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 подмножества языка программирования Golang”

Студент группы **ИУ7-22М** _____ **Р.Д. Третьяк**
(Подпись, дата) (И.О. Фамилия)

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

***2022** 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)

«_____» _____ 20____ г.

ЗАДАНИЕ
на выполнение курсового проекта

по дисциплине Конструирование компиляторов

Студент группы ИУ7-22М

Третьяк Роман Дмитриевич
(Фамилия, имя, отчество)

Тема курсового проекта Компилятор подмножества языка программирования Golang

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

Учебный

Источник тематики (кафедра, предприятие, НИР) кафедра

Задание: Разработать компилятор подмножества языка программирования Golang, используя генератор парсеров ANTLR4 и инструмент кодогенерации LLVM.

Оформление курсового проекта:

Расчетно-пояснительная записка на 20-30 листах формата А4.

Расчетно-пояснительная записка должна содержать постановку, введение, полное описание всех стадий компиляции заключение, список использованной литературы

Перечень графического (иллюстративного) материала (чертежи, плакаты, слайды и т.п.)

Дата выдачи задания «_____»_____20____ г.

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

Студент _____ **Р. Д. Третьяк**
(Подпись, дата) (И. О. Фамилия)

Примечание: Задание оформляется в двух экземплярах: один выдается студенту, второй хранится на кафедре.

Оглавление

РЕФЕРАТ

Расчетно-пояснительная записка 46 с., 2 рис., 4 источник., 3 прил.

Цель работы – разработать компилятор подмножества язка программирования Golang для платформы LLVM.

Задачи работы:

- проанализировать способы и алгоритмы построения компиляторов;
- выбрать инструменты для разработки компилятора;
- спроктировать компилятор;
- реализовать компилятор на языке Python.

В первой части работы проанализированы алгоритмы и инструменты, используемые при разработке компиляторов, описаны особенности языка Golang и платформы LLVM. Во второй части описаны способы реализации составляющих компилятора. В третьей части описаны сборка и использование разработанного компилятора.

Введение

Подавляющая часть программных продуктов в текущих реалиях разрабатваются с применением языков высокого уровня. Они позволяют программисту, оперируя абстракциями языка, сконцентрироваться на логике создаваемой программы, абстрагируясь от низкоуровневых особенностей и рутинных задач. Однако, программа на высокуровневом языке программирования не может в чистом виде быть выполненной компьютером, требуется перевод в код, воспринимаемый платфорой-исполнителем. Эту задачу решает компилятор.

Компилятор – комплекс программных средств, принимающий на вход программный код, написанный на высокоуровневом языке программаирования. Компилятор производит анализ полученного исходного кода на предмет ошибок, в некоторых случаях производит оптимизацию программного кода, и генерирует эквивалентный код на целевом языке. Целевым языком чаще всего является платфомозависимый ассемблерный код или код, воспринимаемый некоторой виртуальной машиной.

Целью работы является разработка компилятора подмножества языка Golang для виртуальной машины LLVM.

1 Аналитический раздел

В разделе приводится описание обобщенной архитектуры компилятора, особенности языка Golang и платформы LLVM.

1.1 Архитектура комплятора

Компилятор производит перевод кода программы, написанного на одном языке, в эквивалетный код на другом языке. Этот процесс условно делится на две части: анализ и кодогенерация [2].

Анализ состоит из трех фаз: лексической, синтаксической и семантической.

1.1.1 Лексический анализ

На этапе лексического анализа входная последовательность символов, пердставляющих из себя текс на некотором языке программирования, преобразуется в поток лексем (токенов). На этом этапе возможно обнаружения лексических ошибок, таких как наличие неподдерживаемых символов, или, например, некорректная запись идентификатора или числовой константы. Полученным токенам могут назначаться типы, сохраняться информация об их местоположении в исходном коде [2].

Лексический анализ может быть представлен и как самостоятельная фаза анализа, 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].

Правила разбиения текста программы на токены могут быть заданы с помощью грамматики языка или регулярных выражений.

1.1.2 Синтаксический анализ

Синтаксический анализ, или разбор, как его еще называют, – это процесс сопоставления последовательности токенов исходного языка с его формальной грамматикой. Данная фаза анализа выполняется при помощи программы-парсера, принимающей на вход поток лексем, в случае, когда лексический анализ выполняется отдельно, или исходный код программы в ином случае. Результатом работы парсера является дерево разбора (или абстрактное синтаксическое дерево), узлами дерева являются оперции, а дочерние узлы являются аргументами операции.

Синтаксический анализатор фиксирует синтаксические ошибки, т.е. ошибки, связанные с нарушением правил построения текста на языке программирования, задаваемых формальной грамматикой [**?**].

Существующие алгоритмы синтаксического анализа можно объеденить в две категории: алгоритмы восходящего и нисходящего анализа.

Нисходящий анализ применим к текстам, построенным по правилом контекстн
свободной LL грамматикам. LL означает, что разбор происходит слева направо и строится левый вывод.

Примером алгоритма нисходящего анализа является рекурсивный спуск. Суть алгоритм заключается в рекурсивном применении правл граматики к входной последовательности лексем с их последовательным ”поглощением”. Для корректной работы алгоритма требуется отсутсвие так называемой левой рекурсии в грамматике.

При восходящем анализе дерево строится от листьев к корню. Сначала распознаются правила грамматики, содержащие в правой части терминалы, затем те, которые содержат в правой части левые части предыдущих правил и так далее. Таким образом на каждом шаге свертки некоторая подстрока, соответствующая правой части продукции, замещается левым символом данной продукции. Примером алгоритма этого типа служит таблично управляемый ана-

лизатор операторного предшествия. Этот алгоритм использует операции ”перенос/свертка применим для LR грамматик.

1.1.3 Семантический анализ

На данном этапе анализируется построенное парсером дерево выражений с целью интерпретации распознанных конструкций и их связей между собой, например, связь переменной с её типом данных, значением, областью видимости.

Семантический анализ позволяет перестроить дерево разбора с учетом смысла конструкций, а также распознать такие ошибки, как:

- несоответствие типов;
- необъявленные переменные;
- множественные объявления переменной;
- отсутствие переменной в области видимости;
- несоответствие параметров.

В результате семантического анализа дерево приводится к виду, пригодному для генерации кода **??**.

1.1.4 Кодогенерация

В ходе этого этапа компиляции происходит обход полученного в результате анализа дерева. Для каждого узла, предствляющего из себя операцию, создается одна или несколько инструкций на целевом языке. На этапе генерации кода полагается, что входное дерево не содержит ошибок.

В ходе генерации или после неё может также проводится оптимизация полученного набора инструкций с помощью программы-оптимизатора.

1.1.5 Стандартные средства построения анализаторов

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 про-

грамма 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 [3].

Все рассмотренные средства включают в себя лексический анализ на основе входной грамматики.

1.2 Язык Golang

Golang это высокоуровневый компилируемый язык программирования, ориентированный на написание многопоточных приложений. В качестве особенностей этого языка можно выделить:

1) концепция использования ”зеленых потоков”;
2) строгая грамматика;
3) наличие небольшого рантайма, включающего только планеровщик и сборщик мусора;
4) компилируемость;
5) отсутсвие синтаксического сахара;
6) строгая типизация.

.

Для реализации было выбрано подмножество языка Golang, содержащее базовые 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;

.

Грамматика в формате ANTLR представлена в приложении А.

1.3 LLVM

LLVM – проект программной инфраструктуры для создания компиляторов. В основе инфраструктуры лежит платформонезависимая система кодирования машинных инструкций – байткод LLVM IR (Intermediate Representation).

LLVM IR в отличие от ассемблера является платформонезависимым, а его инструкции перегруженны для множества типов данных. LLVM IR может выполняться как самой виртуальной машиной LLVM, так и быть транслирован в байткод для множества платформ, включая ARM, x86, x86-64, GPU от AMD и Nvidia и другие.

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.

Кроме того имеется возможность создания множества связанных модулей.

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.

В данной работе реализуется генерация кода LLVM IR с помощью binding'а для языка Python: llvmlite [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

Для создания анализатора в работе используется ANTLR4 [3]. В качестве входных данных для него 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

Помимо парсера ANTLR также генерирует класс, реализующий обход полученного дерева, для каждого типа узлов дерева в классе присутсвует метод его обработки, называние метода начинается со слова ”visit”. В листинге 1 представлен пример метода обработки узла операции объявления функции. В методе происходит генерация инструкции выделения памяти, а также создание записи в таблице переменных текущей области видимости. Кроме того производится попытка привидения типа значения к типу переменной, в случае, если помимо объявления переменной происходит также присвоение ей значения.

Листинг 1 – Метод обработки объявления/определения переменной

```
def visitVarSpec(self, ctx: GoParser.VarSpecContext):
    variable_name = ctx.identifierList().getText()
    variable_type = self.visitType_(ctx.type_())
    variable = self.current_builder.alloca(variable_type, name=variable_name)
    self.variables_scopes[self.current_scope_name][variable_name] = variable
    if ctx.expressionList():
        value = self.visitExpressionList(ctx.expressionList())
        value = self.maybe_convert_type(value, variable.type.pointee)
        self.current_builder.store(value, variable)
    return variable
```

Отедльного внимания заслуживают вспомогательные методы согласования типов. Так как LLVMIR требует строго соответсвия типов аргументов и операции, зачастую необходимо перед выполнением инструкции операции выполнить приведение типов, загрузку значения из памяти. Для этих целей служат вспомогательные функции, одна из которых приведена в листинге **??**

Листинг 2 – Вспомогательный метод согласования типов

```
def maybe_convert_type(self, value, target_type):
    if isinstance(value, ir.Constant):
        value.type = target_type
        return value
    if isinstance(value, ir.AllocaInstr):
        value = self.current_builder.load(value, name=value.name)
    if value.type == target_type:
        return value
    if target_type.intrinsic_name == 'f64':
        value = self.current_builder.sitofp(value, ir.DoubleType())
        return value
    if target_type.intrinsic_name == 'i1':
        value = self.current_builder.icmp_signed('>', value, value.type(0))
        return value
    if target_type.intrinsic_name in CONVERTABLE_INT_EXT[value.type.
    intrinsic_name]:
        value = self.current_builder.sext(value, target_type)
        return value
    if target_type.intrinsic_name in CONVERTABLE_INT_TRUNC[value.type.
    intrinsic_name]:
        value = self.current_builder.trunc(value, target_type)
        return value
    raise Exception('Filed to convert type to target')
```

Так как методы разбора узлов вызывают соответствующие методы разбора дочерних узлов, то обработка всего дерева заключается в разборе корневого элемента вызовом метода ”visitSourceFile”. Когда все узлы дерева обработаны, полученаня последовательность инструкций на языке LLVM IR верифицируется и записывается в файл.

Программный код класса для кодогенерации представлен в приложении Б.

3 Технологический раздел

3.1 Сборка компилятора

Компилятор написан на языке Python версии 3.10, поэтому может быть запущен на любой операционной системе, имеющей компилятор этой версии языка Python.

Перед запуском компилятора необходимо установить необходимые зависимости: скачать и собрать LLVM, установить необходимые библиотеки командой:

Листинг 3 – Установка зависимостей

```
1     pip3 install -r requirements.txt
```

3.2 Запуск компилятора

Для запуска компилятора необходимо выполнить команду

Листинг 4 – Установка зависимостей

```
1     python main.py main.go
```

В результате работы компилятора будет создан файл на языке LLVM IR output.ll, который необходимо скомпилировать для текущей платформы:

Листинг 5 – Установка зависимостей

```
1     llc output.ll -fileType=obj -o output.o && gcc output.o
```

3.3 Пример использования

В качестве примера использования рассматривается код на языке Golang, который выводит в стандартный поток вывода числа от 0 до 10.

Листинг 6 – Исходный код на языке Golang

```
1 package main
2
3 func main() {
4     for i := 0; i < 10; i++ {
```

```
5           printf("%d\n", i)
6       }
7   }
```

Абстрактное синтаксическое дерево для данной программы представлено на рисунке 1.

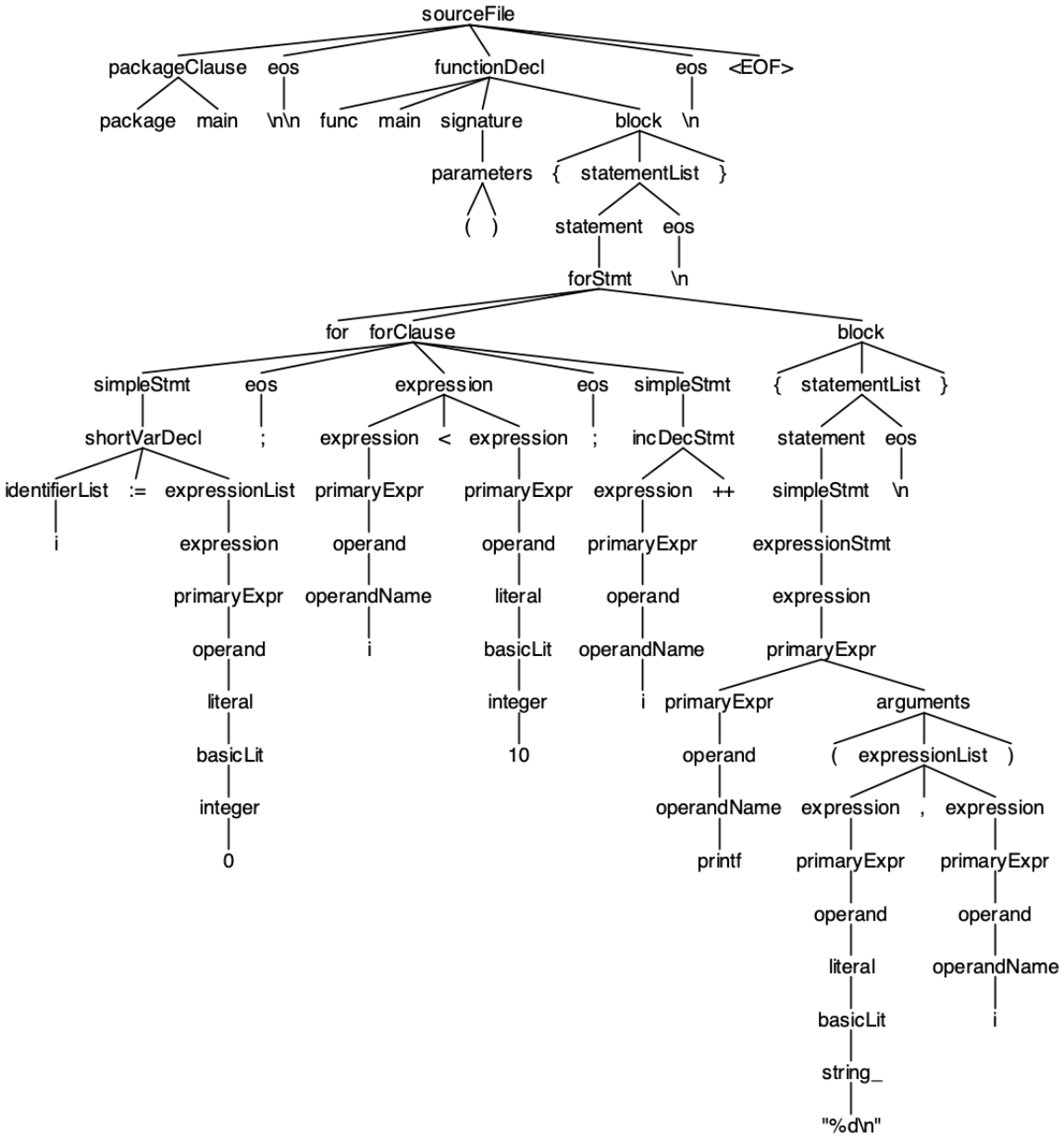

Рисунок 1 – AST программы печати чисел от 0 до 10

Результатом работы компилятора является код на языке промежуточного

представляения LLVM IR, он представлен в приложении В.

При запуске программы, в стандартный поток вывода печатаются числа от 0 до 10, рисунок 2

```
→  print_0_to_10 git:(master) ✗ llc -filetype=obj output.ll && gcc output.o -o output
→  print_0_to_10 git:(master) ✗ ./output
0
1
2
3
4
5
6
7
8
9
```

Рисунок 2 – Вывод программы, печатающей символы от 0 до 10

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.

Были сгенерированы лексический и синтаксический анализаторы по грамматике языка с помощью инструмента ANTLR.

Реализована программа, составляющая синтаксическое дерево. После построения дерева по нему генерируется код на языке промежуточного представления LLVM IR с использованием пакета llvmlite для языка Python. В дальнейшем этот код используется для создания исполняемого файла для целевой платформы.

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 7 – 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()}?
    ;
```

Приложение Б

Листинг 8 – Код программы, выводящей числа от 0 до 10 на языке LLVMIR

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

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

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

define void @"main"()
{
entry:
  %"i" = alloca i32
  store i32 0, i32* %"i"
  %"i.1" = load i32, i32* %"i"
  %".3" = icmp slt i32 %"i.1", 10
  br i1 %".3", label %"entry.if", label %"entry.endif"
for_loop:
  %"i.2" = load i32, i32* %"i"
  %".6" = bitcast [3 x i8]* @"function_main_VoidType_35417" to i8*
  %".7" = call i32 (i8*, ...) @"printf"(i8* %".6", i32 %"i.2")
  %"i.3" = load i32, i32* %"i"
  %".8" = add i32 %"i.3", 1
  store i32 %".8", i32* %"i"
  %"i.4" = load i32, i32* %"i"
  %".10" = icmp slt i32 %"i.4", 10
  br i1 %".10", label %"for_loop.if", label %"for_loop.endif"
entry.if:
  br label %"for_loop"
entry.endif:
  ret void
for_loop.if:
  br label %"for_loop"
for_loop.endif:
  br label %"entry.endif"
}

```

```
37 @"function_main_VoidType_35417" = internal constant [3 x i8] c"%d\0a"
```

Приложение В

Листинг 9 – Код генератора LLVM IR

```
import random
from collections import defaultdict
from typing import Callable

from pygoco.antlr4_base import GoParser
from pygoco.antlr4_base import GoParserVisitor
from llvmlite import ir

from pygoco.core import type_defs
from pygoco.core.type_defs import CONVERTABLE_INT_EXT
from pygoco.core.type_defs import CONVERTABLE_INT_TRUNC


class PygocoVisitor(GoParserVisitor):
    def __init__(self):
        self.module: ir.Module = None
        self.current_function: ir.Function = None
        self.current_block: ir.Block = None
        self.current_builder: ir.IRBuilder = None
        self.current_scope_name: str = None
        self.variables_scopes = defaultdict(dict)
        self.printf: Callable = None
        self.scanf: Callable = None

    def try_convert_to_one_type(self, lhs, rhs):
        rhs = self.get_variable_value(rhs)
        lhs = self.get_variable_value(lhs)
        if lhs.type.intrinsic_name == rhs.type.intrinsic_name:
            return lhs, rhs
        if lhs.type.intrinsic_name == 'i1' or rhs.type.intrinsic_name == 'i1'
    :
            if lhs.type.intrinsic_name == 'i1':
                rhs = self.current_builder.icmp_signed('>', rhs, rhs.type(0))
            else:
                lhs = self.current_builder.icmp_signed('>', lhs, lhs.type(0))
            return lhs, rhs
```

```
36          if lhs.type.intrinsic_name == 'f64' or rhs.type.intrinsic_name == '
    f64':
37              if lhs.type.intrinsic_name == 'f64':
38                  rhs = self.current_builder.sitofp(rhs, ir.DoubleType())
39              else:
40                  lhs = self.current_builder.sitofp(lhs, ir.DoubleType())
41              return lhs, rhs
42          if lhs.type.intrinsic_name in CONVERTABLE_INT_EXT[rhs.type.
    intrinsic_name]:
43              rhs = self.current_builder.sext(rhs, lhs.type)
44              return lhs, rhs
45          if rhs.type.intrinsic_name in CONVERTABLE_INT_EXT[lhs.type.
    intrinsic_name]:
46              lhs = self.current_builder.sext(lhs, rhs.type)
47              return lhs, rhs
48          if lhs.type.intrinsic_name in CONVERTABLE_INT_TRUNC[rhs.type.
    intrinsic_name]:
49              rhs = self.current_builder.trunc(rhs, lhs.type)
50              return lhs, rhs
51          if rhs.type.intrinsic_name in CONVERTABLE_INT_TRUNC[lhs.type.
    intrinsic_name]:
52              lhs = self.current_builder.trunc(lhs, rhs.type)
53              return lhs, rhs
54          raise Exception('Filed to convert types')
55
56      def maybe_convert_type(self, value, target_type):
57          if isinstance(value, ir.Constant):
58              value.type = target_type
59              return value
60          if isinstance(value, ir.AllocaInstr):
61              value = self.current_builder.load(value, name=value.name)
62          if value.type == target_type:
63              return value
64          if target_type.intrinsic_name == 'f64':
65              value = self.current_builder.sitofp(value, ir.DoubleType())
66              return value
67          if target_type.intrinsic_name == 'i1':
68              value = self.current_builder.icmp_signed('>', value, value.type
    (0))
69              return value
```

```
70          if target_type.intrinsic_name in CONVERTABLE_INT_EXT[value.type.
    intrinsic_name]:
71              value = self.current_builder.sext(value, target_type)
72              return value
73          if target_type.intrinsic_name in CONVERTABLE_INT_TRUNC[value.type.
    intrinsic_name]:
74              value = self.current_builder.trunc(value, target_type)
75              return value
76          raise Exception('Filed to convert type to target')
77
78      def declare_printf_function(self):
79          function_type = ir.FunctionType(ir.IntType(32), [ir.IntType(8).
    as_pointer()], var_arg=True)
80          printf_function = ir.Function(self.module, function_type, name='
    printf')
81
82          def printf(fmt: str, *args, builder: ir.IRBuilder) -> ir.Function:
83              fmt_text = fmt
84              fmt = ir.Constant(ir.ArrayType(ir.IntType(8), len(fmt_text)),
    bytearray(fmt_text.encode()))
85              fmt_name = f'{self.current_scope_name}_{str(hash(random.random())
    )[:5]}'
86              fmt_arg = ir.GlobalVariable(self.module, fmt.type, name=fmt_name)
87              fmt_arg.linkage = 'internal'
88              fmt_arg.global_constant = True
89              fmt_arg.initializer = fmt
90              fmt_arg = builder.bitcast(fmt_arg, ir.IntType(8).as_pointer())
91              res = builder.call(printf_function, [fmt_arg, *args])
92              return res
93
94          self.printf = printf
95
96      def declare_scanf_function(self):
97          function_type = ir.FunctionType(ir.IntType(32), [ir.IntType(8).
    as_pointer()], var_arg=True)
98          scanf_function = ir.Function(self.module, function_type, name='scanf'
    )
99
100         def scanf(fmt: str, *args, builder: ir.IRBuilder) -> ir.Function:
101             fmt_text = fmt
```

```
102             fmt = ir.Constant(ir.ArrayType(ir.IntType(8), len(fmt_text)),
    bytearray(fmt_text.encode()))
103             fmt_name = f'{self.current_scope_name}_{str(hash(random.random())
    )[:5]}'
104             fmt_arg = ir.GlobalVariable(self.module, fmt.type, name=fmt_name)
105             fmt_arg.linkage = 'internal'
106             fmt_arg.global_constant = True
107             fmt_arg.initializer = fmt
108             fmt_arg = builder.bitcast(fmt_arg, ir.IntType(8).as_pointer())
109             res = builder.call(scanf_function, [fmt_arg, *args])
110             return res
111
112         self.scanf = scanf
113
114     def visitSourceFile(self, ctx: GoParser.SourceFileContext):
115         self.visitChildren(ctx)
116         return self.module
117
118     def visitPackageClause(self, ctx: GoParser.PackageClauseContext):
119         package_name = ctx.IDENTIFIER().getText()
120         self.module = ir.Module(name=package_name)
121         self.declare_printf_function()
122         self.declare_scanf_function()
123         self.visitChildren(ctx)
124
125     def visitDeclaration(self, ctx: GoParser.DeclarationContext):
126         return self.visitChildren(ctx)
127
128     def visitConstDecl(self, ctx: GoParser.ConstDeclContext):
129         return self.visitChildren(ctx)
130
131     def visitConstSpec(self, ctx: GoParser.ConstSpecContext):
132         return self.visitChildren(ctx)
133
134     def visitIdentifierList(self, ctx: GoParser.IdentifierListContext):
135         return [identifier.getText() for identifier in ctx.children]
136
137     def visitExpressionList(self, ctx: GoParser.ExpressionListContext):
138         if len(ctx.children) == 1:
139             return self.visitChildren(ctx)
```

```
140         values = []
141         expressions = ctx.getChildren(lambda c: isinstance(c, GoParser.
    ExpressionContext))
142         for expression in expressions:
143             values.append(self.visitExpression(expression))
144         return values
145 
146     def visitTypeDecl(self, ctx: GoParser.TypeDeclContext):
147         return self.visitChildren(ctx)
148 
149     def visitTypeSpec(self, ctx: GoParser.TypeSpecContext):
150         return self.visitChildren(ctx)
151 
152     def visitFunctionDecl(self, ctx: GoParser.FunctionDeclContext):
153         name = ctx.IDENTIFIER().getText()
154         return_type = ir.VoidType()
155         parameters = self.visitParameters(ctx.signature().parameters())
156         parameters_names = [param[0] for param in parameters]
157         parameters_types = [param[1] for param in parameters]
158         if ctx.signature().result():
159             return_type = self.visitType_(ctx.signature().result().type_())
160 
161         self.current_function = ir.Function(self.module, ir.FunctionType(
    return_type, parameters_types), name=name)
162         self.current_block = self.current_function.append_basic_block(name='
    entry')
163         self.current_builder = ir.IRBuilder(self.current_block)
164         self.current_scope_name = f'function_{name}_{return_type.__class__.
    __name__}'
165 
166         args = self.current_function.args
167         for arg_name, arg in zip(parameters_names, args):
168             arg.name = arg_name
169             self.variables_scopes[self.current_scope_name][arg_name] = self.
    replace_argument_with_variable(arg)
170 
171         result = self.visitBlock(ctx.block())
172         if result is None:
173             if isinstance(return_type, ir.VoidType):
174                 self.current_builder.ret_void()
```

```
175             elif not self.current_builder.block.is_terminated:
176                 self.current_builder.unreachable()
177         return self.current_function
178 
179     def visitVarDecl(self, ctx: GoParser.VarDeclContext):
180         return self.visitChildren(ctx)
181 
182     def visitVarSpec(self, ctx: GoParser.VarSpecContext):
183         variable_name = ctx.identifierList().getText()
184         variable_type = self.visitType_(ctx.type_())
185         variable = self.current_builder.alloca(variable_type, name=
    variable_name)
186         self.variables_scopes[self.current_scope_name][variable_name] =
    variable
187         if ctx.expressionList():
188             value = self.visitExpressionList(ctx.expressionList())
189             value = self.maybe_convert_type(value, variable.type.pointee)
190             self.current_builder.store(value, variable)
191         return variable
192 
193     def visitBlock(self, ctx: GoParser.BlockContext):
194         return self.visitChildren(ctx)
195 
196     def visitStatementList(self, ctx: GoParser.StatementListContext):
197         return self.visitChildren(ctx)
198 
199     def visitStatement(self, ctx: GoParser.StatementContext):
200         return self.visitChildren(ctx)
201 
202     def visitSimpleStmt(self, ctx: GoParser.SimpleStmtContext):
203         return self.visitChildren(ctx)
204 
205     def visitExpressionStmt(self, ctx: GoParser.ExpressionStmtContext):
206         return self.visitChildren(ctx)
207 
208     def visitIncDecStmt(self, ctx: GoParser.IncDecStmtContext):
209         lhs = self.visitExpression(ctx.expression())
210         value = self.get_variable_value(lhs)
211         if ctx.PLUS_PLUS():
212             add = self.current_builder.add(value, value.type(1))
```

```
213             self.current_builder.store(add, lhs)
214             self.replace_variable(lhs)
215             res = lhs
216         else:
217             sub = self.current_builder.sub(value, value.type(1))
218             self.current_builder.store(sub, lhs)
219             self.replace_variable(lhs)
220             res = lhs
221         return res
222 
223     def replace_variable(self, variable):
224         name = variable.name.split('.')[0]
225         self.variables_scopes[self.current_scope_name][name] = variable
226 
227     def get_variable_value(self, variable):
228         if isinstance(variable, tuple):
229             vector = self.get_variable_value(variable[0])
230             index = self.get_variable_value(variable[1])
231             element = self.current_builder.extract_element(vector, index,
    name=vector.name)
232             return element
233         if isinstance(variable, ir.AllocaInstr):
234             return self.current_builder.load(variable, name=variable.name)
235         return variable
236 
237     def replace_argument_with_variable(self, argument: ir.Argument) -> ir.
    AllocaInstr:
238         variable = self.current_builder.alloca(argument.type, name=argument.
    name)
239         self.current_builder.store(argument, variable)
240         self.replace_variable(variable)
241         return variable
242 
243     def add(self, lhs, rhs):
244         if lhs.type.intrinsic_name == 'f64':
245             return self.current_builder.fadd(lhs, rhs)
246         return self.current_builder.add(lhs, rhs)
247 
248     def sub(self, lhs, rhs):
249         if lhs.type.intrinsic_name == 'f64':
```

```
250             return self.current_builder.fsub(lhs, rhs)
251         return self.current_builder.sub(lhs, rhs)
252 
253     def mul(self, lhs, rhs):
254         if lhs.type.intrinsic_name == 'f64':
255             return self.current_builder.fmul(lhs, rhs)
256         return self.current_builder.mul(lhs, rhs)
257 
258     def div(self, lhs, rhs):
259         if lhs.type.intrinsic_name == 'f64':
260             return self.current_builder.fdiv(lhs, rhs)
261         return self.current_builder.sdiv(lhs, rhs)
262 
263     def assign_to_variable(self, lhs, ctx: GoParser.AssignmentContext):
264         variable = lhs
265         if isinstance(lhs, ir.Argument):
266             variable = self.replace_argument_with_variable(lhs)
267         operation = ctx.assign_op().getText()
268         rhs = self.visitExpressionList(ctx.expressionList(1))
269         rhs = self.get_variable_value(rhs)
270         rhs = self.maybe_convert_type(rhs, variable.type.pointee)
271         if operation == '=':
272             self.current_builder.store(rhs, variable)
273             self.replace_variable(variable)
274             res = variable
275         elif operation == '+=':
276             add = self.add(self.get_variable_value(variable), rhs)
277             self.current_builder.store(add, variable)
278             self.replace_variable(variable)
279             res = variable
280         elif operation == '-=':
281             sub = self.sub(self.get_variable_value(variable), rhs)
282             self.current_builder.store(sub, variable)
283             self.replace_variable(variable)
284             res = variable
285         elif operation == '*=':
286             mul = self.mul(self.get_variable_value(variable), rhs)
287             self.current_builder.store(mul, variable)
288             self.replace_variable(variable)
289             res = variable
```

```
290         elif operation == '/=':
291             div = self.div(self.get_variable_value(variable), rhs)
292             self.current_builder.store(div, variable)
293             self.replace_variable(variable)
294             res = variable
295         else:
296             raise Exception(f'Unknown assigment operation {operation}')
297         return res
298 
299     def assign_to_array_element(self, vector, index, ctx: GoParser.
    AssignmentContext):
300         index = self.get_variable_value(index)
301         if isinstance(vector, ir.Argument):
302             vector = self.replace_argument_with_variable(vector)
303         operation = ctx.assign_op().getText()
304         rhs = self.visitExpressionList(ctx.expressionList(1))
305         rhs = self.get_variable_value(rhs)
306         rhs = self.maybe_convert_type(rhs, vector.type.pointee.element)
307         if operation == '=':
308             assign = self.current_builder.insert_element(self.
    get_variable_value(vector), rhs, index, name=vector.name)
309             self.current_builder.store(assign, vector)
310             self.replace_variable(vector)
311             return vector
312         element = self.get_variable_value((vector, index))
313         if operation == '+=':
314             add = self.add(self.get_variable_value(element), rhs)
315             assign = self.current_builder.insert_element(self.
    get_variable_value(vector), add, index, name=vector.name)
316             self.current_builder.store(assign, vector)
317             self.replace_variable(vector)
318         elif operation == '-=':
319             sub = self.sub(self.get_variable_value(element), rhs)
320             assign = self.current_builder.insert_element(self.
    get_variable_value(vector), sub, index, name=vector.name)
321             self.current_builder.store(assign, vector)
322             self.replace_variable(vector)
323         elif operation == '*=':
324             mul = self.mul(self.get_variable_value(element), rhs)
325             assign = self.current_builder.insert_element(self.
```

```
get_variable_value(vector), mul, index, name=vector.name)
326             self.current_builder.store(assign, vector)
327             self.replace_variable(vector)
328         elif operation == '/=':
329             div = self.div(self.get_variable_value(element), rhs)
330             assign = self.current_builder.insert_element(self.
get_variable_value(vector), div, index, name=vector.name)
331             self.current_builder.store(assign, vector)
332             self.replace_variable(vector)
333         else:
334             raise Exception(f'Unknown assigment operation {operation}')
335         return vector
336 
337     def visitAssignment(self, ctx: GoParser.AssignmentContext):
338         lhs = self.visitExpressionList(ctx.expressionList(0))
339         if isinstance(lhs, tuple):
340             self.assign_to_array_element(lhs[0], lhs[1], ctx)
341         else:
342             self.assign_to_variable(lhs, ctx)
343 
344     def visitAssign_op(self, ctx: GoParser.Assign_opContext):
345         return self.visitChildren(ctx)
346 
347     def visitShortVarDecl(self, ctx: GoParser.ShortVarDeclContext):
348         variable_name = ctx.identifierList().getText()
349         variable_value = self.get_variable_value(self.visitExpressionList(ctx
.expressionList()))
350         variable = self.current_builder.alloca(variable_value.type, name=
variable_name)
351         self.current_builder.store(variable_value, variable)
352         self.variables_scopes[self.current_scope_name][variable_name] =
variable
353         return variable
354 
355     def visitEmptyStmt(self, ctx: GoParser.EmptyStmtContext):
356         return self.visitChildren(ctx)
357 
358     def visitLabeledStmt(self, ctx: GoParser.LabeledStmtContext):
359         return self.visitChildren(ctx)
360 
```

```
361     def visitReturnStmt(self, ctx: GoParser.ReturnStmtContext):
362         result = self.visitExpressionList(ctx.expressionList())
363         result = self.maybe_convert_type(result, self.current_function.
    return_value.type)
364         return self.current_builder.ret(result)
365 
366     def visitBreakStmt(self, ctx: GoParser.BreakStmtContext):
367         return self.visitChildren(ctx)
368 
369     def visitContinueStmt(self, ctx: GoParser.ContinueStmtContext):
370         return self.visitChildren(ctx)
371 
372     def visitIfStmt(self, ctx: GoParser.IfStmtContext):
373         predicate = self.visitExpression(ctx.expression())
374         if len(ctx.children) == 5:
375             with self.current_builder.if_else(predicate) as (then, otherwise)
    :
376                 with then:
377                     self.visitBlock(ctx.block(0))
378                 with otherwise:
379                     self.visitBlock(ctx.block(1))
380         else:
381             with self.current_builder.if_then(predicate):
382                 self.visitBlock(ctx.block(0))
383 
384     def visitTypeList(self, ctx: GoParser.TypeListContext):
385         return self.visitChildren(ctx)
386 
387     def pop_variable(self, variable):
388         var_name = variable.name.split('.')[0]
389         self.variables_scopes[self.current_scope_name].pop(var_name)
390 
391     def visitForStmt(self, ctx: GoParser.ForStmtContext):
392         idx, pred_ctx, iter_ctx = self.visitForClause(ctx.forClause())
393         for_loop_block = self.current_builder.append_basic_block('for_loop')
394         pred1 = self.visitExpression(pred_ctx)
395         with self.current_builder.if_then(pred1) as then:
396             self.current_builder.branch(for_loop_block)
397         with self.current_builder.goto_block(for_loop_block):
398             self.visitBlock(ctx.block())
```

```
399             self.visitSimpleStmt(iter_ctx)
400             pred = self.visitExpression(pred_ctx)
401             with self.current_builder.if_then(pred):
402                 self.current_builder.branch(for_loop_block)
403             self.current_builder.branch(then)
404         self.pop_variable(idx)
405 
406     def visitForClause(self, ctx: GoParser.ForClauseContext):
407         idx = self.visitSimpleStmt(ctx.simpleStmt(0))
408         pred_expr = ctx.expression()
409         iter_stmt = ctx.simpleStmt(1)
410         return idx, pred_expr, iter_stmt
411 
412     def visitRangeClause(self, ctx: GoParser.RangeClauseContext):
413         return self.visitChildren(ctx)
414 
415     def visitType_(self, ctx: GoParser.Type_Context):
416         if ctx.typeName():
417             type_name = ctx.typeName().IDENTIFIER().getText()
418             if type_name in type_defs.INT_TYPES:
419                 return type_defs.INT_TYPES[type_name]
420             if type_name in type_defs.BOOL_TYPES:
421                 return type_defs.BOOL_TYPES[type_name]
422             if type_name in type_defs.FLOAT_TYPES:
423                 return type_defs.FLOAT_TYPES[type_name]
424         elif ctx.typeLit():
425             return self.visitTypeLit(ctx.typeLit())
426         raise Exception('Unknown type')
427 
428     def visitTypeName(self, ctx: GoParser.TypeNameContext):
429         return self.visitChildren(ctx)
430 
431     def visitTypeLit(self, ctx: GoParser.TypeLitContext):
432         return self.visitChildren(ctx)
433 
434     def visitArrayType(self, ctx: GoParser.ArrayTypeContext):
435         length = self.visitArrayLength(ctx.arrayLength()).constant
436         element_type = self.visitElementType(ctx.elementType())
437         return ir.VectorType(element_type, length)
438 
```

```
439     def visitArrayLength(self, ctx: GoParser.ArrayLengthContext):
440         return self.visitChildren(ctx)
441 
442     def visitElementType(self, ctx: GoParser.ElementTypeContext):
443         return self.visitChildren(ctx)
444 
445     def visitPointerType(self, ctx: GoParser.PointerTypeContext):
446         return self.visitChildren(ctx)
447 
448     def visitSliceType(self, ctx: GoParser.SliceTypeContext):
449         return self.visitChildren(ctx)
450 
451     def visitMapType(self, ctx: GoParser.MapTypeContext):
452         return self.visitChildren(ctx)
453 
454     def visitMethodSpec(self, ctx: GoParser.MethodSpecContext):
455         return self.visitChildren(ctx)
456 
457     def visitFunctionType(self, ctx: GoParser.FunctionTypeContext):
458         return self.visitChildren(ctx)
459 
460     def visitSignature(self, ctx: GoParser.SignatureContext):
461         return self.visitChildren(ctx)
462 
463     def visitResult(self, ctx: GoParser.ResultContext):
464         return self.visitChildren(ctx)
465 
466     def visitParameters(self, ctx: GoParser.ParametersContext):
467         if len(ctx.children) == 0:
468             return []
469         parameters = []
470         for param_decl in ctx.getChildren(lambda c: isinstance(c, GoParser.
    ParameterDeclContext)):
471             params = self.visitParameterDecl(param_decl)
472             parameters.extend(params)
473         return parameters
474 
475     def visitParameterDecl(self, ctx: GoParser.ParameterDeclContext):
476         identifiers = self.visitIdentifierList(ctx.identifierList())
477         type_ = self.visitType_(ctx.type_())
```

```
478         parameters = [(identifier, type_) for identifier in identifiers]
479         return parameters
480 
481     def visitExpression(self, ctx: GoParser.ExpressionContext):
482         if ctx.add_op:
483             lhs: ir.Constant = self.visitExpression(ctx.expression(0))
484             rhs: ir.Constant = self.visitExpression(ctx.expression(1))
485             lhs, rhs = self.try_convert_to_one_type(lhs, rhs)
486             if ctx.add_op.text == '+':
487                 return self.add(lhs, rhs)
488             elif ctx.add_op.text == '-':
489                 return self.sub(lhs, rhs)
490             elif ctx.add_op.text == '|':
491                 return self.current_builder.or_(lhs, rhs)
492             elif ctx.add_op.text == '^':
493                 return self.current_builder.xor(lhs, rhs)
494         elif ctx.mul_op:
495             lhs: ir.Constant = self.visitExpression(ctx.expression(0))
496             rhs: ir.Constant = self.visitExpression(ctx.expression(1))
497             lhs, rhs = self.try_convert_to_one_type(lhs, rhs)
498             if ctx.mul_op.text == '*':
499                 return self.mul(lhs, rhs)
500             elif ctx.mul_op.text == '/':
501                 return self.div(lhs, rhs)
502             elif ctx.mul_op.text == '%':
503                 return self.current_builder.srem(lhs, rhs)
504             elif ctx.mul_op.text == '<<':
505                 return self.current_builder.shl(lhs, rhs)
506             elif ctx.mul_op.text == '>>':
507                 return self.current_builder.ashr(lhs, rhs)
508             elif ctx.mul_op.text == '&':
509                 return self.current_builder.and_(lhs, rhs)
510             elif ctx.mul_op.text == '&^':
511                 return self.current_builder.not_(self.current_builder.and_(
    lhs, rhs))
512         elif ctx.rel_op:
513             lhs: ir.Constant = self.visitExpression(ctx.expression(0))
514             rhs: ir.Constant = self.visitExpression(ctx.expression(1))
515             lhs, rhs = self.try_convert_to_one_type(lhs, rhs)
516             res = self.current_builder.icmp_signed(ctx.rel_op.text, lhs, rhs)
```

```
517             return res
518
519         return self.visitChildren(ctx)
520
521     def visitPrimaryExpr(self, ctx: GoParser.PrimaryExprContext):
522         if not ctx.primaryExpr():
523             value = self.visitChildren(ctx)
524             return value
525         left = self.visitPrimaryExpr(ctx.primaryExpr())
526         if ctx.index():
527             vector = left
528             index = self.visitIndex(ctx.index())
529             return vector, index
530         arguments = self.visitArguments(ctx.arguments()) or []
531         if not isinstance(arguments, list):
532             arguments = [arguments]
533         if left == 'printf':
534             retyped_arguments = []
535             for stock_arg in arguments:
536                 retyped_arguments.append(self.get_variable_value(stock_arg))
537             return self.printf(*retyped_arguments, builder=self.
    current_builder)
538         elif left == 'scanf':
539             return self.scanf(*arguments, builder=self.current_builder)
540         elif left:
541             function = self.module.globals.get(left)
542             retyped_arguments = []
543             for stock_arg, arg in zip(arguments, function.args):
544                 retyped_arguments.append(self.maybe_convert_type(stock_arg,
    arg.type))
545             function_call = self.current_builder.call(function,
    retyped_arguments)
546             return function_call
547
548     def visitConversion(self, ctx: GoParser.ConversionContext):
549         return self.visitChildren(ctx)
550
551     def visitNonNamedType(self, ctx: GoParser.NonNamedTypeContext):
552         return self.visitChildren(ctx)
553
```

```
554     def visitOperand(self, ctx: GoParser.OperandContext):
555         if ctx.L_PAREN():
556             return self.visitExpression(ctx.expression())
557         return self.visitChildren(ctx)
558 
559     def visitLiteral(self, ctx: GoParser.LiteralContext):
560         value = self.visitChildren(ctx)
561         return value
562 
563     def visitBasicLit(self, ctx: GoParser.BasicLitContext):
564         if ctx.integer():
565             return self.visitInteger(ctx.integer())
566         elif ctx.string_():
567             return self.visitString_(ctx.string_())
568         number = float(ctx.getText())
569         return ir.DoubleType()(number)
570 
571     def visitInteger(self, ctx: GoParser.IntegerContext):
572         number = int(ctx.getText())
573         return ir.IntType(32)(number)
574 
575     def visitOperandName(self, ctx: GoParser.OperandNameContext):
576         module_globals = list(self.module.globals.keys())
577         name = ctx.getText()
578         if name == 'true':
579             return ir.IntType(1)(1)
580         if name == 'false':
581             return ir.IntType(1)(0)
582         if name in module_globals:
583             return name
584         if name in self.variables_scopes[self.current_scope_name]:
585             return self.variables_scopes[self.current_scope_name][name]
586         raise Exception(f'Unknown operandName "{name}"')
587 
588     def visitCompositeLit(self, ctx: GoParser.CompositeLitContext):
589         return self.visitChildren(ctx)
590 
591     def visitLiteralType(self, ctx: GoParser.LiteralTypeContext):
592         return self.visitChildren(ctx)
593 
```

```
594     def visitLiteralValue(self, ctx: GoParser.LiteralValueContext):
595         return self.visitChildren(ctx)
596 
597     def visitElementList(self, ctx: GoParser.ElementListContext):
598         return self.visitChildren(ctx)
599 
600     def visitKeyedElement(self, ctx: GoParser.KeyedElementContext):
601         return self.visitChildren(ctx)
602 
603     def visitKey(self, ctx: GoParser.KeyContext):
604         return self.visitChildren(ctx)
605 
606     def visitElement(self, ctx: GoParser.ElementContext):
607         return self.visitChildren(ctx)
608 
609     def visitString_(self, ctx: GoParser.String_Context):
610         string = ctx.getText().replace('"', '').replace('\\n', '\n')
611         return string
612 
613     def visitEmbeddedField(self, ctx: GoParser.EmbeddedFieldContext):
614         return self.visitChildren(ctx)
615 
616     def visitFunctionLit(self, ctx: GoParser.FunctionLitContext):
617         return self.visitChildren(ctx)
618 
619     def visitIndex(self, ctx: GoParser.IndexContext):
620         return self.visitExpression(ctx.expression())
621 
622     def visitSlice_(self, ctx: GoParser.Slice_Context):
623         return self.visitChildren(ctx)
624 
625     def visitArguments(self, ctx: GoParser.ArgumentsContext):
626         if ctx.expressionList():
627             value = self.visitExpressionList(ctx.expressionList())
628             return value
629         return self.visitChildren(ctx)
630 
631     def visitReceiverType(self, ctx: GoParser.ReceiverTypeContext):
632         return self.visitChildren(ctx)
633 
```

```
634     def visitEos(self, ctx: GoParser.EosContext):
635         return self.visitChildren(ctx)
```