным представлением является синтаксическое дерево.

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.

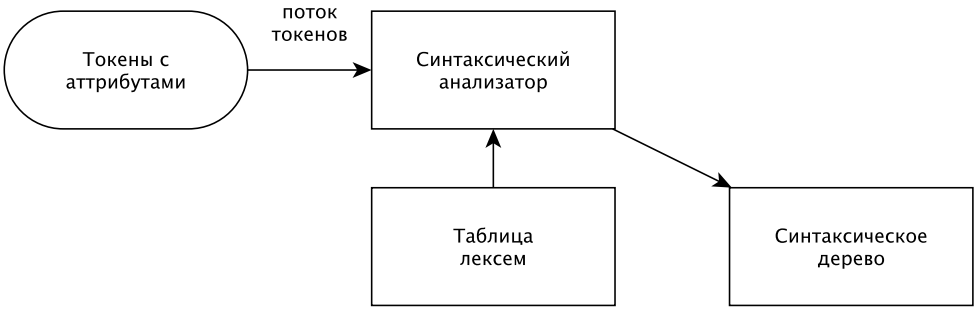

Рисунок 1.2 – Синтаксический анализатор

1.4 Семантический анализатор

Семантический анализатор использует синтаксическое дерево и информацию из таблицы символов для проверки исходной программы на семантическую согласованность с определением языка. Он также собирает информацию о типах и сохраняет все в синтаксическом дереве или в таблице символов для последующего использования в процессе генерации промежуточного кода.

Важной частью семантического анализа является проверка типов, когда компилятор проверяет, имеет ли каждый оператор операнды соответствующего типа.

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.

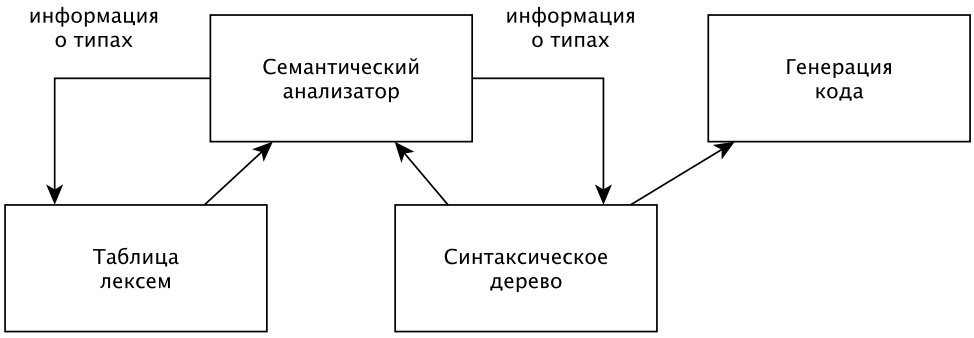

Рисунок 1.3 – Семантический анализатор

1.5 Генерация кода

1.5.1 Генерация промежуточного кода

В процессе трансляции исходной программы в целевой код компилятор может создавать одно или несколько промежуточных представлений различного вида. Синтаксические деревья являются видом промежуточного представления. Обычно они используются в процессе синтаксического и семантического анализа.

После синтаксического и семантического анализа исходной программы многие компиляторы генерируют явное низкоуровневое или машинное промежуточное представление исходной программы, которое можно рассматривать как программу для абстрактной вычислительной машины. Такое промежуточное представление должно обладать двумя важными свойствами: оно должно легко генерироваться и легко транслироваться в целевой машинный язык.

1.5.2 Оптимизация

Фаза машинно-независимой оптимизации кода пытается улучшить промежуточный код, чтобы затем получить более качественный целевой код. Например, более быстрый код, короткий код, или код, использующий меньшее количество ресурсов.

1.5.3 Генерация кода на целевом языке

Генератор кода получает в качестве входных данных промежуточное представление исходной программы и отображает его в целевой язык. Если

целевой язык представляет собой машинный код, для каждой переменной, используемой программой, выбираются соответствующие регистры или ячейки памяти. Затем промежуточные команды транслируются в последовательности машинных команд, выполняющих те же действия.

1.6 Таблица символов

Важная функция компилятора состоит в том, чтобы записывать имена переменных в исходной программе и накапливать информацию о разных атрибутах каждого имени. Эти атрибуты могут предоставлять информацию о выделенной памяти для данного имени, его типе, области видимости и, в случае имен процедур, такие сведения, как количество и типы их аргументов, метод передачи каждого аргумента, а также возвращаемый тип. Таблица символов представляет собой структуру данных, содержащую записи для каждого имени переменной, с полями для атрибутов имени.

1.7 Синтаксическое дерево

Синтаксическое дерево — дерево, в котором каждый внутренний узел представляет операцию, а дочерние узлы — аргументы этой операции. Порядок операций в дереве согласуется с обычными правилами, например, умножение имеет более высокий приоритет, чем сложение, и должно быть выполнено до сложения.

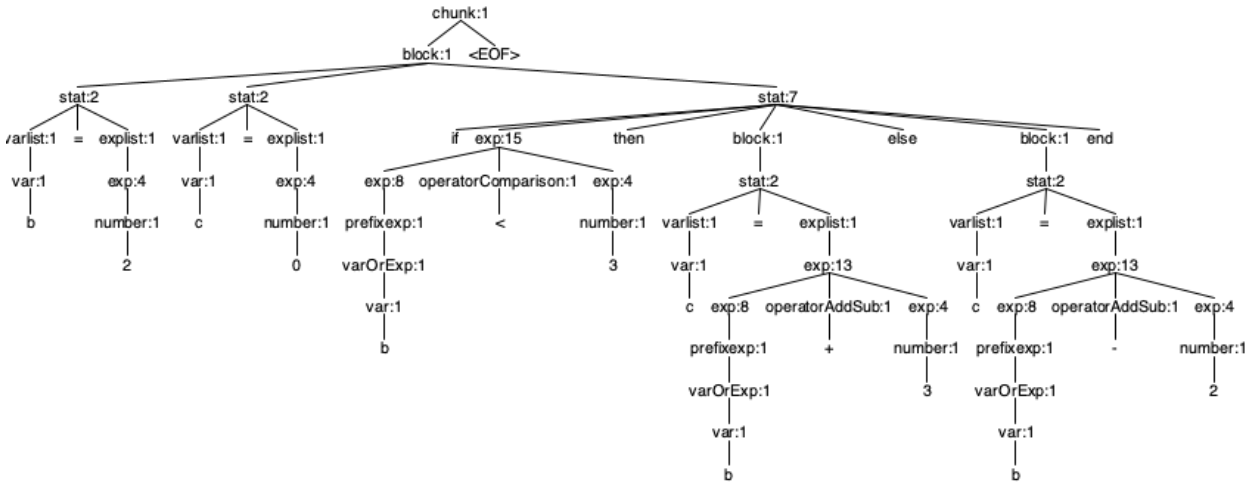

Рисунок 1.4 – Пример синтаксического дерева

1.8 Генераторы лексических анализаторов

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

Lex — стандартный инструмент для получения лексических анализаторов в операционных системах Unix [3]. В результате обработки входного потока получается исходный файл на языке C. Lex-файл разделяется на три блока: блок определений, правил и кода на C.

Flex заменяет Lex в системах на базе пакетов GNU и имеет аналогичную функциональность [4].

ANTLR (ANother Tool for Language Recognition) — генератор лексических и синтаксических анализаторов, позволяет создавать анализаторы на таких языках, как: Java, C#, Go, C++ и других [5].

ANTLR генерирует классы нисходящего рекурсивного синтаксического анализатора, на основе правил, заданных грамматикой.

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.

1.9 Генераторы синтаксических анализаторов

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 [6].

Coco/R — генератор лексических и синтаксических анализаторов [7]. Лексические анализаторы работают по принципу конечных автоматов, а синтаксические используют рекурсивный спуск. Поддерживаются такие языки программирования, как C, C#, Java и другие.

Yacc принимает на вход контекстно-свободную грамматику и использует LALR-разбор (LR-разбор с предпросмотром). Канонические LR-анализаторы имеют незначительно большую распознавающую способность, чем LALR-

анализаторы, однако требуют намного больше памяти для таблиц, поэтому их используют очень редко.

ANTLR и Coco/R принимают на вход контекстно-свободную грамматику, ANTLR использует LL(*)-разбор, а также умеет работать с левой рекурсией, чего не могут обычные LL-анализаторы, Coco/R использует LL(1)-разбор.

LL-анализатор называется LL(k)-анализатором, если данный анализатор использует предпросмотр на k токенов при разборе входного потока.

LR-анализаторы, в отличие от LL-анализаторов, строящих левосторонний вывод, производят наиболее правую продукцию контекстно-свободной грамматики. LR-анализ может применяться к большему количеству языков, чем LL-анализ, а также лучше в части сообщения об ошибках, то есть он определяет синтаксические ошибки там, где вход не соответствует грамматике, как можно раньше. В отличие от этого, LL(k) анализаторы могут задерживать определение ошибки до другой ветки грамматики из-за отката, часто затрудняя определение места ошибки в местах общих длинных префиксов. Однако, в большинстве случаев LL-разбор работает быстрее LR-разбора, а LR-анализаторы и построение их таблиц сложнее в реализации.

1.10 LLVM

LLVM (Low Level Virtual Machine) — проект программной инфраструктуры для создания компиляторов и сопутствующих им утилит [8]. В его основе лежит платформонезависимая система кодирования машинных инструкций — байткод LLVM IR. LLVM может создавать байткод для множества платформ, включая ARM, x86, x86-64, GPU от AMD и Nvidia и другие. 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.

LLVM поддерживает целые числа произвольной разрядности, числа с плавающей точкой, массивы, структуры и функции. Большинство инструкций в LLVM принимает два аргумента и возвращает одно значение. Значения в LLVM определяются текстовым идентификатором. Тип операндов всегда указывается явно и однозначно определяет тип результата. Операнды арифметических инструкций должны иметь одинаковый тип, но сами ин-

струкции «перегружены» для любых числовых типов и векторов. 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. 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. 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.

1.11 Вывод

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 Конструкторский раздел

2.1 Концептуальная модель разрабатываемого компилятора

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

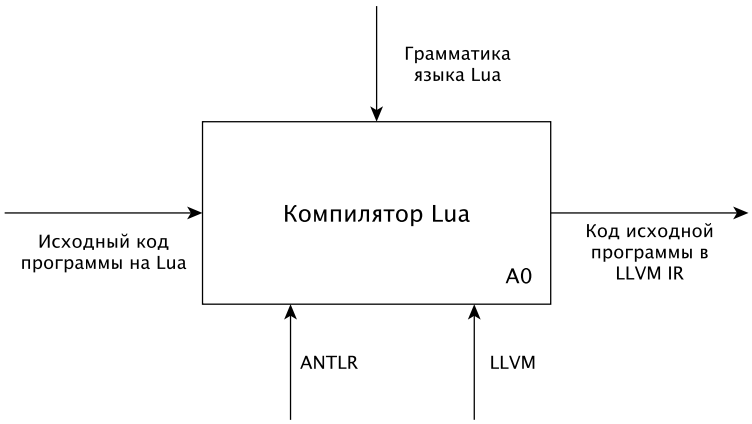

Рисунок 2.1 – Концептуальная модель разрабатываемого компилятора в нотации IDEF0

2.2 Язык Lua

Lua — встраиваемый язык сценариев. Он поддерживает процедурное программирование, объектно-ориентированное программирование, функциональное программирование, программирование, управляемое данными, и описание данных. Lua сочетает в себе процедурный синтаксис с конструкциями описания данных, основанными на ассоциативных массивах и расширяемой семантике. Lua динамически типизирован, работает путем интерпретации байт-кода с помощью виртуальной машины на основе регистров и имеет автоматическое управление памятью с инкрементальной сборкой мусора, что делает его идеальным для конфигурирования, написания сценариев и быстрого прототипирования.

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.

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. В данной работе используется паттерн Visitor.

2.5 Вывод

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

3 Технологический раздел

3.1 Выбор средств программной реализации

В качестве языка программирования был выбран язык Go, ввиду следующих причин.

— Компилятор, написанный на Go может быть запущен на различных платформах.

— Для языка Go существуют библиотеки, позволяющие генерировать код для LLVM.

— Выбранный генератор лексических и синтаксических анализаторов ANTLR позволяет генерировать код на Go.

3.2 Основные компоненты программы

В результате работы ANTLR были сгенерированы интерфейсы для Visitor и Listener, файлы с данными для интерпретатора ANTLR, файлы с токенами и реализации анализаторов.

На языке Go был реализован интерфейс Visitor, так как, несмотря на большие требования по контролю за обходом дерева, он предоставляет больше гибкости в анализе дерева и позволяет возвращать значения из обработчиков. Для каждой семантической единицы была прописана собственная логика анализа.

Листинг 3.1 – Пример реализации посещения для оператора логического "или"

```
1 func (v *Visitor) VisitExpOperatorOr(ctx
     *parser.ExpOperatorOrContext) interface{} {
2     values := make([]value.Value, len(ctx.AllExp()))
3     for i, e := range ctx.AllExp() {
4         val, ok := v.Visit(e).(value.Value)
5         if !ok { // обработка ошибки }
6         values[i] = val
7     }
8     return v.currentEntry().NewCall(v.funcs["or"], values[0],
        values[1])
9 }
```

Примеры программ на Lua и соответствующий им результат работы компилятора на LLVM IR представлены в Приложении Б.

Язык Lua имеет обладает специфическими особенностями, что вызвало необходимость в реализации дополнительного функционала. Далее представлено более подробное описание особенностей Lua.

Динамическая типизация

Язык Lua динамически типизирован, а Go и LLVM IR — строго типизированы. Чтобы компилятор поддерживал работу с динамическими типами, был реализован специальный тип Generic.

Листинг 3.2 – Реализация типа Generic на LLVM IR

```
%Generic = type {
    i32,      ; тип данных (0=int, 1=float, 2=string, 3=bool,
       4=table, 5=nil)
    i8*       ; указатель на данные
  }
```

Этот тип хранит в себе идентификатор одного из доступных типов данных и указатель на область памяти со значением. Значение любого типа языка Lua, преобразуется в переменную типа Generic на LLVM IR.

Go и LLVM IR, в отличие от Lua, не позволяют реализовывать операции для работы с разными типами данных. Например, нельзя напрямую сравнить целое число с вещественным. Для поддержания такого функционала для каждой функции, которую необходимо применять к операндам разного типа, было реализовано множество подфункций для работы с каждой комбинацией типов операндов, внутри которой операнды приводятся к одному типу, когда это необходимо и возможно.

Работа с таблицами

В языке Lua таблицы, массивы и структуры представлены одним типом таблиц или ассоциативных массивов.

Листинг 3.3 – Таблицы, массивы и структуры в языке Lua

```
-- массив
days1 = {"понедельник", "вторник", "среда", "четверг"}
-- структура
person = {fio = "Иванов Степан Васильевич",
          post = "слесарь-инструментальщик",
          bdate = "08.08.1973"}
-- таблица
```

```
workDays = {["понедельник"]=true, ["вторник"]=true,
   ["среда"]=true}
```

С помощью типа Generic, описанного ранее, также можно представить и таблицу языка Lua. На LLVM IR были реализованы стандартные функции для работы с таблицами: создание объекта таблицы, добавление элемента, получение элемента по ключу и пр.

Возврат значений из функций

Язык Lua, как и многие другие языки, поддерживает определение функций, не возвращающих никаких значений. В LLVM IR функция всегда должна возвращать ровно одно значение. Для реализации функционала по возврату нескольких значений из функции была предусмотрена возможность возврата таблицы как набора возвращаемых функцией значений, формально представленным одним значением. Если функция на Lua не возвращает значений, то из функции на LLVM IR возвращается значение null.

Листинг 3.4 – Реализация функции создания null-переменной на LLVM IR

```
define %Generic* @create_nil() {
entry:
  %size = ptrtoint %Generic* getelementptr inbounds (%Generic,
     %Generic* null, i32 1) to i64
  %nil = call i8* @malloc(i64 %size)
  %nil_generic = bitcast i8* %nil to %Generic*

  %type_ptr = getelementptr inbounds %Generic, %Generic*
     %nil_generic, i32 0, i32 0
  store i32 5, i32* %type_ptr, align 4
  %data_ptr = getelementptr inbounds %Generic, %Generic*
     %nil_generic, i32 0, i32 1
  store i8* null, i8** %data_ptr, align 8

  ret %Generic* %nil_generic
}
```

Возврат ошибок

Для вывода ошибок в процессе компиляции в структуру Visitor был добавлен список ошибок, который наполняется в ходе компиляции и выводится в стандартный поток ввода-вывода. Также, реализована проверка областей видимости переменных: в структуру Visitor добавлен словарь объявленных на текущий момент переменных, содержание которого меняется в зависимости

от анализируемого блока программы.

Для обработки ошибочных ситуаций в процессе исполнения скомпилированного кода была реализована функция panic, которая обеспечивает вывод причины сбоя в программе в стандартный поток ввода-вывода. Например, такой ошибочной ситуацией может быть передача целочисленного операнда функции конкатенации строк.

Листинг 3.5 – Реализация функции panic на LLVM IR

```
define void @panic(i8* %msg) {
entry:
  %fmt = getelementptr inbounds [46 x i8], [46 x i8]*
     @.str.error, i32 0, i32 0
  call i32 (i8*, ...) @printf(i8* %fmt, i8* %msg)
  call void @exit(i64 1)
  ret void
}
```

Базовые функции языка

На LLVM IR были написаны базовые функции. В том числе:

— реализация арифметических операций;

— реализация логических операций;

— реализация операций сравнения;

— реализация функций для строкового типа: конкатенация, получение длины;

— реализация функций для таблиц, массивов, структур: создание объекта, добавление элемента, получение элемента по ключу и пр.;

— реализация функции вывода в стандартный поток ввода-вывода.

3.3 Тестирование

Было проведено тестирование работы компилятора для базовых конструкций языка Lua в соответствии с грамматикой.

Тестовые коды программ представлены в Приложении В.

Ниже представлен пример работы программы.

Листинг 3.6 – Пример исходного кода на Lua

```
b = 2

if b < 3 then
    b = b + 3
end

print(b)
```

Листинг 3.7 – Результат работы программы для примера

```
define i64 @main() {
0:
    %1 = call %Generic* @create(i32 0, i8* inttoptr (i64 2 to
       i8*))
    %2 = call %Generic* @create(i32 0, i8* inttoptr (i64 3 to
       i8*))
    %3 = call %Generic* @lt(%Generic* %1, %Generic* %2)
    %4 = call i1 @check(%Generic* %3)
    br i1 %4, label %6, label %5

5:
    call void @print(%Generic* %1)
    ret i64 0

6:
    %7 = call %Generic* @create(i32 0, i8* inttoptr (i64 3 to
       i8*))
    %8 = call %Generic* @add(%Generic* %1, %Generic* %7)
    call void @copy(%Generic* %8, %Generic* %1)
    br label %5
}
```

В результате исполнения программы в стандартный поток ввода-вывода было выведено число 5.

3.4 Вывод

В данном разделе был описан выбор средств программной реализации, описано тестирование разработанной программы и приведены примеры работы компилятора.

ЗАКЛЮЧЕНИЕ

В ходе выполнения курсовой работы были решены следующие задачи:

1) Проанализирована грамматика языка Lua и выделены ее ключевые составляющие.

2) Изучены существующие средства для анализа исходных кодов программ, системы для генерации низкоуровневого кода, запуск которого возможен на большинстве из используемых платформ и операционных систем.

3) Разработан прототип компилятора на языке Go, выполняющий синтаксический анализ исходного текста программы и построение абстрактного синтаксического дерева.

4) Проведено преобразование абстрактного синтаксического дерева в IR с использованием LLVM.

Таким образом, была достигнута цель работы: была разработан компилятор языка Lua.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. *Шамашов М.* Основные структуры данных и алгоритмы компиляции // Учебное пособие. — 1999. — С. 5—6.
2. *Alfred V. A.*, *Monica S. L.*, *Jeffrey D. U.* Compilers principles, techniques & tools. — pearson Education, 2007.
3. *Lesk M. E.*, *Schmidt E.* Lex: A lexical analyzer generator. Т. 39. — Bell Laboratories Murray Hill, NJ, 1975.
4. How to test program generators? A case study using flex / P. Sampath [и др.] // Fifth IEEE International Conference on Software Engineering and Formal Methods (SEFM 2007). — IEEE. 2007.
5. What's ANTLR / T. Parr [и др.]. — 2004.
6. *C. D.* BISON the YACC-compatible parser generator // Technical report. — 1988. — № 2.
7. *Mössenböck H.*, *Wöss A.*, *Löberbauer M.* Der Compilergenerator Coco/R. — na, 2003.
8. *Sarda S.*, *Pandey M.* LLVM essentials. — Packt Publishing Ltd, 2015.

ПРИЛОЖЕНИЕ А
Грамматика языка Lua

Листинг А.1 – Грамматика языка Lua

```
grammar Lua;

chunk
    : block EOF
    ;

block
    : stat* retstat?
    ;

stat
    : ';'

      # StatEmptySemicolon
    | varlist '=' explist

      # StatAssignment
    | functioncall

      # StatFunctionCall
    | 'do' block 'end'

      # StatDo
    | 'while' exp 'do' block 'end'
                                                          #
      StatWhile
    | 'repeat' block 'until' exp
                                                            #
      StatRepeat
    | 'if' exp 'then' block ('elseif' exp 'then' block)*
      ('else' block)? 'end'      # StatIfThenElse
    | 'for' NAME '=' exp ',' exp (',' exp)? 'do' block 'end'
                               # StatNumericFor
    | 'for' namelist 'in' explist 'do' block 'end'
                                        # StatGenericFor
    | 'function' funcname funcbody
                                                          #
```

```
       StatFunction
22     ;
23
24 attnamelist
25     : NAME attrib (',' NAME attrib)*
26     ;
27
28 attrib
29     : ('<' NAME '>')?
30     ;
31
32 retstat
33     : 'return' explist? ';'?                                          #
          StatReturn
34     ;
35
36 funcname
37     : NAME ('.' NAME)* (':' NAME)?
38     ;
39
40 varlist
41     : var (',' var)*
42     ;
43
44 namelist
45     : NAME (',' NAME)*
46     ;
47
48 explist
49     : exp (',' exp)*
50     ;
51
52 exp
53     : 'nil'
          # ExpNil
54     | 'false'
          # ExpFalse
55     | 'true'
```

```
        # ExpTrue
56    | number

        # ExpNumber
57    | string

        # ExpString
58    | functiondef

        # ExpFunctionDef
59    | prefixexp

        # ExpPrefixExp
60    | tableconstructor

        # ExpTableConstructor
61    | <assoc=right> exp operatorPower exp
                                                    #
        ExpOperatorPower
62    | operatorUnary exp

        # ExpOperatorUnary
63    | exp operatorMulDivMod exp
                                                          #
        ExpOperatorMulDivMod
64    | exp operatorAddSub exp
                                                             #
        ExpOperatorAddSub
65    | <assoc=right> exp operatorStrcat exp
                                                 #
        ExpOperatorStrcat
66    | exp operatorComparison exp
                                                       #
        ExpOperatorComparison
67    | exp operatorAnd exp

        # ExpOperatorAnd
68    | exp operatorOr exp

        # ExpOperatorOr
```

```
69      ;
70
71  prefixexp
72      : varOrExp nameAndArgs*
73      ;
74
75  functioncall
76      : varOrExp nameAndArgs+
77      ;
78
79  varOrExp
80      : var | '(' exp ')'
81      ;
82
83  var
84      : (NAME | '(' exp ')' varSuffix) varSuffix*
85      ;
86
87  varSuffix
88      : nameAndArgs* ('[' exp ']' | '.' NAME)
89      ;
90
91  nameAndArgs
92      : (':' NAME)? args
93      ;
94
95  args
96      : '(' explist? ')' | tableconstructor | string
97      ;
98
99  functiondef
100     : 'function' funcbody
101     ;
102
103 funcbody
104     : '(' parlist? ')' block 'end'
105     ;
106
107 parlist
108     : namelist (',' '...')? | '...'
109     ;
```

```
110
111 tableconstructor
112     : '{' fieldlist? '}'
113     ;
114
115 fieldlist
116     : field (fieldsep field)* fieldsep?
117     ;
118
119 field
120     : '[' exp ']' '=' exp | NAME '=' exp | exp
121     ;
122
123 fieldsep
124     : ',' | ';'
125     ;
126
127 operatorOr
128     : 'or';
129
130 operatorAnd
131     : 'and';
132
133 operatorComparison
134     : '<' | '>' | '<=' | '>=' | '~=' | '==';
135
136 operatorStrcat
137     : '..';
138
139 operatorAddSub
140     : '+' | '-';
141
142 operatorMulDivMod
143     : '*' | '/' | '%' | '//';
144
145 operatorUnary
146     : 'not' | '#' | '-';
147
148 operatorPower
149     : '^';
150
```

```
151 number
152     : INT | FLOAT
153     ;
154
155 string
156     : NORMALSTRING
157     ;
158
159 // LEXER
160
161 NAME
162     : [a-zA-Z_][a-zA-Z_0-9]*
163     ;
164
165 NORMALSTRING
166     : '"' ( EscapeSequence | ~('\\'|'"') )* '"'
167     ;
168
169 fragment
170 NESTED_STR
171     : '=' NESTED_STR '='
172     | '[' .*? ']'
173     ;
174
175 INT
176     : Digit+
177     ;
178
179 FLOAT
180     : Digit+ '.' Digit* ExponentPart?
181     | '.' Digit+ ExponentPart?
182     | Digit+ ExponentPart
183     ;
184
185 fragment
186 ExponentPart
187     : [eE] [+-]? Digit+
188     ;
189
190 fragment
191 EscapeSequence
```

```
192     : '\\' [abfnrtvz"'\\]
193     | '\\' '\r'? '\n'
194     | DecimalEscape
195     | HexEscape
196     | UtfEscape
197     ;
198
199 fragment
200 DecimalEscape
201     : '\\' Digit
202     | '\\' Digit Digit
203     | '\\' [0-2] Digit Digit
204     ;
205
206 fragment
207 HexEscape
208     : '\\' 'x' HexDigit HexDigit
209     ;
210
211 fragment
212 UtfEscape
213     : '\\' 'u{' HexDigit+ '}'
214     ;
215
216 fragment
217 Digit
218     : [0-9]
219     ;
220
221 fragment
222 HexDigit
223     : [0-9a-fA-F]
224     ;
225
226 COMMENT
227     : '--[' NESTED_STR ']' -> channel(HIDDEN)
228     ;
229
230 LINE_COMMENT
231     : '--'
232     (                                                    // --
```

```
233     | '[' '='*                                                  // --[==
234     | '[' '='* ~('='|'['|'\r'|'\n') ~('\r'|'\n')*              // --[==AA
235     | ~('['|'\r'|'\n') ~('\r'|'\n')*                           // --AAA
236     ) ('\r\n'|'\r'|'\n'|EOF)
237     -> channel(HIDDEN)
238     ;
239
240 WS
241     : [ \t\u000C\r\n]+ -> skip
242     ;
243
244 SHEBANG
245     : '#' '!' ~('\n'|'\r')* -> channel(HIDDEN)
246     ;
```

ПРИЛОЖЕНИЕ Б
Тестовые программы с результатом на LLVM IR

Листинг Б.1 – Вычисление n-го числа Фибоначчи на Lua

```
function fibonacci(n)
    a = 0
    b = 1

    for i = 0, n, 1 do
        a, b = b, a + b
    end

    return a
end

print(fibonacci(7))
```

Листинг Б.2 – Вычисление n-го числа Фибоначчи на LLVM IR

```
define i64 @main() {
0:
    %1 = call %Generic* @create(i32 0, i8* inttoptr (i64 7 to
       i8*))
    %2 = call %Generic* @fibonacci(%Generic* %1)
    call void @print(%Generic* %2)
    ret i64 0
}

define %Generic* @fibonacci(%Generic* %n) {
0:
    %1 = call %Generic* @create(i32 0, i8* inttoptr (i64 0 to
       i8*))
    %2 = call %Generic* @create(i32 0, i8* inttoptr (i64 1 to
       i8*))
    %3 = call %Generic* @create(i32 0, i8* inttoptr (i64 0 to
       i8*))
    %4 = call %Generic* @create(i32 0, i8* inttoptr (i64 0 to
       i8*))
    %5 = call %Generic* @create(i32 0, i8* inttoptr (i64 1 to
       i8*))
    br label %6

```

```
6:
    %7 = call %Generic* @lt(%Generic* %5, %Generic* %3)
    %8 = call i1 @check(%Generic* %7)
    br i1 %8, label %17, label %14

9:
    ret %Generic* %1

10:
    %11 = call %Generic* @add(%Generic* %4, %Generic* %5)
    call void @copy(%Generic* %11, %Generic* %4)
    br label %6

12:
    %13 = call %Generic* @add(%Generic* %1, %Generic* %2)
    call void @copy(%Generic* %2, %Generic* %1)
    call void @copy(%Generic* %13, %Generic* %2)
    br label %10

14:
    %15 = call %Generic* @lt(%Generic* %4, %Generic* %n)
    %16 = call i1 @check(%Generic* %15)
    br i1 %16, label %12, label %9

17:
    %18 = call %Generic* @gt(%Generic* %4, %Generic* %n)
    %19 = call i1 @check(%Generic* %18)
    br i1 %19, label %12, label %9
}
```

Листинг Б.3 – Вычисление n-го числа Фибоначчи с рекурсией на Lua

```
function fibonacci(n)
    if n<3 then
        return 1
    else
        return fibonacci(n-1) + fibonacci(n-2)
    end
end

print(fibonacci(7))
```

Листинг Б.4 – Вычисление n-го числа Фибоначчи с рекурсией на LLVM IR

```
define i64 @main() {
0:
    %1 = call %Generic* @create(i32 0, i8* inttoptr (i64 7 to
       i8*))
    %2 = call %Generic* @fibonacci(%Generic* %1)
    call void @print(%Generic* %2)
    ret i64 0
}

define %Generic* @fibonacci(%Generic* %n) {
0:
    %1 = call %Generic* @create(i32 0, i8* inttoptr (i64 3 to
       i8*))
    %2 = call %Generic* @lt(%Generic* %n, %Generic* %1)
    %3 = call i1 @check(%Generic* %2)
    br i1 %3, label %7, label %4

4:
    br label %9

5:
    %6 = call %Generic* @create_nil()
    ret %Generic* %6

7:
    %8 = call %Generic* @create(i32 0, i8* inttoptr (i64 1 to
       i8*))
    ret %Generic* %8

9:
    %10 = call %Generic* @create(i32 0, i8* inttoptr (i64 1 to
       i8*))
    %11 = call %Generic* @sub(%Generic* %n, %Generic* %10)
    %12 = call %Generic* @fibonacci(%Generic* %11)
    %13 = call %Generic* @create(i32 0, i8* inttoptr (i64 2 to
       i8*))
    %14 = call %Generic* @sub(%Generic* %n, %Generic* %13)
    %15 = call %Generic* @fibonacci(%Generic* %14)
    %16 = call %Generic* @add(%Generic* %12, %Generic* %15)
    ret %Generic* %16
}
```

Листинг Б.5 – Инвертирование списка на Lua

```
function revert_array(arr)
    l = 0
    for i, v in arr do
        l = l + 1
    end

    res = {}
    cur = 0
    for i = l-1, -1, -1 do
        res[cur] = arr[i]
        cur = cur + 1
    end

    return res
end

inp = {0,1,2,3,4}
print(revert_array(inp))
```

Листинг Б.6 – Инвертирование списка на LLVM IR

```
define i64 @main() {
0:
    %1 = call %Generic* @lua_table_new()
    %2 = call %Generic* @create(i32 0, i8* inttoptr (i64 0 to
       i8*))
    %3 = call %Generic* @create(i32 0, i8* inttoptr (i64 0 to
       i8*))
    call void @lua_table_set(%Generic* %1, %Generic* %3,
       %Generic* %2)
    %4 = call %Generic* @create(i32 0, i8* inttoptr (i64 1 to
       i8*))
    %5 = call %Generic* @create(i32 0, i8* inttoptr (i64 1 to
       i8*))
    call void @lua_table_set(%Generic* %1, %Generic* %5,
       %Generic* %4)
    %6 = call %Generic* @create(i32 0, i8* inttoptr (i64 2 to
       i8*))
    %7 = call %Generic* @create(i32 0, i8* inttoptr (i64 2 to
       i8*))
    call void @lua_table_set(%Generic* %1, %Generic* %7,
```

```
        %Generic* %6)
13      %8 = call %Generic* @create(i32 0, i8* inttoptr (i64 3 to
            i8*))
14      %9 = call %Generic* @create(i32 0, i8* inttoptr (i64 3 to
            i8*))
15      call void @lua_table_set(%Generic* %1, %Generic* %9,
            %Generic* %8)
16      %10 = call %Generic* @create(i32 0, i8* inttoptr (i64 4 to
            i8*))
17      %11 = call %Generic* @create(i32 0, i8* inttoptr (i64 4 to
            i8*))
18      call void @lua_table_set(%Generic* %1, %Generic* %11,
            %Generic* %10)
19      %12 = call %Generic* @revert_array(%Generic* %1)
20      call void @print(%Generic* %12)
21      ret i64 0
22  }
23
24  define %Generic* @revert_array(%Generic* %arr) {
25  0:
26      %1 = call %Generic* @create(i32 0, i8* inttoptr (i64 0 to
            i8*))
27      %2 = call %Generic* @create(i32 0, i8* inttoptr (i64 0 to
            i8*))
28      %3 = call %Generic* @create(i32 0, i8* inttoptr (i64 1 to
            i8*))
29      %4 = call %Generic* @create_nil()
30      %5 = call %Generic* @create_nil()
31      br label %19
32
33  6:
34      %7 = call %Generic* @lua_table_new()
35      %8 = call %Generic* @create(i32 0, i8* inttoptr (i64 0 to
            i8*))
36      %9 = call %Generic* @create(i32 0, i8* inttoptr (i64 0 to
            i8*))
37      %10 = call %Generic* @create(i32 0, i8* inttoptr (i64 1 to
            i8*))
38      %11 = call %Generic* @sub(%Generic* %1, %Generic* %10)
39      %12 = call %Generic* @create(i32 0, i8* inttoptr (i64 1 to
            i8*))
```

```
40      %13 = call %Generic* @neg(%Generic* %12)
41      %14 = call %Generic* @create(i32 0, i8* inttoptr (i64 1 to
            i8*))
42      %15 = call %Generic* @neg(%Generic* %14)
43      br label %28
44
45  16:
46      %17 = call %Generic* @lua_table_get_key_at(%Generic* %arr,
            %Generic* %2)
47      %18 = call %Generic* @lua_table_get_value_at(%Generic*
            %arr, %Generic* %2)
48      call void @copy(%Generic* %17, %Generic* %4)
49      call void @copy(%Generic* %18, %Generic* %5)
50      br label %25
51
52  19:
53      %20 = call %Generic* @lua_table_len(%Generic* %arr)
54      %21 = call %Generic* @ge(%Generic* %2, %Generic* %20)
55      %22 = call i1 @check(%Generic* %21)
56      br i1 %22, label %6, label %16
57
58  23:
59      %24 = call %Generic* @add(%Generic* %2, %Generic* %3)
60      call void @copy(%Generic* %24, %Generic* %2)
61      br label %19
62
63  25:
64      %26 = call %Generic* @create(i32 0, i8* inttoptr (i64 1 to
            i8*))
65      %27 = call %Generic* @add(%Generic* %1, %Generic* %26)
66      call void @copy(%Generic* %27, %Generic* %1)
67      br label %23
68
69  28:
70      %29 = call %Generic* @lt(%Generic* %15, %Generic* %9)
71      %30 = call i1 @check(%Generic* %29)
72      br i1 %30, label %41, label %38
73
74  31:
75      ret %Generic* %7
76
```

```
32:
    %33 = call %Generic* @add(%Generic* %11, %Generic* %15)
    call void @copy(%Generic* %33, %Generic* %11)
    br label %28

34:
    %35 = call %Generic* @lua_table_get(%Generic* %arr,
      %Generic* %11)
    call void @lua_table_set(%Generic* %7, %Generic* %8,
      %Generic* %35)
    %36 = call %Generic* @create(i32 0, i8* inttoptr (i64 1 to
      i8*))
    %37 = call %Generic* @add(%Generic* %8, %Generic* %36)
    call void @copy(%Generic* %37, %Generic* %8)
    br label %32

38:
    %39 = call %Generic* @lt(%Generic* %11, %Generic* %13)
    %40 = call i1 @check(%Generic* %39)
    br i1 %40, label %34, label %31

41:
    %42 = call %Generic* @gt(%Generic* %11, %Generic* %13)
    %43 = call i1 @check(%Generic* %42)
    br i1 %43, label %34, label %31
}
```

ПРИЛОЖЕНИЕ В
Тестовые программы

Листинг В.1 – Тестовая программа 1

```
a = "hello \x43\u{2601}\43"
print(a)

print(5 * 8)
a, b = 5, 6.0
print((a - 5) * (b - 1))
print(5 * -8)
print(5 / 8)
print(9 % 8)
print(25 ^ -0.5)
print(true or false and false or false)
```

Листинг В.2 – Тестовая программа 2

```
a = {}     -- create a table and store its reference in ‘a’
k = "x"
a[k] = 10         -- new entry, with key="x" and value=10
a[20] = "great"  -- new entry, with key=20 and value="great"
print(a["x"])     --> 10
k = 20
print(a[k])       --> "great"
a["x"] = a["x"] + 1     -- increments entry "x"
print(a["x"])     --> 11

a[20] = 30

for i, v in a do
    a[i] = a[i] + 1
end

print(a)

for i in a do
    print(i)
end

for i = 10, 0, -3 do
    a[i] = i
```

```
25 end
26
27 print(a)
28
29 days1 = {"понедельник", "вторник", "среда", "четверг",
      "пятница", "суббота", "воскресенье"}
30
31 print(days1)
32
33 person = {tabnum = 123342,                    -- Табельный номер
34           fio = "Иванов Степан Васильевич",  -- Ф.И.О.
35           post = "слесарь-инструментальщик", -- Должность
36           salary = 25800.45,                 -- Оклад
37           sdate = "23.10.2013",              -- Дата приёма на
                работу
38           bdate = "08.08.1973"}              -- Дата рождения
39
40 person.fio = "Иванов Иван Иванович"
41 postfix="-гений-миллиардер"
42 person.post = person.post..postfix
43
44 for i, v in person do
45     print(i)
46     print(v)
47 end
48
49 print(person.aaabbb)
```

Листинг В.3 – Тестовая программа 3

```
1  function analyze_table(table)
2      for k, v in table do
3          if k > v then
4              v = v + 1
5          elseif k < v then
6              table[k] = table[k] - 1
7          else
8              table[k] = v + k
9          end
10     end
11
12     return table
13 end
```

```
14
15 t = {[-8]=-1, [1]=10, [6]=2, [5]=5, [7]=9}
16
17 analyze_table(t)
18 print(t)
19
20 a,b,c,d,e = analyze_table(t)
21 print(c)
22
23 function fact(n)
24     if n <= 0 then
25         return 1
26     else
27         return n*fact(n-1)
28     end
29 end
30
31 for k, v in t do
32     t[k] = fact(v)
33 end
34 print(t)
35
36 function foo(n)
37     return n, n + 1
38 end
39
40 a, b = foo(5)
41 print(a)
42 print(b)
```

Листинг B.4 – Тестовая программа 4

```
1 days1 = {"понедельник", "вторник", "среда", "четверг",
     "пятница", "суббота", "воскресенье"}
2 workDays = {["понедельник"]=true, ["вторник"]=true,
     ["среда"]=true, ["четверг"]=true, ["пятница"]=true,
     ["суббота"]=false, ["воскресенье"]=false}
3
4 i = 0
5 while i < 7 do
6     if workDays[days1[i]] then
7         days1[i] = days1[i].." - рабочий"
8     else
```

```
9           days1[i] = days1[i].." - выходной"
10      end
11      i = i + 1
12  end
13
14  print(days1)
15
16  days1 = {"понедельник", "вторник", "среда", "четверг",
        "пятница", "суббота", "воскресенье"}
17
18  i = i - 1
19
20  repeat
21      if workDays[days1[i]] and workDays[days1[i - 1]] then
22          print("с "..days1[i].." по "..days1[i - 1].." можно
                выспаться")
23      else
24          print("с "..days1[i].." по "..days1[i - 1].." нельзя
                выспаться")
25      end
26      i = i - 1
27  until i > 0
```