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**»

КУРСОВОЙ ПРОЕКТ

На тему: “Построение компилятора (транслятора)”

Пояснительная записка

Студент группы **ИУ7-12М** _____ **Р.Д. Третьяк**

(Подпись, дата) (И.О. Фамилия)

Преподаватель _____ **Ступников А.А.**

(Подпись, дата) (И.О. Фамилия)

***2022** г.*

Задание на курсовое проектирование

Используя фреймворк для создания парсеров ANTLR4 и программную инфраструктуру LLVM, создать компилятор языка программирования Golang с поддержкой следующего функционала:

1) создание и использование переменных;
2) типы данных double, int, array;
3) арифметические выражения;
4) логические выражения;
5) подпрограммы FUNC;
6) цикл FOR;
7) оператор ветвления IF-ELSE;
8) оператор printf, аналогичный fmt.Printf из стандартной библиотеки Golang;

В качестве языка промежуточного представления следует использовать LLVM IR, код которого будет скомпилирован компилятором LLC в бинарный исполняемый файл, являющийся результатом работы разрабатываемого компилятора.

Предоставляемые результаты работы включают:

1) исходный код компилятора на языке Python;
2) исходный код на языке Golang программы, использующей поддерживаемый разрабатываемым компилятором функционал, сортирующей массив целых чисел;
3) пояснительная записка.

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**»

РАССЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К КУРСОВОМУ ПРОЕКТУ

НА ТЕМУ:

”Компилятор языка Golang”

Студент группы **ИУ7-22М** _____ **Р.Д. Третьяк**

(Подпись, дата) (И.О. Фамилия)

Преподаватель _____ **Ступников А.А.**

(Подпись, дата) (И.О. Фамилия)

***2022** г.*

РЕФЕРАТ

Отчет по технологической практике по конструированию и созданию компиляторов 28 с., 1 рис., 4 источник., 2 прил.

Компилятор, Golang, Python, LLVM, LLVM IR, ANTLR4.

Объектом разработки является компилятор подмножества языка Golang.

Цель работы – разработать приложение, выполняющее генерацию байт кода для исполнения на виртуальной машине LLVM на основе исходного кода языка Golang.

В задачи проекта входили:

- разработка технического задания на курсовой проект;
- проектирование системы.

В результате работы был создан компилятор языка Golang. В разработке был применен объектно-ориентированный подход.

СОДЕРЖАНИЕ

1 Аналитический раздел

1.1 Лексический и синтаксический анализ

Задачей лексического анализа является аналитический разбора входной последовательности символов составляющих текст компилируемой программы с целью получения на выходе последовательности символов, называемых «токенами», которые характеризуются определенными типом и значением.

Лексический анализатор функционирует в соответствии с некоторыми правилами построения допустимых входных последовательностей. Данные правила могут быть определены, например,в виде детерминированного конечного автомата, регулярного выражения или праволинейной грамматики. С практической точки зрения наиболее удобным способом является формализация работы лексического анализатора с помощью грамматики.

Лексический анализ может быть представлен и как самостоятельная фаза трансляции, и как составная часть фазы синтаксического анализа. В первом случае лексический анализатор реализуется в виде отдельного модуля, который принимает последовательность символов, составляющих текст компилируемой программы, и выдаёт список обнаруженных лексем. Во втором случае лексический анализатор фактически является подпрограммой, вызываемой синтаксическим анализатором для получения очередной лексемы [1].

В процессе лексического анализа обнаруживаются лексические ошибки – простейшие ошибки компиляции, связанные с наличием в тексте программы недопустимых символов, некорректной записью идентификаторов, числовых констант и пр.

Синтаксический анализ, или разбор, как его еще называют, – это процесс сопоставления линейной последовательности токенов исходного языка с его формальной грамматикой. Результатом обычно является дерево разбора (или абстрактное синтаксическое дерево).

Синтаксический анализатор фиксирует синтаксические ошибки, т.е. ошиб-

ки, связанные с нарушением принятой структуры программы.

1.2 Методы реализации лексического и синтаксического анализаторов

Лексический и синтаксический анализаторы могут быть разработаны «с нуля» на основе существующих алгоритмов анализа, а могут быть созданы с помощью стандартных средств генерации анализаторов.

Существуют две основные стратегии синтаксического анализа: нисходящий анализ и восходящий анализ.

1.2.1 Алгоритмы лексического и синтаксического анализа

В нисходящем анализе дерево вывода цепочки строится от корня к листьям, т.е. дерево вывода «реконструируется» в прямом порядке, и аксиома грамматики «развертывается» в цепочку. В общем виде нисходящий анализ представлен в анализе методом рекурсивного спуска, который может использовать откаты, т.е. производить повторный просмотр считанных символов [2].

В восходящем анализе дерево вывода строится от листьев к корню и анализируемая цепочка «сворачивается» в аксиому. На каждом шаге свертки некоторая подстрока, соответствующая правой части продукции, замещается левым символом данной продукции. Примерами восходящих синтаксических анализаторов являются синтаксические анализаторы приоритета операторов, LR-анализат (SLR, LALR) [2].

1.2.2 Стандартные средства

Имеется множество различных стандартных средств для построения синтаксических анализаторов: Lex и Yacc, Coco/R, ANTLR, JavaCC и др. Генератор Yacc предназначен для построения синтаксического анализатора контекстно-свободного языка. Анализируемый язык описывается с помощью грамматики в виде, близком форме Бэкуса-Наура. Результатом работы Yacc’а является программа на Си, реализующая восходящий LALR(1) распознаватель. Как правило, Yacc используется в связке с Lex – стандартным генератором лексических анализаторов. Для обоих этих инструментов существуют свободные реализации – Bison и Flex.

Coco/R читает файл с атрибутивной грамматикой исходного языка в расширенной форме Бэкуса – Наура и создает файлы лексического и синтаксического анализаторов. Лексический анализатор работает как конечный автомат. Синтаксический анализатор использует методику нисходящего рекурсивного спуска.

ANTLR (ANother Tool for Language Recognition) – это генератор синтаксических анализаторов для чтения, обработки или трансляции как структурированных текстовых, так и бинарных файлов. ANTLR широко используется для разработки компиляторов, прикладных программных инструментов и утилит. На основе заданной грамматики языка ANTLR генерирует код синтаксического анализатора, который может строить абстрактное синтаксического дерево и производить его обход.

1.3 Построение грамматики

Для реализации было выбрано подмножество языка Golang, содержащее основные его возможности:

1) создание и использование переменных;
2) типы данных float, int, array;
3) арифметические выражения;
4) логические выражения;
5) подпрограммы func;
6) цикл for;
7) оператор ветвления if-else;
8) оператор printf, аналогичный fmt.Printf из стандартной библиотеки Golang;

.

1.3.1 Лексер

В листинге 1 представлена часть кода лексера, полный код находится в приложении А.

Листинг 1 – Лексер

```
lexer grammar GoLexer;

// Keywords

BREAK                   : 'break' -> mode(NLSEMI);
DEFAULT                 : 'default';
FUNC                    : 'func';
CASE                    : 'case';
MAP                     : 'map';
ELSE                    : 'else';
PACKAGE                 : 'package';
CONST                   : 'const';
IF                      : 'if';
RANGE                   : 'range';
TYPE                    : 'type';
CONTINUE                : 'continue' -> mode(NLSEMI);
FOR                     : 'for';
RETURN                  : 'return' -> mode(NLSEMI);
VAR                     : 'var';
```

1.3.2 Парсер

В листинге 2 представлена часть кода парсера, полный код находится в приложении Б.

Листинг 2 – Парсер

```
parser grammar GoParser;

options {
    tokenVocab = GoLexer;
    superClass = GoParserBase;
}

sourceFile:
    packageClause eos (
        (functionDecl | declaration) eos
    )* EOF;

packageClause: PACKAGE packageName = IDENTIFIER;

```

```
15 declaration: constDecl | typeDecl | varDecl;
16
17 constDecl: CONST (constSpec | L_PAREN (constSpec eos)* R_PAREN);
18
19 constSpec: identifierList (type_? ASSIGN expressionList)?;
```

1.4 Инструменты генерации синтаксического и лексического анализаторов

Выше были рассмотрены такие инструменты генерации синтаксических анализаторов: Lex и Yacc, Coco/R, ANTLR.

Принимая во внимание эффективность и простоту использования ANTLR, для построения кода синтаксического анализатора было решено применить данное средство [3].

1.5 Генерация кода

Генерация кода осуществляется с помощью нисхождения по абстрактному синтаксическому дереву, полученному в результате синтаксического анализа. Для каждого узла дерева определяется его тип и связанное с ним правило в грамматике языка. Исходя из этой информации генерируется промежуточный код на языке LLVM IR для исполнения на виртуальной машине или компиляции в исполняемый файл посредством использования инструмента LLC [4].

2 Конструкторский раздел

2.1 Структура компилятора

Компилятор состоит из 3-ех модулей:

- лексический анализатор, преобразовывающий текст программы в поток токенов;
- синтаксический анализатор, строящий AST-дерево;
- генератор LLRM IR кода.

2.2 Генерация лексического и синтаксического анализатора

В качестве входных данных для ANTLR выступает файл с описанием грамматики исходного языка. Данный файл содержит только правила грамматики без добавления кода, исполнение которого соответствует применению определённых правил. Подобное разделение позволяет использовать один и тот же файл грамматики для построения различных приложений (например, компиляторов, генерирующих код для различных сред исполнения).

На основе правил заданной грамматики языка ANTLR генерирует класс нисходящего рекурсивного синтаксического анализатора. Для каждого правила грамматики в полученном классе имеется свой рекурсивный метод. Разбор входной последовательности начинается с корня синтаксического дерева и заканчивается в листьях.

Сгенерированный ANTLR синтаксический анализатор выдаёт абстрактное синтаксическое дерево в чистом виде, и реализует методы для его построения и последующего обхода. Дерево разбора для заданной входной последовательности символов можно получить, вызвав метод, соответствующий аксиоме в исходной грамматике языка. В грамматике языка Golang аксиомой является нетерминал sourceFile, поэтому построение дерева следует начинать с вызова метода sourceFile() объекта класса синтаксического анализатора, являющегося корнем дерева.

2.3 Обнаружение и обработка лексических и синтаксических ошибок

Все ошибки, которые обнаруживаются лексическим и синтаксическим анализаторами ANTLR, по умолчанию выводятся в стандартный поток вывода ошибок. Данные ошибки возможно перехватить стандартным обработчиком ошибок языка на котором ведется разработка компилятора.

2.4 Генерация кода

Компилятор генерирует файл типа LLVM IR (Low Level Virtual Machine Intermediate representation), являющийся промежуточным представлением генерируемого кода для виртуальной машины LLVM.

Данный файл содержит инструкции описывающие ход выполнения программы на языке более высокого уровня, чем Ассемблер. Такой подход позволяет описывать код программы более естественным образом. LLVM IR позволяет зарегистрировать следующие типы данных:

- целые числа произвольной разрядности;
- числа с плавающей точкой: float, double, а также ряд типов, специфичных для конкретной платформы (например, x86_fp80);
- указатели;
- массивы;
- структуры;
- векторы;
- функции.

Большинство инструкций в LLVM принимают два аргумента(операнда) и возвращают одно значение(трёхадресный код). Значения определяются текстовым идентификатором. Локальные значения обозначаются префиксом %, а глобальные — @. Локальные значения также называют регистрами, а LLVM — виртуальной машиной с бесконечным числом регистров.

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.

Код модуля кодогенерации представлен на GitHub `https://github.com/LuckySting/pygoco`.

3 Использование компилятора

Рассмотрим работу компилятора на примере компиляции программы сортировки массива целых чисел на языке Golang. Исходный код представлен в листинге 3

Листинг 3 – Исходный код на языке Golang

```
1  package main
2
3  func bubble_sort(arr [5]int32) [5]int32 {
4      for i := 0; i < 5; i++ {
5          for j := 0; j < 5-i; j++ {
6              if (arr[j] < arr[j+1]) {
7                  temp := arr[j+1]
8                  arr[j+1] = arr[j]
9                  arr[j] = temp
10             }
11         }
12     }
13     return arr
14 }
15
16 func main() {
17     var arr [5]int32
18     arr[0] = 5
19     arr[1] = 3
20     arr[2] = 4
21     arr[3] = 1
22     arr[4] = 2
23     sorted_arr := bubble_sort(arr)
24     for i := 0; i < 5; i++ {
25         printf("%d -> %d\n", i, sorted_arr[i])
```

```
26          }
27  }
```

Абстрактное синтаксическое дерево для данной программы слишком большое, поэтому на рисунке 1 представлен пример AST для более простой программы, выводящей в стандартный поток вывода числа от 0 до 10.

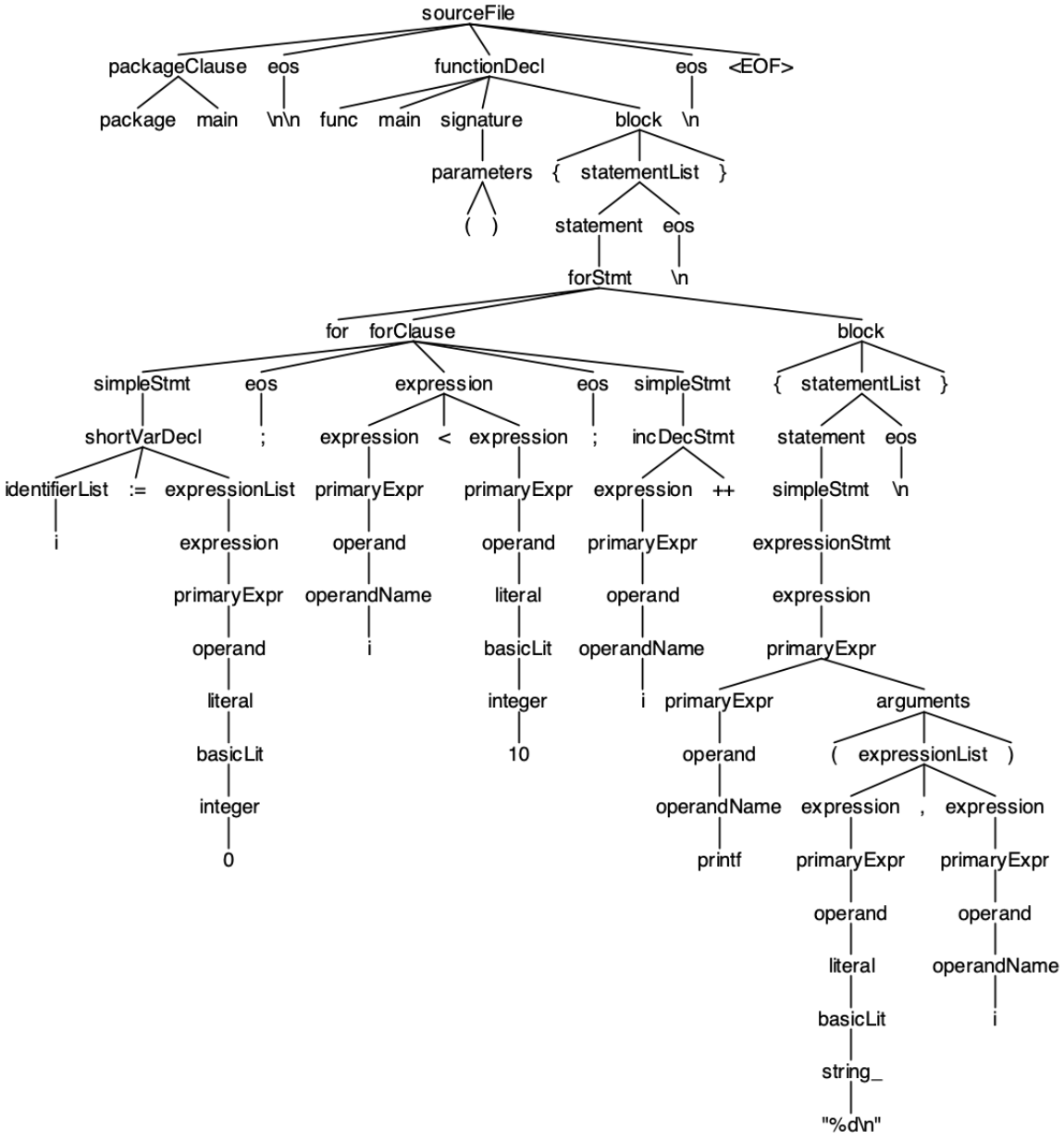

Рисунок 1 – AST программы печати чисел от 0 до 10

Результатом работы компилятора является код на языке промежуточного

представляения LLVM IR, он представлен в листинге 4.

Листинг 4 – Код промежуточного предствления LLVM IR

```
; ModuleID = "main"
target triple = "x86_64-apple-darwin19.6.0"
target datalayout = ""

declare i32 @"printf"(i8* %".1", ...)

define <5 x i32> @"bubble_sort"(<5 x i32> %"arr")
{
entry:
  %"arr.1" = alloca <5 x i32>
  store <5 x i32> %"arr", <5 x i32>* %"arr.1"
  %"i" = alloca i32
  store i32 0, i32* %"i"
  %"i.1" = load i32, i32* %"i"
  %".5" = icmp slt i32 %"i.1", 5
  br i1 %".5", label %"entry.if", label %"entry.endif"
for_loop:
  %"j" = alloca i32
  store i32 0, i32* %"j"
  %"i.2" = load i32, i32* %"i"
  %".9" = sub i32 5, %"i.2"
  %"j.1" = load i32, i32* %"j"
  %".10" = icmp slt i32 %"j.1", %".9"
  br i1 %".10", label %"for_loop.if", label %"for_loop.endif"
entry.if:
  br label %"for_loop"
entry.endif:
  %"arr.1.9" = load <5 x i32>, <5 x i32>* %"arr.1"
  ret <5 x i32> %"arr.1.9"
for_loop.1:
  %"j.2" = load i32, i32* %"j"
  %".13" = add i32 %"j.2", 1
  %"arr.1.1" = load <5 x i32>, <5 x i32>* %"arr.1"
  %"arr.1.1.1" = extractelement <5 x i32> %"arr.1.1", i32 %".13"
  %"arr.1.2" = load <5 x i32>, <5 x i32>* %"arr.1"
  %"j.3" = load i32, i32* %"j"
  %"arr.1.2.1" = extractelement <5 x i32> %"arr.1.2", i32 %"j.3"
```

```
38   %".14" = icmp slt i32 %"arr.1.2.1", %"arr.1.1.1"
39   br i1 %".14", label %"for_loop.1.if", label %"for_loop.1.endif"
40 for_loop.if:
41   br label %"for_loop.1"
42 for_loop.endif:
43   %"i.4" = load i32, i32* %"i"
44   %".29" = add i32 %"i.4", 1
45   store i32 %".29", i32* %"i"
46   %"i.5" = load i32, i32* %"i"
47   %".31" = icmp slt i32 %"i.5", 5
48   br i1 %".31", label %"for_loop.endif.if", label %"for_loop.endif.endif"
49 for_loop.1.if:
50   %"j.4" = load i32, i32* %"j"
51   %".16" = add i32 %"j.4", 1
52   %"arr.1.3" = load <5 x i32>, <5 x i32>* %"arr.1"
53   %"arr.1.3.1" = extractelement <5 x i32> %"arr.1.3", i32 %".16"
54   %"temp" = alloca i32
55   store i32 %"arr.1.3.1", i32* %"temp"
56   %"j.5" = load i32, i32* %"j"
57   %".18" = add i32 %"j.5", 1
58   %"arr.1.4" = load <5 x i32>, <5 x i32>* %"arr.1"
59   %"j.6" = load i32, i32* %"j"
60   %"arr.1.4.1" = extractelement <5 x i32> %"arr.1.4", i32 %"j.6"
61   %"arr.1.5" = load <5 x i32>, <5 x i32>* %"arr.1"
62   %"arr.1.6" = insertelement <5 x i32> %"arr.1.5", i32 %"arr.1.4.1", i32 %"
       .18"
63   store <5 x i32> %"arr.1.6", <5 x i32>* %"arr.1"
64   %"j.7" = load i32, i32* %"j"
65   %"temp.1" = load i32, i32* %"temp"
66   %"arr.1.7" = load <5 x i32>, <5 x i32>* %"arr.1"
67   %"arr.1.8" = insertelement <5 x i32> %"arr.1.7", i32 %"temp.1", i32 %"j.7"
68   store <5 x i32> %"arr.1.8", <5 x i32>* %"arr.1"
69   br label %"for_loop.1.endif"
70 for_loop.1.endif:
71   %"j.8" = load i32, i32* %"j"
72   %".22" = add i32 %"j.8", 1
73   store i32 %".22", i32* %"j"
74   %"i.3" = load i32, i32* %"i"
75   %".24" = sub i32 5, %"i.3"
76   %"j.9" = load i32, i32* %"j"
```

```
  %".25" = icmp slt i32 %"j.9", %".24"
  br i1 %".25", label %"for_loop.1.endif.if", label %"for_loop.1.endif.endif"
for_loop.1.endif.if:
  br label %"for_loop.1"
for_loop.1.endif.endif:
  br label %"for_loop.endif"
for_loop.endif.if:
  br label %"for_loop"
for_loop.endif.endif:
  br label %"entry.endif"
}

define void @"main"()
{
entry:
  %"arr" = alloca <5 x i32>
  %"arr.1" = load <5 x i32>, <5 x i32>* %"arr"
  %"arr.2" = insertelement <5 x i32> %"arr.1", i32 5, i32 0
  store <5 x i32> %"arr.2", <5 x i32>* %"arr"
  %"arr.3" = load <5 x i32>, <5 x i32>* %"arr"
  %"arr.4" = insertelement <5 x i32> %"arr.3", i32 3, i32 1
  store <5 x i32> %"arr.4", <5 x i32>* %"arr"
  %"arr.5" = load <5 x i32>, <5 x i32>* %"arr"
  %"arr.6" = insertelement <5 x i32> %"arr.5", i32 4, i32 2
  store <5 x i32> %"arr.6", <5 x i32>* %"arr"
  %"arr.7" = load <5 x i32>, <5 x i32>* %"arr"
  %"arr.8" = insertelement <5 x i32> %"arr.7", i32 1, i32 3
  store <5 x i32> %"arr.8", <5 x i32>* %"arr"
  %"arr.9" = load <5 x i32>, <5 x i32>* %"arr"
  %"arr.10" = insertelement <5 x i32> %"arr.9", i32 2, i32 4
  store <5 x i32> %"arr.10", <5 x i32>* %"arr"
  %"arr.11" = load <5 x i32>, <5 x i32>* %"arr"
  %".7" = call <5 x i32> @"bubble_sort"(<5 x i32> %"arr.11")
  %"sorted_arr" = alloca <5 x i32>
  store <5 x i32> %".7", <5 x i32>* %"sorted_arr"
  %"i" = alloca i32
  store i32 0, i32* %"i"
  %"i.1" = load i32, i32* %"i"
  %".10" = icmp slt i32 %"i.1", 5
  br i1 %".10", label %"entry.if", label %"entry.endif"
```

```
117 for_loop:
118   %"i.2" = load i32, i32* %"i"
119   %"sorted_arr.1" = load <5 x i32>, <5 x i32>* %"sorted_arr"
120   %"i.3" = load i32, i32* %"i"
121   %"sorted_arr.1.1" = extractelement <5 x i32> %"sorted_arr.1", i32 %"i.3"
122   %".13" = bitcast [9 x i8]* @"function_main_VoidType_31791" to i8*
123   %".14" = call i32 (i8*, ...) @"printf"(i8* %".13", i32 %"i.2", i32 %"
      sorted_arr.1.1")
124   %"i.4" = load i32, i32* %"i"
125   %".15" = add i32 %"i.4", 1
126   store i32 %".15", i32* %"i"
127   %"i.5" = load i32, i32* %"i"
128   %".17" = icmp slt i32 %"i.5", 5
129   br i1 %".17", label %"for_loop.if", label %"for_loop.endif"
130 entry.if:
131   br label %"for_loop"
132 entry.endif:
133   ret void
134 for_loop.if:
135   br label %"for_loop"
136 for_loop.endif:
137   br label %"entry.endif"
138 }
139
140 @"function_main_VoidType_31791" = internal constant [9 x i8] c"%d -> %d\0a"
```

ЗАКЛЮЧЕНИЕ

Рассмотрены основные фазы функционирования приложения, выполняющего компиляцию кода языка Golang в байт-код для LLVM.

Приведен обзор основных алгоритмов лексического и синтаксического анализа. Рассмотрены стандартные средства построения синтаксических анализаторов.

Реализована программа, составляющая синтаксическое дерево, получающая промежуточный код LLVM IR представления виртуальный машины LLVM путем, который в дальнейшем можно использовать для создания исполняемых

файлов.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

Список литературы

[1] Серебряков В.А., Галочкин М.П. Основы конструирования компиляторов. — М. : Едиториал УРСС, 1999. — С. 193.

[2] Ахо А., Сети Р., Ульман Д. Компиляторы. Принципы, технологии, инструменты. — М. : Вильямс, 2001.

[3] Parr Terence. Definitive ANTLR4 reference. — Pragmatic Bookshelf, 2013. — P. 305.

[4] Lopes Bruno Cardoso. Getting Started with LLVM Core Libraries. — Packt Publishing, 2014. — P. 295.

ПРИЛОЖЕНИЕ А

Листинг 5 – Лексер

```
lexer grammar GoLexer;

// Keywords

BREAK                   : 'break' -> mode(NLSEMI);
DEFAULT                 : 'default';
FUNC                    : 'func';
CASE                    : 'case';
MAP                     : 'map';
ELSE                    : 'else';
PACKAGE                 : 'package';
CONST                   : 'const';
IF                      : 'if';
RANGE                   : 'range';
TYPE                    : 'type';
CONTINUE                : 'continue' -> mode(NLSEMI);
FOR                     : 'for';
RETURN                  : 'return' -> mode(NLSEMI);
VAR                     : 'var';

NIL_LIT                 : 'nil' -> mode(NLSEMI);

IDENTIFIER              : LETTER (LETTER | UNICODE_DIGIT)* -> mode(NLSEMI);

// Punctuation

L_PAREN                 : '(';
R_PAREN                 : ')' -> mode(NLSEMI);
L_CURLY                 : '{';
R_CURLY                 : '}' -> mode(NLSEMI);
L_BRACKET               : '[';
R_BRACKET               : ']' -> mode(NLSEMI);
ASSIGN                  : '=';
COMMA                   : ',';
SEMI                    : ';';
COLON                   : ':';
```

```
37 PLUS_PLUS              : '++' -> mode(NLSEMI);
38 MINUS_MINUS            : '--' -> mode(NLSEMI);
39 DECLARE_ASSIGN         : ':=';
40 ELLIPSIS               : '...';
41
42 // Logical
43
44 LOGICAL_OR             : '||';
45 LOGICAL_AND            : '&&';
46
47 // Relation operators
48
49 EQUALS                 : '==';
50 NOT_EQUALS             : '!=';
51 LESS                   : '<';
52 LESS_OR_EQUALS         : '<=';
53 GREATER                : '>';
54 GREATER_OR_EQUALS      : '>=';
55
56 // Arithmetic operators
57
58 OR                     : '|';
59 DIV                    : '/';
60 MOD                    : '%';
61 LSHIFT                 : '<<';
62 RSHIFT                 : '>>';
63 BIT_CLEAR              : '&^';
64
65 // Unary operators
66
67 EXCLAMATION            : '!';
68
69 // Mixed operators
70
71 PLUS                   : '+';
72 MINUS                  : '-';
73 CARET                  : '^';
74 STAR                   : '*';
75 AMPERSAND              : '&';
76
```

```
77 // Number literals
78
79 DECIMAL_LIT            : ('0' | [1-9] ('_'? [0-9])*) -> mode(NLSEMI);
80 BINARY_LIT             : '0' [bB] ('_'? BIN_DIGIT)+ -> mode(NLSEMI);
81 OCTAL_LIT              : '0' [oO]? ('_'? OCTAL_DIGIT)+ -> mode(NLSEMI);
82 HEX_LIT                : '0' [xX]  ('_'? HEX_DIGIT)+ -> mode(NLSEMI);
83
84
85 FLOAT_LIT : (DECIMAL_FLOAT_LIT | HEX_FLOAT_LIT) -> mode(NLSEMI);
86
87 DECIMAL_FLOAT_LIT      : DECIMALS ('.' DECIMALS? EXPONENT? | EXPONENT)
88                        | '.' DECIMALS EXPONENT?
89                        ;
90
91 HEX_FLOAT_LIT          : '0' [xX] HEX_MANTISSA HEX_EXPONENT
92                        ;
93
94 fragment HEX_MANTISSA  : ('_'? HEX_DIGIT)+ ('.' ( '_'? HEX_DIGIT )*)?
95                        | '.' HEX_DIGIT ('_'? HEX_DIGIT)*;
96
97 fragment HEX_EXPONENT  : [pP] [+-]? DECIMALS;
98
99
100 IMAGINARY_LIT          : (DECIMAL_LIT | BINARY_LIT |  OCTAL_LIT | HEX_LIT |
      FLOAT_LIT) 'i' -> mode(NLSEMI);
101
102 // Rune literals
103
104 fragment RUNE               : '\'' (UNICODE_VALUE | BYTE_VALUE) '\'';//: '\''
       (~[\n\\] | ESCAPED_VALUE) '\'';
105
106 RUNE_LIT                : RUNE -> mode(NLSEMI);
107
108
109
110 BYTE_VALUE : OCTAL_BYTE_VALUE | HEX_BYTE_VALUE;
111
112 OCTAL_BYTE_VALUE: '\\' OCTAL_DIGIT OCTAL_DIGIT OCTAL_DIGIT;
113
114 HEX_BYTE_VALUE: '\\' 'x'  HEX_DIGIT HEX_DIGIT;
```

```
115
116 LITTLE_U_VALUE: '\\' 'u' HEX_DIGIT HEX_DIGIT HEX_DIGIT HEX_DIGIT;
117
118 BIG_U_VALUE: '\\' 'U' HEX_DIGIT HEX_DIGIT HEX_DIGIT HEX_DIGIT HEX_DIGIT
      HEX_DIGIT HEX_DIGIT HEX_DIGIT;
119
120 // String literals
121
122 RAW_STRING_LIT         : '`' ~'`'*                        '`' -> mode(NLSEMI);
123 INTERPRETED_STRING_LIT : '"' (~["\\] | ESCAPED_VALUE)*  '"' -> mode(NLSEMI);
124
125 // Hidden tokens
126
127 WS                     : [ \t]+             -> channel(HIDDEN);
128 COMMENT                : '/*' .*? '*/'      -> channel(HIDDEN);
129 TERMINATOR             : [\r\n]+            -> channel(HIDDEN);
130 LINE_COMMENT           : '//' ~[\r\n]*      -> channel(HIDDEN);
131
132 fragment UNICODE_VALUE: ~[\r\n'] | LITTLE_U_VALUE | BIG_U_VALUE |
      ESCAPED_VALUE;
133
134 // Fragments
135
136 fragment ESCAPED_VALUE
137     : '\\' ('u' HEX_DIGIT HEX_DIGIT HEX_DIGIT HEX_DIGIT
138             | 'U' HEX_DIGIT HEX_DIGIT HEX_DIGIT HEX_DIGIT HEX_DIGIT HEX_DIGIT
      HEX_DIGIT HEX_DIGIT
139             | [abfnrtv\\'"]
140             | OCTAL_DIGIT OCTAL_DIGIT OCTAL_DIGIT
141             | 'x' HEX_DIGIT HEX_DIGIT)
142     ;
143
144 fragment DECIMALS
145     : [0-9] ('_'? [0-9])*
146     ;
147
148 fragment OCTAL_DIGIT
149     : [0-7]
150     ;
151
```

```
152 fragment HEX_DIGIT
153     : [0-9a-fA-F]
154     ;
155
156 fragment BIN_DIGIT
157     : [01]
158     ;
159
160 fragment EXPONENT
161     : [eE] [+-]? DECIMALS
162     ;
163
164 fragment LETTER
165     : UNICODE_LETTER
166     | '_'
167     ;
168
169 mode NLSEMI;
170
171
172 // Treat whitespace as normal
173 WS_NLSEMI                     : [ \t]+             -> channel(HIDDEN);
174 // Ignore any comments that only span one line
175 COMMENT_NLSEMI                : '/*' ~[\r\n]*? '*/'      -> channel(HIDDEN);
176 LINE_COMMENT_NLSEMI : '//' ~[\r\n]*      -> channel(HIDDEN);
177 // Emit an EOS token for any newlines, semicolon, multiline comments or the
      EOF and
178 //return to normal lexing
179 EOS:              ([\r\n]+ | ';' | '/*' .*? '*/' | EOF)            -> mode(
      DEFAULT_MODE);
180 // Did not find an EOS, so go back to normal lexing
181 OTHER: -> mode(DEFAULT_MODE), channel(HIDDEN);
```

ПРИЛОЖЕНИЕ Б

Листинг 6 – Парсер

```
parser grammar GoParser;

options {
    tokenVocab = GoLexer;
    superClass = GoParserBase;
}

sourceFile:
    packageClause eos (
        (functionDecl | declaration) eos
    )* EOF;

packageClause: PACKAGE packageName = IDENTIFIER;

declaration: constDecl | typeDecl | varDecl;

constDecl: CONST (constSpec | L_PAREN (constSpec eos)* R_PAREN);

constSpec: identifierList (type_? ASSIGN expressionList)?;

identifierList: IDENTIFIER (COMMA IDENTIFIER)*;

expressionList: expression (COMMA expression)*;

typeDecl: TYPE (typeSpec | L_PAREN (typeSpec eos)* R_PAREN);

typeSpec: IDENTIFIER ASSIGN? type_;

// Function declarations

functionDecl: FUNC IDENTIFIER (signature block?);

varDecl: VAR (varSpec | L_PAREN (varSpec eos)* R_PAREN);

varSpec:
    identifierList (
```

```
37          type_ (ASSIGN expressionList)?
38          | ASSIGN expressionList
39      );
40
41  block: L_CURLY statementList? R_CURLY;
42
43  statementList: ((SEMI? | EOS? | {closingBracket()}?) statement eos)+;
44
45  statement:
46      declaration
47      | labeledStmt
48      | simpleStmt
49      | returnStmt
50      | breakStmt
51      | continueStmt
52      | block
53      | ifStmt
54      | forStmt;
55
56  simpleStmt:
57      incDecStmt
58      | assignment
59      | expressionStmt
60      | shortVarDecl;
61
62  expressionStmt: expression;
63
64  incDecStmt: expression (PLUS_PLUS | MINUS_MINUS);
65
66  assignment: expressionList assign_op expressionList;
67
68  assign_op: (
69          PLUS
70          | MINUS
71          | OR
72          | CARET
73          | STAR
74          | DIV
75          | MOD
76          | LSHIFT
```

```
77          | RSHIFT
78          | AMPERSAND
79          | BIT_CLEAR
80      )? ASSIGN;
81
82  shortVarDecl: identifierList DECLARE_ASSIGN expressionList;
83
84  emptyStmt: EOS | SEMI;
85
86  labeledStmt: IDENTIFIER COLON statement?;
87
88  returnStmt: RETURN expressionList?;
89
90  breakStmt: BREAK IDENTIFIER?;
91
92  continueStmt: CONTINUE IDENTIFIER?;
93
94
95  ifStmt:
96      IF ( expression
97              | eos expression
98              | simpleStmt eos expression
99              ) block (
100         ELSE (ifStmt | block)
101     )?;
102
103 typeList: (type_ | NIL_LIT) (COMMA (type_ | NIL_LIT))*;
104
105 forStmt: FOR (expression? | forClause | rangeClause?) block;
106
107 forClause:
108     initStmt = simpleStmt? eos expression? eos postStmt = simpleStmt?;
109
110 rangeClause: (
111         expressionList ASSIGN
112         | identifierList DECLARE_ASSIGN
113     )? RANGE expression;
114
115 type_: typeName | typeLit | L_PAREN type_ R_PAREN;
116
```

```
117 typeName: IDENTIFIER;
118
119 typeLit:
120     arrayType
121     | pointerType
122     | functionType
123     | sliceType
124     | mapType;
125
126 arrayType: L_BRACKET arrayLength R_BRACKET elementType;
127
128 arrayLength: expression;
129
130 elementType: type_;
131
132 pointerType: STAR type_;
133
134 sliceType: L_BRACKET R_BRACKET elementType;
135
136 // It's possible to replace `type` with more restricted typeLit list and also
        pay attention to nil maps
137 mapType: MAP L_BRACKET type_ R_BRACKET elementType;
138
139 methodSpec:
140     IDENTIFIER parameters result
141     | IDENTIFIER parameters;
142
143 functionType: FUNC signature;
144
145 signature:
146     parameters result
147     | parameters;
148
149 result: parameters | type_;
150
151 parameters:
152     L_PAREN (parameterDecl (COMMA parameterDecl)* COMMA?)? R_PAREN;
153
154 parameterDecl: identifierList? ELLIPSIS? type_;
155
```

```
156 expression:
157     primaryExpr
158     | unary_op = (
159         PLUS
160         | MINUS
161         | EXCLAMATION
162         | CARET
163         | STAR
164         | AMPERSAND
165     ) expression
166     | expression mul_op = (
167         STAR
168         | DIV
169         | MOD
170         | LSHIFT
171         | RSHIFT
172         | AMPERSAND
173         | BIT_CLEAR
174     ) expression
175     | expression add_op = (PLUS | MINUS | OR | CARET) expression
176     | expression rel_op = (
177         EQUALS
178         | NOT_EQUALS
179         | LESS
180         | LESS_OR_EQUALS
181         | GREATER
182         | GREATER_OR_EQUALS
183     ) expression
184     | expression LOGICAL_AND expression
185     | expression LOGICAL_OR expression;
186
187 primaryExpr:
188     operand
189     | conversion
190     | primaryExpr (
191         index
192         | slice_
193         | arguments
194     );
195
```

```
196
197 conversion: nonNamedType L_PAREN expression COMMA? R_PAREN;
198
199 nonNamedType: typeLit | L_PAREN nonNamedType R_PAREN;
200
201 operand: literal | operandName | L_PAREN expression R_PAREN;
202
203 literal: basicLit | compositeLit | functionLit;
204
205 basicLit:
206     NIL_LIT
207     | integer
208     | string_
209     | FLOAT_LIT;
210
211 integer:
212     DECIMAL_LIT
213     | BINARY_LIT
214     | OCTAL_LIT
215     | HEX_LIT
216     | IMAGINARY_LIT
217     | RUNE_LIT;
218
219 operandName: IDENTIFIER;
220
221 compositeLit: literalType literalValue;
222
223 literalType:
224     arrayType
225     | L_BRACKET ELLIPSIS R_BRACKET elementType
226     | sliceType
227     | mapType
228     | typeName;
229
230 literalValue: L_CURLY (elementList COMMA?)? R_CURLY;
231
232 elementList: keyedElement (COMMA keyedElement)*;
233
234 keyedElement: (key COLON)? element;
235
```

```
key: expression | literalValue;

element: expression | literalValue;

string_: RAW_STRING_LIT | INTERPRETED_STRING_LIT;

embeddedField: STAR? typeName;

functionLit: FUNC signature block; // function

index: L_BRACKET expression R_BRACKET;

slice_:
    L_BRACKET (
        expression? COLON expression?
        | expression? COLON expression COLON expression
    ) R_BRACKET;


arguments:
    L_PAREN (
        (expressionList | nonNamedType (COMMA expressionList)?) ELLIPSIS?
    COMMA?
    )? R_PAREN;

//receiverType: typeName | '(' ('*' typeName | receiverType) ')';

receiverType: type_;

eos:
    SEMI
    | EOF
    | EOS
    | {closingBracket()}?
    ;
```