Министерство образования и науки Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«Московский государственный технический университет
имени Н.Э. Баумана»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»
КАФЕДРА «Теоретическая информатика и компьютерные технологии»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА К КУРСОВОМУ ПРОЕКТУ НА ТЕМУ:

Компилятор объектно-ориентированного статически типизированного языка программирования для LLVM

Студент ИУ9-72Б Зворыгин А.В. _____
(ф.и.о.) (подпись, дата)

Руководитель курсового проекта Коновалов А.В. _____
(ф.и.о.) (подпись, дата)

Москва, 2021г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Сегодня существует достаточно много высокоуровневых языков программирования. Каждый из них создавался с какой-то определенной целью. Некоторые из них приносят определенные новшества и улучшения, кардинально не меняя базовый язык, например Kotlin и C#. Другие занимают нишу определенной специализации, например такие как Haskell и 1C, остальные же часто представляют собой академические проекты, например Scala. Создание языка программирования всегда будет актуальным направлением разработки, поскольку при проектировании каждого нового языка программисты привносят новые подходы, идеи и принципы, которые могут быть полезны не только в конструировании компиляторов.

Целью данной работы будет являться изучение этапов конструирования компиляторов на примере создания собственного объектно-ориентированного статически типизированного языка программирования для LLVM.

1 Исследование

В данном разделе будут рассмотрены особенности нового языка программирования, его парадигмы, свойства и синтаксис. Прежде чем перейти к исследованию необходимо дать определения некоторых терминов использующихся в конструировании компиляторов и теории формальных языков, а также описать основные фазы компиляции.

1.1 Основные термины

Алфавит — множество атомарных (неделимых) символов [1] [2].

Слово (также — *цепочка*) — произвольная последовательность символов из данного алфавита [2].

Формальный язык –множество конечных слов над конечным алфавитом [2].

Формальная грамматика или просто *грамматика* в теории формальных языков — способ описания формального языка, то есть выделения некоторого подмножества из множества всех слов некоторого конечного алфавита [1].

Компилятор — это программа, осуществляющая перевод программ из исходного языка S в целевой язык T [3].

Интерпретатор — это программа, осуществляющая выполнение программ, написанных на языке L [3].

1.2 Фазы компиляции

Компиляция обычно проходит в несколько основных фаз [4] [5] [6].

Лексический анализ — это фаза компиляции, объединяющая последовательно идущие во входном потоке символы в слова, называемые лексемами исходного языка [3].

Синтаксический анализ — это фаза компиляции, группирующая лексемы, порождаемые на фазе лексического анализа, в синтаксические структуры [3].

Семантический анализ — это фаза компиляции, выполняющая проверку синтаксического дерева на соответствие его компонентов контекстным ограничениям [3] [2].

Генерация промежуточного представления — это фаза компиляции, выполняющая перевод синтаксического дерева в форму, удобную для последующей оптимизации и генерации кода [3].

Генерация кода — это фаза компиляции, выполняющая перевод программы из промежуточного представления в целевой язык [3].

Постобработка — это фаза компиляции, связанная с оптимизацией объектной программы, полученной в результате генерации кода [3].

1.3 Парадигмы и основные свойства языка программирования

Далее необходимо описать основные свойства будущего языка программирования:

1. Статически сильно(строго) типизированный
2. Компилируемый и не интерпретируемый
3. Управление памятью — автоматическое, с помощью сборщика мусора
4. Все элементы языка — объект
5. Отсутствие синтаксического сахара
6. Пространство имен — с помощью пакетов (модулей)
7. Поддержка ООП — с помощью классов

Главной парадигмой языка является простота. Полагается, что, чем язык проще, тем легче его использовать. Простота будет достигаться на различных этапах использования языка. В дальнейшем будет использоваться название языка **lang**.

1.4 Система типов языка программирования

Неотъемлемой и, несомненно, базовой частью языка является система типов [7]. В языке lang на данный момент существует 7 основных типов, диапазоны значений представлены для 32 и более разрядных платформ:

1. byte, char [−127; +127]

2. short [−32767, +32767]

3. int [−2 147 483 647, +2 147 483 647]

4. long [−9 223 372 036 854 775 807, +9 223 372 036 854 775 807]

5. float, соответствует «IEEE 754 бинарный формат с плавающей запятой одинарной точности» [4]

6. double, соответствует «IEEE 754 бинарный формат с плавающей запятой двойной точности» [4]

7. Ссылка — указатель на объект в адресном пространстве, размер эквивалентен long

1.5 Синтаксические конструкции

В этом разделе будут описаны синтаксические конструкции языка lang.

1.5.1 Программа

Как и большинстве языков программирования [4] для написания программы требуется объявить функцию **main —** точку входа в программу (см. Листинг 1). Вложенность синтаксических конструкций достигается за счет необходимой табуляции.

Листинг 1 — Минимальная программа на языке lang:

```
1.   () int main ->
2.       // Комментарии обозначаются двойным символом /
3.       return 0
```

1.5.2 Объявления и определения

Основной особенностью объявлений языка lang является их зафиксированный синтаксис. Сначала обозначается тип переменной, а затем ее название (см. Листинг 2).

Листинг 2 — Объявление переменных:

```
1.   () int main ->
2.       int a
3.       double d
4.       boolean b
5.       return 0
```

Не определенные переменные принимают значения по умолчанию.

1. Для типа boolean — false
2. Для типа ссылка — null
3. Для типов float и double — 0.0
4. Для типов short, int, long — 0

Определение переменных этих типов производится с помощью знака равно следующим образом (см. Листинг 3)

Листинг 3 — Определение переменных

```
1.   () int main ->
2.       int a = 0
3.       double d = 0.0
4.       boolean b = false
5.       return 0
```

1.5.3 Арифметические и логические операции

В языке существует стандартный набор арифметических и логических операций. К арифметическим операциям относятся операции

1. сложения — a + b
2. вычитания — a - b
3. умножения — a * b
4. деления — a / b
5. постфиксного и префиксного декремента и инкремента –a++, --b
6. остатка от деления — a % b

К логическим операциям относятся операции

1. Дополнения –! a

2. Равенства — a == b

3. Логическое или — a || b

4. Логическое и — a && b

5. Операторы сравнения –<=, >=, <, >

Примеры применения операций можно увидеть далее (см. Листинг 4)

Листинг 4 — Арифметические и логические операции

```
1.   () int main ->
2.       int num = 1
3.       num = 1 + 1
4.       num = num — 1
5.       num = num * 4
6.       num = num / 2
7.
8.       boolean result = num != 0 && num % 3 == 1 || num == 1
9.
10.      return 0
```

1.5.4 Управляющие конструкции условных переходов

В языке существуют стандартные управляющие конструкции условных переходов. Для их использования требуется применение ключевых слов — **if**, **elif**, **else** (см Листинг 5). Результирующим значением выражения, которое применяется в таких конструкциях, должно быть значение типа boolean.

Листинг 5 –Конструкции условных переходов

```
1.   () int main ->
2.       int num = 1
3.
4.       if num == 0
5.           …
6.       elif num == 1
7.           …
8.       elif num == 2
9.           …
10.      else
11.          …
12.      return 0
```

Также существует тернарная операция if-else (см. Листинг 6)

Листинг 6 –Тернарная операция if-else

```
1.   () int main ->
2.       int num = 1
3.
4.       int var = num == 1 ? 0 : 2
5.
6.       return 0
```

1.5.5 Управляющие конструкции циклов

В языке на данный момент существует конструкция цикла вида **while** (см. Листинг 7).

Листинг 7 — Цикл while

```
1.   () int main ->
2.       int num = 10
3.
4.       while num != 0
5.           num--
6.
7.       return 0
```

Для циклов предусмотрены дополнительные условные управляющие конструкции **break** и **continue** (см. Листинг 8 и Листинг 9)

Листинг 8 — Пример break

```
1.   () int main ->
2.       while true
3.           break
4.
5.       return 0
```

Листинг 9 — Пример continue

```
1.   () int main ->
2.       while true
3.           continue
4.
5.       return 0
```

1.5.6 Функции

Определение функций выполняется по правилу:

$$(type1\ name1, type2\ name2, \dots)\ type\ function\_name$$

В круглых скобках указываются типы и имена входных параметров, после указывается тип возвращаемого значения и имя функции. Таким образом тип функции можно определить как набор типов входных параметров и тип возвращаемого значения, т.е.

$$(type1\ name1, type2\ name2, \dots)\ type = function\_type$$

Объявление функции сводится к записи

$$function\_type\ function\_name$$

Пример объявления функции можно увидеть далее (см. Листинг 10).

Листинг 10 –Объявления функции

```
1.   (int a, boolean b, short s) double foo
```

Для определения функции необходимо использовать выражение вида

$$function\_type\ function\_name\ \rightarrow\ body$$

Пример определения и вызова функции можно увидеть далее (см. Листинг 11).

Листинг 11 — Определение и вызов функции

```
1.   (double a, boolean b, short s) double foo ->
2.       double c = a
3.       c = c * 0.5
4.       return c < 0.5 ? c : foo(a, b, c)
```

Выход из функции обеспечивается ключевым словом **return**

1.5.7 Массивы

В языке поддерживаются многомерные массивы, которые заполняются значениями по умолчанию. Пример использования массива можно увидеть далее (см. Листинг 12). Для создания массива используется ключевое слово **new**.

Листинг 12 — Объявление и определение массива

```
1.   () int main ->
2.       // объявление и определение двухмерного массива
3.       int [][] array = new int[50][20]
4.
5.       int size = 10
6.       int [][][] array2
7.
8.       // создание трехмерного массива
9.       array2 = new int[size][size][size]
10.
11.      array2[1][2][3] = 4
12.      int var = array2[1][2][3]
13.      array2[1][2] = null
14.
15.      return 0
```

Массивы создаются рекурсивно. В данном случае создания трехмерного массива будет создано 1 + 10 + 100 = 111 массивов.

1.5.8 Классы

Для поддержки ООП в языке существуют классы. Класс можно описать с помощью полей и методов, которые он содержит. Пример определения класса языка lang (см. Листинг 13).

Листинг 13 — Определение класса на языке lang:

```
class Point
    int x             // определение полей
    int y

    (int x, int y) -> // определение конструктора
        this.x = x
        this.y = y

    () int getX ->    // определения методов
        return this.x

    (int x) void setX ->
        this.x = x
```

Для создания объекта используется ключевое слово **new** (см. Листинг 14)

Листинг 14 — Создание и использование объекта

```
() int main ->

    Point p = new Point(1, 2)  // создание объекта

    p.x = 2                    // обращения к полю объекта
    p.setX(3)                  // вызовы метода
    p.y = p.getX()

    return 0
```

Инкапсуляция в языке не поддерживается. Также на данный момент не поддерживается наследование.

1.5.9 Элементы стандартной библиотеки

На данный момент, в стандартной библиотеке языка реализована поддержка строковых литералов (см. Листинг 15). Функции putchar и puts являются интерфейсами для вызова функций стандартной библиотеки языка C (stdlib.h) [6].

Листинг 15 — Использование строковых литералов

```
1.  (char[] s) int puts
2.
3.  (int n) int putchar
4.
5.  (String s) void println ->
6.      if s == null
7.          println("null")
8.          return
9.
10.     int i = 0
11.     int l = s.length
12.
13.     while i < l
14.         // <int> -операция приведения типа
15.         int c = <int>s.bytes[i]
16.         putchar(c)
17.         i++
18.     putchar(10)
19.
20. class String
21.     char[] bytes
22.     int length
23.
24.     (char [] bytes, int length) ->
25.         this.bytes = bytes
26.         this.length = length
27.
28. () int main ->
29.     println("Hello world!")
30.     return 0
```

1.6 Лексика языка

Формальное описание лексики дано в приложении А — Лексика языка.

1.7 Грамматика языка

Формальное описание грамматики дано в приложении Б — Грамматика языка.

2 Проектирование

В данном разделе необходимо определить общую архитектуру компилятора и межкомпонентные интерфейсы. Исходный код курсовой работы располагается в репозитории сервиса Github [8].

2.1 Общая архитектура компилятора

Архитектура представляет собой последовательную цепочку модулей (см. Рисунок 1), которые выполняют поэтапную трансформацию программы для перевода её из языка lang в язык LLVM (этап трансляции программы из кода LLVM в объектный код и трансляции из объектного кода в исполняемый файл выполняются программами LLVM и Clang соответственно).

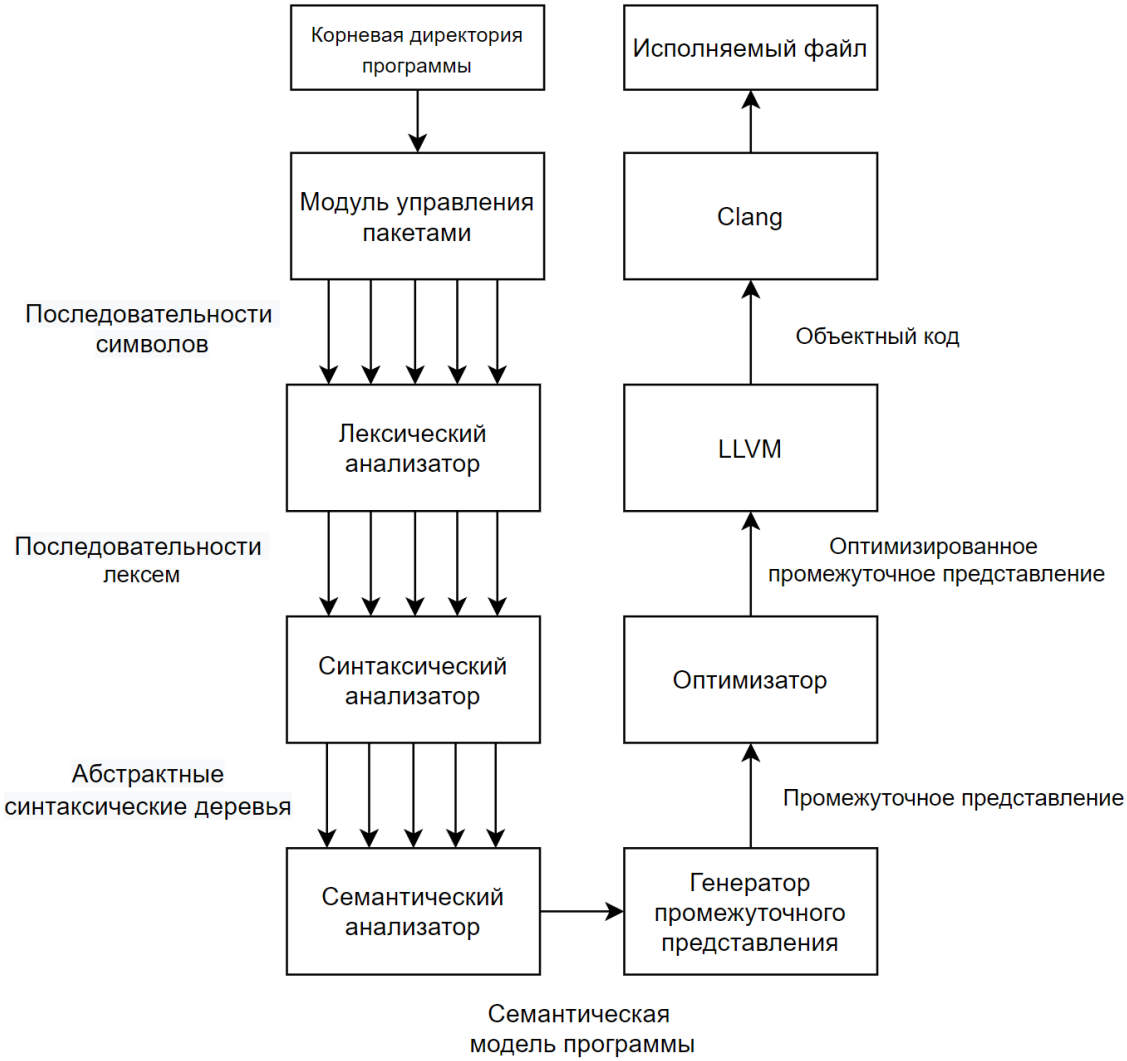

Рисунок 1 — Архитектура компилятора

Модуль управления пакетами предназначен для получения доступа ко всем файлам программы для проведения лексического анализа, а в последствии на этапе семантического анализа определяет пространство имен для каждого из исходных файлов.

2.2 Проектирование лексического анализатора

Для проектирования лексического анализатора необходимо определить его интерфейсы и зависимости от других модулей. Задачей лексического анализатора является объединение идущих в одном потоке символов в лексемы языка.

Таким образом можно схематично спроектировать интерфейс лексического анализатора — функция, принимающая на вход поток символов и возвращающая поток лексем. Лексемы, которые не определены явно в приложении А — Лексика языка, приводят к лексической ошибке программы, за исключением пробельных символов, которые не влияют на поведение программы.

2.3 Проектирование синтаксического анализатора

Задачей синтаксического анализатора является построение абстрактного синтаксического дерева на основе потока лексем. Таким образом, интерфейсом синтаксического анализатора является функция, принимающая на вход поток лексем и возвращающая корень абстрактного синтаксического дерева.

Поток лексем, не соответствующий приложению Б — Грамматика языка приводит к синтаксической ошибке.

2.4 Проектирование семантического анализатора

Задачей семантического анализатора является агрегация абстрактных синтаксических деревьев в единую модель программы. Также семантический анализатор должен выполнить:

1. проверки на принадлежность имен пространству имен

2. связать вызовы функций с их объявлениями

3. связать объявления функций с их определениями

4. определить корректность использования типов

5. определить неявные преобразования типов

6. подготовить классы и функции для последующей генерации промежуточного представления — определить корректность использования методов и конструкторов.

Таким образом, интерфейс семантического анализатора представляет собой функцию, принимающую набор синтаксических деревьев и модуль управления пакетами. Функция возвращает семантическую модель программы, на базе которой будет строится промежуточное представление.

2.5 Проектирование генератора промежуточного представления

Генератор промежуточного представления должен обработать семантическую модель программы и сгенерировать промежуточное представление в виде абстрактной модели (не обязательно ориентированной на LLVM). Для этого необходимо создать набор логических сущностей, которые будут представлять собой промежуточное представление. Основными логическими сущностями будут:

1. Базовый блок — непрерывная последовательность команд, не содержащая промежуточных терминаторов (не считая последней команды).

2. Терминатор базового блока — условный или безусловный переход в базовый блок / базовые блоки.

3. Команда, оперирующая значениями — операция вида

$y \leftarrow op\ x1, x2, x3$ ... , где y, x1, x2, x3 –значения

4. Значение — абстракция, определяющая операнд команды, может быть именем переменной, типом, числовой константой и так далее.

2.6 Проектирование оптимизатора

В данной работе оптимизатор будет выполнять минимальный набор оптимизаций, таких как удаление пустых блоков и мертвого кода. Оптимизатор принимает на вход промежуточное представление и возвращает оптимизированное промежуточное представление.

2.7 Проектирование дополнительных модулей

Для успешного завершения компиляции необходимо разработать модуль преобразования абстрактной логической модели промежуточного представления в код LLVM.

Также, для проверки корректности работы компилятора, необходимо разработать модуль для предоставления дебаг-информации — вывод структуры синтаксического дерева и генерацию Dot графа для промежуточного представления для визуализации структуры скомпилированной программы.

3 Реализация

В данном разделе будут описаны детали реализации отдельных компонент компилятора, особое внимание будет уделено генерации промежуточного представления и реализации сборщика мусора.

3.1 Реализация лексического анализатора

Реализация лексического анализатора основана на использовании регулярных выражений. Основным элементом для работы лексического анализатора является класс токен (см. Листинг 16).

Листинг 16 — Класс токен

```
1.  public class Token {
2.
3.      private final String content;     // содержимое токена
4.      private final TokenType tokenType;// тип токена
5.      private final Position start;     // начало токена
6.      private final Position end;       // конец токена
7.
8.      public enum TokenType {
9.          PLUS("else","else"),
10.         MINUS("minus","\\-"),
11.         PERCENT("percent","\\%")
12.           ...
13.     }
14. }
```

Класс позиции определяет положение токена в исходном тексте программы (см. Листинг 17).

Листинг 17 — Класс позиции

```
1.  public class Position {
2.      private final int column;  // номер в строке
3.      private final int line;    // номер строки
4.      private final int position;// номер в тексте
5.  }
```

Тип токена определяет название группы в регулярном выражении и регулярное выражение распознающее токен. Лексический анализатор с помощью метода nextToken (см. Листинг 18) генерирует последовательность лексем языка.

Листинг 18 — Класс лексического анализатора

```
public class Lexer {
    private static final String PATTERN = Arrays
            .stream(Token.TokenType.values())
            .map(Token.TokenType::getRegexp)
            .collect(Collectors.joining("|"));

    public Token peekToken() {
        return lastToken;
    }

    public Token nextToken() {
        if (matcher.find()) {
            Token.TokenType tokenType =
                        Arrays.stream(Token.TokenType.values())
                    .filter(t ->
                         matcher.group(t.getGroupName()) != null)
                    .findFirst()
                    .orElse(Token.TokenType.EOF);
    ...
}
```

3.2 Реализация синтаксического анализатора

Синтаксический анализатор реализован с помощью метода рекурсивного спуска согласно приложению Б — грамматике языка. Каждый метод синтаксического анализатора возвращает объект (см. Листинг 19), описывающий вершину абстрактного синтаксического дерева, анализатор не поддерживает восстановление после ошибок.

Для элементов грамматики, которые не являются LL (1) реализован возвратный стек токенов, в который складываются токены, пока не будет получен однозначный путь разбора.

Листинг 19 — Структура синтаксического анализатора

```
public class Parser {
    public FileNode parse() {
        . . .
    }

    private TranslationNode parseTranslationNode() {
        . . .
    }

    private StatementNode parseCompoundStatement() {
        . . .
    }
    . . .
}
```

Примерами не LL (1) элементов грамматики являются случаи, связанные с идентификаторами классов. Для примера можно рассмотреть два исходных текста (см. Листинг 20 и Листинг 21).

Листинг 20 — Создание массива

```
1.   () int main ->
2.       Point[] p = new Point[4]
3.
4.       return 0
```

Листинг 21 — Вызов функции элемента массива

```
1.   () int main ->
2.       T[] Point = new T[4]
3.
4.       Point[3].functionCall()
5.
6.       return 0
```

Так как синтаксический анализатор не обладает информацией о классах, объявленных в данной области видимости на этапе синтаксического анализа, невозможно определить, чем является токен типа идентификатор (строка 2 и 4 соответственно) — названием класса или названием переменной.

Однозначно определить это можно только достигнув выражения внутри квадратных скобок. Если оно имеется, то это обращение к элементу массива. Иначе это объявление и токен является названием класса.

3.3 Реализация семантического анализатора

Реализация семантического анализатора заключается в агрегации абстрактных синтаксических деревьев и организации пространств имен с помощью импорта пакетов.

Семантический анализатор многопроходный. Во время проходов совершается набор действий для подготовки абстрактных синтаксических деревьев для генерации промежуточного представления:

1. Создание дерева областей видимости (англ. scope) (см. Листинг 22), объекты, определенные в родительской области, доступны в дочерних областях. Области видимости соответствуют вложенным конструкциям — классам, функциям, блокам если/иначе и циклам.

Листинг 22 — Класс области видимости

```
1.  public class Scope {
2.      private final Scope parentScope; // родительская область
3.      // внешние, дополнительные области, например import или наследование
4.      private final List<Scope> alternativeScopes = new ArrayList<>();
5.      // набор вершин, которые затрагивает область
6.      private final List<AstNode> nodes;
7.      // набор объявлений созданных в области
8.      private final List<AstNode> declarations;
9.  }
```

2. Проверка и сопоставление типов. При обработке каждой вершины типа Expression проводится проверка на корректность аргументов и возвращаемого значения. Например, для AdditionalExpression требуется наличие аргументов типа int, short, long, float, double. Для типов ссылка и boolean будет генерироваться ошибка.

3. Производится сопоставления вызова функции и ее определения, а также разрешение конфликтов.

4. Производится анализ и корректность использования методов и конструкторов классов.

5. Определяются возможные конфликты имен, за счет создания дерева областей видимости. В случае неразрешимости генерируется ошибка.

6. Производится анализ неявных приведений типов. Например, при сложении long и int результатом будет значение типа long.

7. Определяется корректность использования выражений в конструкциях языка. Например, if (2) некорректная условная конструкция. Также, например, нельзя использовать оператор this вне области видимости методов и конструкторов класса.

3.4 Реализация генератора промежуточного представления

Генератор промежуточного представления выполняет следующие преобразования:

1. Класс преобразуется в набор функций и структуру данных, содержащую требующие поля.

2. Для каждой функции вызывается генерация промежуточного представления.

3. Добавляются необходимые константные значения и строковые литералы

Для генерации промежуточного представления функции осуществляется рекурсивный спуск по синтаксическому дереву функции, которое было обогащено необходимой информацией во время семантического разбора.

3.4.1 Реализация модели промежуточного представления

Для описания промежуточно представления вводится набор сущностей. Интерфейс значения (см. Листинг 23).

Листинг 23 — Интерфейс значения

```
1.   public interface Value {
2.
3.       public Type getType();
4.
5.   }
```

Класс типа, реализует интерфейс значения (см. Листинг 24)

Листинг 24 — Класс типа

```
1.   public class Type implements Value {
2.       public static final Type VOID = new Type();
3.       public static final Type INT_1 = new Type();
4.       public static final Type INT_8 = new Type();
5.       public static final Type INT_16 = new Type();
6.       public static final Type INT_32 = new Type();
7.       public static final Type INT_64 = new Type();
8.
9.       @Override
```

```
10.       public Type getType() {
11.           return this;
12.       }
13.
14.       public int getSize() {
15.           if (this == INT_1 || this == INT_8) {
16.               return 1;
17.           } else if (this == INT_16) {
18.               return 2;
19.           } else if (this == INT_32) {
20.               return 4;
21.           } else if (this == INT_64) {
22.               return 8;
23.           } else {
24.               throw new IllegalArgumentException();
25.           }
26.       }
27.   }
```

Класс типа наследуют другие классы — ссылка, массив, функция и так далее. Ключевым классом является функция. Он содержит набор базовых блоков функции (см. Листинг 25).

Листинг 25 — Класс функции

```
1.   public class Function extends Type implements Value {
2.       private final List<BasicBlock> blocks;
3.       private final String name;
4.       private final boolean systemFunction;
5.       private BasicBlock currentBlock;
6.       private BasicBlock returnBlock;
7.       private Value returnValue;
8.       private VariableValue thisValue;
9.       private Type resultType;
10.      private List<Type> parameterTypes;
11.  }
```

При создании функции, набор базовых блоков является пустым. Он наполняется новыми базовыми блоками при проходе по синтаксическому дереву.

Базовый блок содержит набор входных и выходных блок, терминатор, набор команд (см. Листинг 26).

Листинг 26 — Класс базового блока

```
1.   public class BasicBlock {
2.       private final List<Command> commands;
3.       private final String name;
4.       private final List<BasicBlock> input;
5.       private final List<BasicBlock> output;
6.       private Terminator terminator;
7.   }
```

Каждая команда определяется как набор входных параметров, тип операции, место сохранения результата (см. Листинг 27).

Листинг 27 — Класс команды

```
1.   public class Command implements Value {
2.       private final Value result;
3.       private final Operation operation;
4.       private final List<Value> parameters;
5.   }
```

Для каждого типа вершины абстрактного синтаксического дерева существует обработчик, который выполняет необходимую трансляцию. Пример разбора блока while (см. Листинг 28).

Листинг 28 –Трансляция while

```
1.       private void translateWhile(Function function, WhileStatementNode node) {
2.           BasicBlock last = function.getCurrentBlock();
3.
4.           BasicBlock condition = function.appendBlock("while_condition");
5.           createBranch(last, condition);
6.
7.           whileToConditionBlock.put(node, condition);
8.
9.           Value value = translateExpression(function, node.getConditionNode());
10.          last = function.getCurrentBlock();
11.
12.          BasicBlock merge = function.appendBlock("merge");
13.          whileToMergeBlock.put(node, merge);
14.
15.          BasicBlock body = function.appendBlock("while_body");
16.          translateStatement(function, node.getBodyNode());
17.          createBranch(function.getCurrentBlock(), condition);
18.
19.          BasicBlock whileMerge = function.appendBlock("while_merge");
20.          createBranch(merge, whileMerge);
21.
22.          createConditionalBranch(last, value, body, merge);
23.      }
```

Для каждого выражения, возвращающего какое-то значение предусмотрена специальная функция трансляции (см. Листинг 29).

Листинг 29 — Пример трансляции выражения

```
1.       private Value translateBinaryOperation(Function function,
2.                                              ExpressionNode left,
3.                                              ExpressionNode right,
4.                                              Operation operation,
5.                                              Type resultType) {
6.           Value leftValue = translateExpression(function, left);
7.           Value rightValue = translateExpression(function, right);
8.
9.           Command command = new Command(createTempVariable(resultType),
10.                                         operation,
11.                                         List.of(leftValue, rightValue));
12.
13.          BasicBlock current = function.getCurrentBlock();
14.          current.addCommand(command);
15.
16.          return command.getResult();
17.      }
```

3.4.2 Реализация сборщика мусора

Реализация сборщика мусора заключается в генерации деструкторов для каждого класса и автоматическое применение умных указателей. Такой способ работы с памятью обычно применяется разработчиками в языке C++.

Алгоритм работы сборщика мусора:

1) При создании нового объекта счетчик копий устанавливается равным 0

2) При присвоении или при передаче объекта в функцию счетчик инкрементируется.

3) При возвращении значения из функции счетчик возвращаемого значения инкрементируется.

4) Если в переменной до присвоения было записано какое-либо не пустое значение, вызывается деструктор.

5) При выходе из функции деструктор вызывается на всех объектах типа ссылка.

6) Для временных объектов определено специальное поведение. Например, выражение new Point (1, 2).x в стандартном случае не вызовет деструктор.

7) Деструктор представляет собой функцию, которая декрементирует счетчик. При достижении нуля деструктор вызывается для каждого поля объекта типа ссылка.

8) На данный момент, разрешения циклических зависимостей в сборщике мусора не реализовано.

Вызов деструктора является виртуальным. Набор деструкторов хранится в статическом массиве, в котором каждому типу объекта соответствует требуемая функция деструктора. Для корректной работы такого механизма при аллокации нового объекта резервируется место для счетчика копий и номера типа объекта.

Каждый объект при создании хранит набор необходимой служебной информации. Код конструктора пустого объекта, полученный при генерации промежуточного представления, выглядит следующим образом (см. Листинг 30).

Листинг 30 — Конструктор пустого объекта

```
define %struct.$0_A* @$_constructor_0(){
    header_33:
        %$$_ret_value_1 = alloca %struct.$0_A*
        %$$_this_value_5 = alloca %struct.$0_A*
        // минимальный размер объекта = 8 байт
        %0 = call i64* @malloc(i32 8)
        %1 = bitcast i64* %0 to %struct.$0_A*
        store %struct.$0_A* %1, %struct.$0_A** %$$_this_value_5
        store %struct.$0_A* %1, %struct.$0_A** %$$_ret_value_1

        // Обнуление счетчика ссылок
        %3 = getelementptr inbounds %struct.$0_A ,
                                %struct.$0_A* %1 , i64 0 , i32 0
        store i32 0, i32* %3
        // Тип структуры - описывается с помощью int32
        %4 = getelementptr inbounds %struct.$0_A ,
                                %struct.$0_A* %1 , i64 0 , i32 1
        store i32 3, i32* %4
        br label %entry_38
    return_34:
        %6 = load %struct.$0_A*,%struct.$0_A** %$$_ret_value_1
        ret %struct.$0_A* %6
    entry_38:
        // Блок пустой, поскольку конструктор пустой
        br label %return_34
}
```

При вызове виртуальной функции вызывается общая функция, которая с помощью статического массива делегирует вызов необходимой функции в зависимости от типа объекта. Данный способ похож на таблицу виртуальных функций в языке C++.

Первой и обязательной (генерируется автоматически и недоступен пользователю) виртуальной функцией в программе является деструктор. Функция деструктора вызывается для всех объектов. Для каждого объекта определяется и вызывается его специальный деструктор.

Листинг 31 — Виртуальный деструктор

```
@$$$_destructors_array = global [3 x void (i32*)*] [
                void (i32*)* @$_common_struct_destructor_,
                void (i32*)* @$_common_array_destructor_,
                void (i32*)* @$_destructor_0]

define void @$_common_destructor(i32*){
    header_19:
        %$$_this_value_4 = alloca %struct.$$$_commmon_struct*
        %1 = bitcast i32* %0 to %struct.$$$_commmon_struct*
        store %struct.$$$_commmon_struct* %1,
                    %struct.$$$_commmon_struct** %$$_this_value_4
        br label %end_function_destructor_26
    return_20:
        ret void
    end_function_destructor_26:
```

```
16.           %12 = getelementptr inbounds %struct.$$$_commmon_struct ,
17.                      %struct.$$$_commmon_struct* %7 , i64 0 , i32 1
18.           %13 = load i32,i32* %12
19.           %14 = sub i32 %13,1
20.           %15 = getelementptr inbounds [3 x void (i32*)*]  ,
21.                      [3 x void (i32*)*] * @$$$_destructors_array ,
22.                      i64 0 , i32 %14
23.           %16 = load void (i32*)*,void (i32*)** %15
24.           call void %16(i32* %0)
25.           br label %return_20
26.   }
```

В примере пропущены (т.к. не существенны) необходимые проверки на null.

3.4.3 Реализация многомерного массива

Для выделения памяти многомерного массива требуется рекурсивная генерация кода (см. Листинг 32).

Листинг 32 — Код аллокации двухмерного массива (схематично)

```
1.  array = alloc (size_0)
2.  iterator_0 = 0
3.  while iterator_0 < size_0
4.     array[iterator_0] = alloc(size_1)
5.     iterator_1=0
6.
7.     while iterator_1 < size_1
8.           array[iterator_0][iterator_1] = 0
9.           iterator_1++
10.
11.    iterator_0++
```

Аналогично работает деструктор –для каждого объекта массива вызывается виртуальный деструктор.

4 Тестирование

Для проведения тестирования разработанного компилятора, необходимо провести тестирование отдельных модулей и интеграционное тестирование.

4.1 Тестирование лексического и синтаксического анализаторов

Для тестирования лексического и синтаксического анализатора необходимо протестировать все возможные правила грамматики, описанные в приложении Б. Для этого был разработан набор специальных тестов.

Также был разработан модуль для вывода дебаг-информации, он позволяет визуализировать синтаксическое дерево функции (см. Листинг 33).

Листинг 33 — Вывод абстрактного синтаксического дерева

```
....FunctionDefinition:
........Function:
............BasicType: INT
............Parameters:
........Identifier: main
........CompoundStatement:
............ReturnStatement:
................Int: 0
............EmptyStatement
```

4.2 Тестирование генератора промежуточного представления

Для тестирования генератора промежуточного представления разработан набор различных тестов, а также разработан модуль визуализации промежуточного представления в виде графа потока управления (см. Рисунок 2).

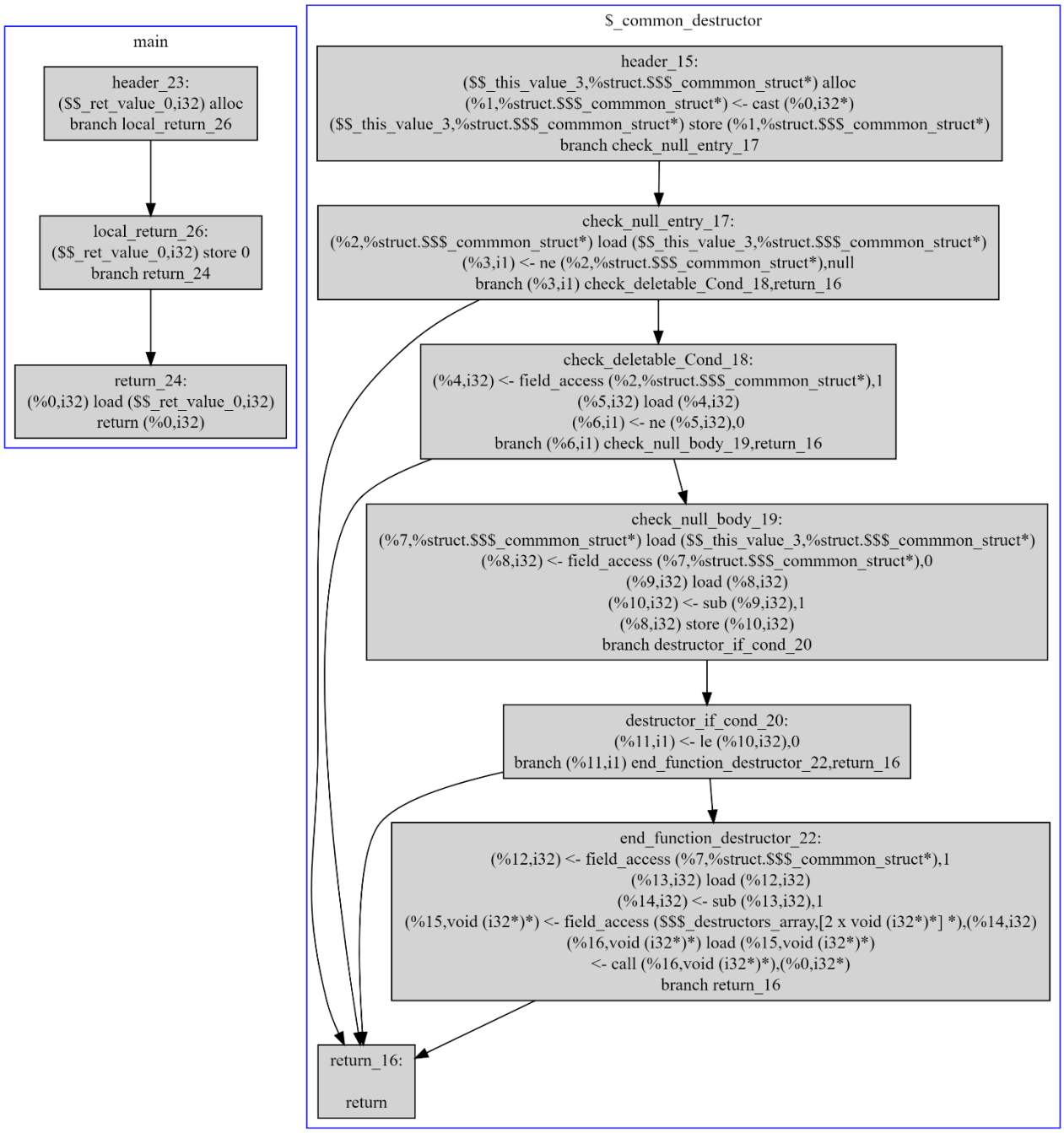

Рисунок 2 — Граф потока управления

Черными дугами обозначаются условные и безусловные переходы.

4.3 Тестирование реализации сборщика мусора

Для тестирования корректности работы сборщика мусора в компиляторе поддерживается тестирование с запуском программы с помощью valgrind. Это необходимо для определения утечек памяти, некорректных операций чтения и записи и так далее.

Также разработан набор тестов, демонстрирующий корректность работы сборщика мусора. Один из тестов представлен далее (см. Листинг 34).

Листинг 34 — Пример теста

```
1.  class Point
2.      int x
3.      int y
4.
5.      (int x, int y) ->
6.          this.x = x
7.          this.y = y
8.
9.  (Point p) Point incrementPoint ->
10.     if p == null
11.         return new Point(3, 2)
12.     return new Point(p.x + 1, p.y +1 )
13.
14.  () int main ->
15.     Point p = new Point(1, 2)
16.     Point s = null
17.
18.     incrementPoint(p).x
19.
20.     return p.x + p.y
```

4.4 Интеграционное тестирование

Для проведения интеграционного тестирования был разработан проект, реализующий быструю сортировку массива (см. Листинг 35). В нем также применяются строки и строковые литералы, многомерные массивы объектов и составные классы.

Листинг 35 — Код быстрой сортировки

```
(int[] numbers, int left, int right) void quickSort ->
    int pivot
    int l_hold = left
    int r_hold = right

    pivot = numbers[left]

    while left < right
        while numbers[right] >= pivot && left < right
            right--

        if left != right
            numbers[left] = numbers[right]
            left++

        while numbers[left] <= pivot && left < right
            left++

        if left != right
            numbers[right] = numbers[left]
            right--

    numbers[left] = pivot
    pivot = left

    left = l_hold
    right = r_hold

    if left < pivot
        quickSort(numbers, left, pivot -1)

    if right > pivot
        quickSort(numbers, pivot + 1, right)
```

ЗАКЛЮЧЕНИЕ

В данной работе был разработан компилятор объектно-ориентированного статически типизированного языка программирования для LLVM. Были исследованы различные этапы компиляции.

Разумеется, для того чтобы можно было использовать этот язык полноценно, необходимо разработать стандартную библиотеку языка, обеспечить интерфейсы ввода/вывода, работы с сетью, операционной системой и так далее. Также необходимо поддержать возможность многопоточного программирования, т.е. разработать и специфицировать модель памяти языка программирования. Данные аспекты будут рассмотрены в выпускной квалификационной работе.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 Рейуорд-Смит В.Д. Теория формальных языков. Вводный курс. - Москва: Радио и связь, - 1988.

2 Мартыненко Б.К. Языки и трансляции. - Санкт-Петербург: Издательство С.-Петерб. ун-та, - 2009.

3 Ахо А., Сети Р., Ульман Д. Компиляторы. Принципы, технологии, инструменты. 2nd ed. - Москва: Издательский дом «Вильямс», - 2001.

4 Карпов Ю.Г. Теория и технология программирования. Основы построения трансляторов. - Санкт-Петербург: БХВ-Петербург, - 2005.

5 Мозговой М.В. Классика программирования: алгоритмы, языки, автоматы, компиляторы. Практический подход. - Санкт-Петербург: Наука и Техника, - 2006.

6 Хантер Р. Основные концепции компиляторов. - Москва: Издательский дом «Вильямс», - 2002.

7 Вирт Н. Построение компиляторов. - Москва: ДМК-Пресс, - 2010.

8 https://github.com/don-dron/compiler // Github. 2021. (дата обращения: 23.11.2021).

9 Пратт Т. Языки программирования: разработка и реализация. - Москва: Мир, - 1979.

ПРИЛОЖЕНИЯ

Приложение А — Лексика языка lang

IntConstant ::= [1-9] [0-9] *

FloatConstant ::= [0-9]+ ‘.’ [0-9]*

KeyWord ::= class | while | if | elif | else | return | break | continue | null | void | int
| long | float | double | short | boolean | char | import | this
| true | false | new

Identifier ::= [_a-zA-Z][_a-zA-Z0-9]*

StringConstant ::= “ (.|[^\”])*? ”

Arrow ::= ‘-’ ‘>’

Operator ::= ‘+’ | ‘-‘ | ‘*’ | ‘/’ | ‘%’ | ‘!’ | ‘==’ | ‘!=’ | ‘<=’ | ‘>=’ | ‘<’ | ‘>’ | ‘&&’ | ‘||’ |
‘:’ | ‘++’ | ‘ -- ‘ | ‘=’ | ‘?’ | ‘,’ | ‘.’

Comment ::= ‘/’ ‘/’ .* ‘\n’

Tab ::= ‘\t’ | (‘ ’){4}

Braces := ‘[‘ | ‘]’ | ‘(‘ | ‘)’

Приложение Б — Формальная грамматика языка lang

Translation ::= GlobalStatement | Import

Import ::= ‘import’ Expression

GlobalStatement ::= DeclarationStatement

DeclarationStatement ::= Type Identifier

| Type Identifier ‘->’ Statement

| Type Identifier ‘=’ Expression

FunctionType ::= ParametersList Type

ParametersType ::= ‘(‘ ‘)’ | ‘(‘ (Type Identifier) (‘,’ Type Identifier)* ‘)’

Type ::= PrimaryType | ArrayType | FunctionType

ArrayType ::= PrimaryType (‘[‘ ‘]’) +

PrimaryType ::= void | int | short | char | long | float | double | Identifier

Statement ::= IfStatement | ElifStatement | ElseStatement | CompoundStatement

| WhileStatement | ReturnStatement | BreakStatement

| ClassStatement | ContinueStatement | DeclarationStatement

| ExpressionStatement

IfStatement ::= ‘if’ Expression NewLine Statement

ElIfStatement ::= ‘elif’ Expression NewLine Statement

ElseStatement ::= ‘else’ NewLine Statement

WhileStatement ::= ‘while’ Expression NewLine Statement

ReturnStatement ::= ‘return’ Expression?

BreakStatement ::= ‘break’

ClassStatement ::= ‘class’ Identifier NewLine Translation

ContinueStatement ::= ‘continue’

ExpressionStatement ::= Expression

CompoundStatement := (Tab* Statement) +

Expression ::= ConditionalExpression (‘=’ ConditionalExpression)*

ConditionalExpression ::= LogicalOrExpression (‘?’ Expression ‘:’ Expression) ?

LogicalOrExpression ::= LogicalAndExpression (‘||’ LogicalAndExpression)*

LogicalAndExpression ::= EqualityExpression (‘&&’ EqualityExpression)*

EqualityExpression ::= RelationalExpression ((‘==’ | ‘!=’) RelationalExpression)*

RelationalExpression ::=

AdditiveExpression ((‘<=’ | ‘>=’ | ‘>’ | ‘<’) AdditiveExpression)*

AdditiveExpression ::= MultiplicativeExpression ((‘+’ | ‘-’) MultiplicativeExpression)*

MultiplicativeExpression ::= CastExpression ((‘*’ | ‘/’ | ‘%’) CastExpression)*

CastExpression ::= (‘<’ Type ‘>’)? UnaryExpression

UnaryExpression ::= (‘++’ | ‘ -- ‘)? UnaryExpression | PostExpression

PostExpression ::= PrimaryExpression ‘++’

| PrimaryExpression ‘ -- ‘

| PrimaryExpression ‘.’ Identifier

| PrimaryExpression ‘(‘ ‘)’

| PrimaryExpression ‘(‘ Expression (‘,’ Expression)* ‘)’

| PrimaryExpression ‘[‘ Expression ‘]’

PrimaryExpression ::= Identifier | ‘this’ | StringConstant | ‘null’ | ‘true’ | ‘false’

| ‘(‘ Expression ‘)’ | ConstructorCall

ConstructorCall ::= ‘new’ Type (‘[‘ Expression ‘]’)+

| ‘new’ Type ‘(‘ ‘)’

| ‘new’ Type ‘(‘ Expression (‘,’ Expression)* ‘)’