Министерство науки и высшего образования РФ Федеральное государственное бюджетное образовательное учреждение высшего образования
«Волгоградский государственный технический университет»
Факультет электроники и вычислительной техники
Кафедра «Программное обеспечение автоматизированных систем»

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
к семестровой работе

по дисциплине Теория формальных языков и методов трансляции

на тему «Разработка компилятора исходного кода языка Scala в байт-код виртуальной машины Java»

Студенты Д.А. Смирнов, Р.В. Юрасов, Н.В. Денисов
Группа ПрИн-467

Руководитель работы (проекта) _____ О.А. Сычев
(подпись и дата подписания) (инициалы и фамилия)

Волгоград 2026

Оглавление

1) Описание синтаксиса и семантики (смысла) компилируемого языка

Scala — современный статически типизированный язык программирования, работающий на JVM и сочетающий объектно-ориентированный и функциональный подходы. Он ориентирован на выразительность, мощную систему типов и компактность кода. Синтаксис Scala гибкий и ёмкий, минимизирующий шаблонный код за счёт «синтаксического сахара». Язык обладает полной взаимосовместимостью с Java.

Объявление переменных использует ключевые слова val (неизменяемые) и var (изменяемые). Функции определяются с def, классы — с class.

Примеры синтаксиса:

Объявление переменных:

```
val x: Int = 5           // Неизменяемая переменная (final)
var y: Int = 10          // Изменяемая переменная
```

Функции:

```
def helloWorld(greetingNum: Int): Unit = {
        Predef.println("Hello, World! That's my greeting number " +
greetingNum)
    }

def multiply(a: Int, b: Int): Int = return a * b
```

Классы:

```
class Account {
    private var balance: Int = 100
    protected var ownerName: String = "DefaultOwner"
    var id: Int = 1

    def getBalance(): Int = balance

    protected def addFunds(amount: Int): Unit = {
        balance = balance + amount
    }
```

```
    def printInfo(): Unit = {
        Predef.print("Account id=")
        Predef.print(id)
        Predef.print(" owner=")
        Predef.print(ownerName)
        Predef.print(" balance=")
        Predef.println(balance)
    }
}

class SavingsAccount extends Account {
    def deposit(amount: Int): Unit = {
        addFunds(amount)
    }

    def getOwner(): String = {
        ownerName
    }

    def setOwner(name: String): Unit = {
        ownerName = name
    }
}
```

Условные операторы

```
if (x > 0) {
    Predef.println("Positive")
} else if (x < 0) {
    Predef.println("Negative")
} else {
    Predef.println("Zero")
}
```

Циклы

```
// Цикл for с итерацией по массиву
val arr: Array[Int] = Array(10, 20, 30, 40, 50)
for (x: Int <- arr) {
     Predef.print(x)
     Predef.print(" ")
}

// Цикл for с генератором (вместо until могут быть to, by)
for (i: Int <- 0 until arr.length()) {
     Predef.print(arr(i))
     Predef.print(" ")
}
```

```
// Цикл while
while (condition) {
    condition = counter < 10
    counter = counter + 1
}

// Цикл do-while
do {
    counter++
} while (counter < 5)
```

Одной из ключевых особенностей Scala является смесь объектно-ориентированной и функциональной(практически все в языка является выражением и имеет результат вычисления) парадигмы. Другая ключевая особенность – система идентификаторов. Идентификаторами в ней могут быть привычные операторы других языков программирования (арифметические, синтаксические). Символ “;”, обозначающий конец выражения, опционален. В случае его отсутствия конец/начало выражения определяются с помощью трех правил, описанных в спецификации языка.
При написании метода можно опустить ключевое слово return. В таком случае возвращаемым значением считается результат выполнения последнего выражения в методе.

Scala 2 полностью совместима с Java и работает поверх JVM, позволяя напрямую использовать библиотеки и фреймворки экосистемы Java. Для асинхронного и параллельного программирования применяются Future, Promise и библиотека Akka, реализующая акторную модель. Экосистема Scala включает инструмент сборки sbt, а также популярные библиотеки и фреймворки, такие как Play Framework для веб-разработки и Apache Spark, где Scala является одним из основных языков.

2) Пример программы на компилируемом языке, с использованием всех реализованных возможностей

```
// Test 14: BubbleSort with range generators
final class A {
    def main(args: Array[String]): Unit = {
        new BubbleSort().bubbleSort()
    }
}

final class BubbleSort {
```

```
    def bubbleSort(arr: Array[Int]): Unit = {
        for (i: Int <- 0 until (arr.length() - 1)) {
            for (j: Int <- 0 until (arr.length() - i - 1)) {
                if (arr(j) > arr(j + 1)) {
                    val temp: Int = arr(j)
                    arr(j) = arr(j + 1)
                    arr(j + 1) = temp
                }
            }
        }
    }

    def printArray(arr: Array[Int]): Unit = {
        for (i: Int <- 0 until arr.length()) {
            Predef.print(arr(i))
            Predef.print(" ")
        }
    }

    def bubbleSort(): Unit = {
        Predef.println("=== Bubble Sort Test ===")

        Predef.println("Размер: ")
        val n: Int = StdIn.readInt()
        val array: Array[Int] = new Array[Int](n)

        for(i: Int <- 0 until n) {
            Predef.println("Элемент: ")
            array(i) = StdIn.readInt()
        }

        Predef.println("Before:")
        printArray(array)

        bubbleSort(array)

        Predef.println("\nAfter:")
        printArray(array)
    }
}

final class A {
    def main(args: Array[String]): Unit = {
        new OopTest().oopTest()
    }
}

class Mother {
    def speak(): Unit = {
        Predef.println("Mother")
    }
}
```

```
class Son extends Mother {
    override def speak(): Unit = {
        super.speak()
        Predef.println("Son")
    }
}

class Daughter extends Mother {
    override def speak(): Unit = {
        Predef.println("Daughter")
    }
}

final class OopTest {
    def oopTest(): Unit = {
        Predef.println("=== OOP Inheritance Test ===")

        val mom: Mother = new Mother()
        val son: Mother = new Son()
        val daughter: Mother = new Daughter()

        Predef.println("--- Individual calls ---")
        mom.speak()
        son.speak()
        daughter.speak()

        Predef.println("--- Polymorphic array ---")
        val family: Array[Mother] = Array(mom, son, daughter)
        for (member: Mother <- family) {
            member.speak()
            Predef.println("---")
        }
    }
}

final class A {
    def main(args: Array[String]): Unit = {
        new OverloadingTest().overloadingTest()
    }
}

class Vec(var x: Int,var y: Int) {
    def +(other: Vec): Vec = {
        return new Vec(x+other.x, y - other.y)
    }
}

final class OverloadingTest {
    def overloadingTest(): Unit = {
        var v1: Vec = new Vec(5,5)
        var v2: Vec = new Vec(1,1)
        var v3: Vec = v1 + v2
```

```
        Predef.println(v3.x)
        Predef.println(v3.y)
    }
}
```

3) Таблица с распределением реализованной части

Язык:	Scala 2.13.18
Участники:	Смирнов Д.С, Юрасов Р.В, Н.В. Денисов

Возможность	Вес	Особенности в языке	Кто реализует
Локальные переменные встроенных типов данных: ● целые числа (один тип), ● символы ● строки Выражения с использованием локальных переменных, арифметических операций (4 вида), операций сравнения и присваивания	4	val - неизмен. пер. var - измен. пер. Поддерживаемые типы данных: Int, String, Boolean, Char Операции: +, -, *, /, %, <, <=, >, >=, ==, !=	Смирнов
Одномерные массивы, операция доступа к элементу массива	3	Индексация с 0 Array: Неизменяемый размер Обращение к элементу массива arr(i)	Смирнов
Числовые (целые числа), символьные и строковые константы (литералы) с поддержкой всех видов констант и служебных последовательностей символов	2	Целые: 123, символьные: 'a', строковые: "hello", с поддержкой escape-последовательностей: \n, \t, \" и т.д.	Юрасов

Управляющие структуры: развилки	3	1) if 2) if-else 3) if-else if-...	Юрасов
Управляющие структуры: циклы	3	for, while, do while Итерирование по циклу for через массив или генераторы	Смирнов
Функции	3	def <<name>>() : <<type>>= {}	Юрасов
Классы/функции для работы с консолью (ввод/вывод базовых типов данных)	2	Predef.println() Predef.print() StdIn.readInt()	Юрасов
Классы, свойства и методы, одиночное наследование с динамическим связыванием	4	class Person { }	Смирнов
Переменные, константы и операции (арифметические, сравнения, присваивания) для чисел с плавающей точкой	3	Double	Юрасов
Логические операции (И, ИЛИ, НЕ)	2	&& || !	Смирнов
Контроль доступа к свойствам и методам класса (открытые/ защищенные свойства и методы)	1	public, private, protected Ключевого слова public не существует. Все объекты по умолчанию считаются public, если явно не указан другой уровень доступа.	Смирнов
Перегрузка функций по типу параметров	1		Юрасов

Перегрузка операций	4	Перегрузка операций по типу и количеству параметров. В случае составных (самописных) операторов приоритет определяется по первому символу.	Смирнов
Перекрытия имен переменных в областях видимости	2	Система перекрытия имен переменных в областях видимости, сходная по работе с C++: в пересекающихся областях видимости могут существовать переменные с одинаковыми именами и разными типами, в таком случае происходит “затенение” переменной родительской области видимости	Юрасов
Расширенная система именований переменных, методов, классов	2	Именами методов могут являться арифметические операнды (+, - и т.д.), логические (<=, >). Можно делать собственные составные операторы	Юрасов

		(например, +------+).	
Многомерные массивы	2		Денисов
Перечисляемый тип данных	2		Денисов
Обработка исключений	3		Денисов
Конструкторы с параметрами	1	Первичные и вторичные конструкторы. Пользователь не может писать тело первичного конструктора. только его параметры, которые впоследствии станут полями класса. Первичный конструктор всегда неявно вызывает конструктор родителя, чтобы обеспечить корректную инициализацию данных во всей иерархии. Также первичный конструктор должен идти первой инструкцией во вторичном. Пример первичного	Юрасов

		конструктора: class Point(val x: Int, val y: Int)	
Статические свойства и методы классов	2	В Scala нет ключевого слова static, вместо него используется object – это синглтон, все члены которого доступны без создания экземпляра и фактически выполняют роль статических. Если class и object имеют одно и то же имя и объявлены в одном файле, они образуют companion object: такой object имеет доступ к приватным членам класса и наоборот.	Денисов
Подстановки значений переменных и выражений в строки	2		Денисов

4) Таблица лексем

Служебные лексемы (символы и операторы)

Вид лексемы	**Семантическое значение**	**Описание и способ задания**
{ }	Открывающая и закрывающая фигурные скобки	Используются для задания блоков кода: тела классов, функций, условных операторов и циклов.
( )	Открывающая и закрывающая круглые скобки	Используются при вызове функций, в выражениях и условиях, при обращении к элементу массива по индексу.
[ ]	Открывающая и закрывающая квадратные скобки	Используются для объявления массивов. Между [ ] указывается тип элементов
;	Конец инструкции	Завершает оператор или выражение (опциональный оператор).
,	Разделитель	Разделяет аргументы функций, элементы массивов и параметры.
:	Аннотация типа	Используется при указании типа переменной или параметра.
.	Доступ к члену	Используется для доступа к полям и методам объекта.
=	Присваивание	Присваивает значение переменной.
+ - * / %	Арифметические операции	Операции сложения, вычитания, умножения, деления и остатка (считаются идентификаторами).

+= -= *= /= %=	Составные операции присваивания	Выполняют операцию и присваивание одновременно (считаются идентификаторами).
== !=	Операции сравнения	Проверка на равенство и неравенство (считаются идентификаторами).
< > <= >=	Операции сравнения	Сравнение числовых значений.
&&	Логическое И	Логическая операция И.
!	Логическое НЕ	Логическое отрицание.
<-	Извлечение элемента генератора	Извлечение значения из контекста (биндинга) с возможным побочным эффектом сопоставления или генерации.

Идентификаторы

Вид лексемы	**Семантическое значение**	**Описание и способ задания**
Идентификатор	Имя сущности	Названия переменных, функций, классов и параметров. Могут содержаться буквы, цифры, специальные символы: _, +, !, #, %, &, *, -, /, <, >, ?, @, \, ^, ~, +=, -=, *=, /=, %=, ==, !=, >=, <=. В случае, если начинаются с буквы или цифры, перед указанием вышеперечисленных символов необходимо поставить символ ‘_’. Если же идентификатор начинается с специального символа, в нем могут быть использованы только специальные символы. При этом, после специального символа не могут

		содержаться буквы и/или цифры (исключение: символ “_”).

Ключевые слова управления потоком

Лексема	**Семантическое значение**	**Описание**
if	Условный оператор	Выполнение кода при истинном условии.
else	Альтернативная ветка	Выполняется, если условие if ложно.
for	Цикл	Итерация по диапазону или массиву.
while	Цикл с условием	Выполняется, пока условие истинно.
do	Цикл do-while	Минимум одна итерация цикла.
return	Возврат значения	Возврат значения из функции.

Ключевые слова объявлений

Лексема	**Семантическое значение**	**Описание**
var	Изменяемая переменная	Объявление переменной с возможностью изменения.
val	Неизменяемая переменная	Объявление константной переменной.
def	Функция	Объявление функции.

class	Класс	Объявление класса.
this	Конструктор	Объявление конструктора класса.
enum	Перечисление	Объявление enum-типа.

Модификаторы доступа и поведения

Лексема	**Семантическое значение**	**Описание**
final	Запрет наследования	Класс или метод нельзя переопределять.
override	Переопределение	Переопределение метода суперкласса.
private	Приватный доступ	Доступ только внутри класса.
protected	Защищённый доступ	Доступен в классе и наследниках.

Ключевые слова типов

Лексема	**Семантическое значение**	**Описание**
Int	Целое число	32-битное целое число.
Double	Вещественное число	Число двойной точности.
String	Строка	Последовательность символов.
Char	Символ	Один символ.
Boolean	Логический тип	Значение true или false.

Array	Массив	Коллекция элементов одного типа.
Unit	Пустой объект	Аналог void из C++, Java

Литералы (константы)

Вид лексемы	**Семантическое значение**	**Описание**
Целочисленный литерал	Integer	Десятичные, двоичные (0b) и шестнадцатеричные (0x) числа.
Вещественный литерал	Double / Float	Числа с плавающей точкой.
Символьный литерал	Char	Один символ в одинарных кавычках.
Строковый литерал	String	Строка в кавычках. Поддерживаются escape-последовательности
true / false	Логические значения	Булевы константы.

Комментарии

Вид комментария	**Семантическое значение**	**Описание**
//	Однострочный комментарий	Игнорируется компилятором до конца строки.
/* */	Многострочный комментарий	Может быть вложенным, игнорируется компилятором.

5) Грамматика языка

```
%nonassoc LOW_PREC
%nonassoc RETURN IF FOR NL
%nonassoc ELSE WHILE DO TRY THROW VAL VAR NEW YIELD MATCH CASE
%right '=' PLUS_ASSIGNMENT MINUS_ASSIGNMENT MUL_ASSIGNMENT DIV_ASSIGNMENT
MOD_ASSIGNMENT
%left '|' ID_VERTICAL_SIGN
%left '^' ID_CIRCUMFLEX
%left '&' ID_AMPERSAND
%left EQUAL NOT_EQUAL ID_EQUALITY '!' ID_EXCLAMATION
%left '<' '>' GREATER_OR_EQUAL LESS_OR_EQUAL ID_LESS ID_GREAT
%right ID_COLON
%left '+' '-' ID_MINUS ID_PLUS
%left '*' '/' '%' ID_ASTERISK ID_SLASH ID_PERCENT
%left '~' ID_TILDE
%left ID '#' '?' '@' '\\'
%right UMINUS UPLUS
%right ULOGNOT UBINNOT
%left '.'
%nonassoc ':'
%nonassoc '(' '['
%nonassoc CATCH
%nonassoc FINALLY
%nonassoc END_TEMPLATE


%start scalaFile


%%

scalaFile: topStatSeq ;

expr: IF '(' expr ')' nls expr ELSE expr
    | IF '(' expr ')' nls expr
    | WHILE '(' expr ')' nls expr
    | tryExpr
    | DO expr semio WHILE '(' expr ')'
    | THROW expr
    | RETURN
    | RETURN expr
    | FOR '(' enumerators ')' nls expr
    | FOR '(' enumerators ')' nls YIELD expr
    | FOR '{' enumerators '}' nls expr
    | FOR '{' enumerators '}' nls YIELD expr
    | infixExpr
    | assignment
    ;

assignment: fullID '=' expr
          | simpleExpr '.' fullID '=' expr
```

```
          | simpleExpr1 argumentExprs '=' expr
          ;

enumerators: generator
           | enumerators semi enumeratorPart
           ;

enumeratorPart: generator
              | fullID compoundTypeO '=' expr
              ;

generator: fullID compoundTypeO LEFT_ARROW expr
         ;

compoundTypeO: /* empty */
          | ':' compoundType
          ;

infixExpr: prefixExpr
         | infixExpr '+' nlo infixExpr
         | infixExpr '!' nlo infixExpr
         | infixExpr '#' nlo infixExpr
         | infixExpr '%' nlo infixExpr
       | infixExpr '&' nlo infixExpr
       | infixExpr '*' nlo infixExpr
       | infixExpr '-' nlo infixExpr
       | infixExpr '/' nlo infixExpr
       | infixExpr '<' nlo infixExpr
       | infixExpr '>' nlo infixExpr
       | infixExpr '?' nlo infixExpr
       | infixExpr '@' nlo infixExpr
       | infixExpr '\\' nlo infixExpr
       | infixExpr '^' nlo infixExpr
       | infixExpr '~' nlo infixExpr
       | infixExpr PLUS_ASSIGNMENT nlo infixExpr
       | infixExpr MINUS_ASSIGNMENT nlo infixExpr
       | infixExpr MUL_ASSIGNMENT nlo infixExpr
       | infixExpr DIV_ASSIGNMENT nlo infixExpr
       | infixExpr MOD_ASSIGNMENT nlo infixExpr
       | infixExpr EQUAL nlo infixExpr
       | infixExpr NOT_EQUAL nlo infixExpr
       | infixExpr GREATER_OR_EQUAL nlo infixExpr
       | infixExpr LESS_OR_EQUAL nlo infixExpr
       | infixExpr ID nlo infixExpr
       | infixExpr ID_EQUALITY nlo infixExpr
       | infixExpr ID_VERTICAL_SIGN nlo infixExpr
       | infixExpr ID_AMPERSAND nlo infixExpr
       | infixExpr ID_CIRCUMFLEX nlo infixExpr
       | infixExpr ID_LESS nlo infixExpr
       | infixExpr ID_GREAT nlo infixExpr
       | infixExpr ID_MINUS nlo infixExpr
       | infixExpr ID_PLUS nlo infixExpr
       | infixExpr ID_ASTERISK nlo infixExpr
```

```
       | infixExpr ID_SLASH nlo infixExpr
       | infixExpr ID_PERCENT nlo infixExpr
       | infixExpr ID_EXCLAMATION nlo infixExpr
       | infixExpr ID_TILDE nlo infixExpr
       | infixExpr ID_COLON nlo infixExpr
         ;

prefixExpr: simpleExpr
          | '+' simpleExpr %prec UPLUS
          | '-' simpleExpr %prec UMINUS
          | '~' simpleExpr %prec UBINNOT
          | '!' simpleExpr %prec ULOGNOT
          ;

simpleExpr: NEW stableId argumentExprs
        | NEW stableId
        | NEW ARRAY '[' compoundType ']' argumentExprs
        | NEW ARRAY argumentExprs
          | '{' blockStats '}'
          | simpleExpr1
          ;

simpleExpr1: literal
           | fullID
           | SUPER '.' fullID
           | THIS
           | simpleExpr '.' THIS
           | simpleExpr '.' fullID
           | '(' expr ')'
           | '(' ')'
           | simpleExpr1 argumentExprs
           | ARRAY '[' compoundType ']' argumentExprs
           | ARRAY argumentExprs
           ;

argumentExprs: '(' exprs ')' ;

exprs: /* empty */
     | expr
     | exprs ',' expr
     ;

compoundType: simpleType
            | compoundType WITH simpleType
            ;

simpleType: stableId
          | ARRAY '[' compoundType ']'
          ;

stableId: fullID
        | SUPER '.' fullID
        | THIS '.' fullID
```

```
        | stableId '.' fullID
        ;

blockStats: blockStat
          | blockStats semi blockStat
          ;

blockStat: /* empty */
       | varDefs
         | expr
         ;

ids: fullID
   | ids ',' fullID
   ;

tryExpr: TRY expr
       | TRY expr CATCH expr
       | TRY expr CATCH expr FINALLY expr
       | TRY expr FINALLY expr
       ;

funcParamClause: nlo '(' ')'
               | nlo '(' funcParams ')'
               ;

funcParams: funcParam
          | funcParams ',' funcParam
          ;

funcParam: fullID compoundTypeO assignExprO ;

assignExprO: /* empty */
           | '=' expr
           ;

classParamClause: nlo '(' ')'
                | nlo '(' classParams ')'
                ;

classParams: classParam
           | classParams ',' classParam
           ;

classParam: modifiers VAL fullID ':' compoundType assignExprO
          | modifiers VAR fullID ':' compoundType assignExprO
          | modifiers fullID ':' compoundType assignExprO
          ;

modifiers: /* empty */
         | modifiers modifier
         ;
```

```
modifier: ABSTRACT
        | FINAL
        | SEALED
        | accessModifier
        | OVERRIDE
        ;

accessModifier: PRIVATE
            | PROTECTED
            ;

templateBody: nlo '{' templateStats '}' ;

templateStats: templateStat
             | templateStats semi templateStat
             ;

templateStat: /* empty */
            | modifiers def
            | modifiers dcl
            ;

dcl: VAL ids ':' compoundType
   | VAR ids ':' compoundType
   | DEF funSig compoundTypeO
   ;

funSig: fullID funcParamClause ;

varDefs: VAL ids compoundTypeO '=' expr
       | VAR ids compoundTypeO '=' expr
       ;

def: varDefs
   | DEF funDef
   | tmplDef
   ;

funDef: funSig compoundTypeO '=' expr
      | THIS funcParamClause '=' constrExpr
      ;

constrExpr: THIS argumentExprs
          | '{' THIS argumentExprs semi blockStats '}'
          | '{' THIS argumentExprs '}'
          ;

tmplDef: CLASS classDef
       | OBJECT fullID classTemplateOpt
       | TRAIT fullID traitTemplateOpt
       | ENUM enumDef
       ;
```

```
classDef: fullID accessModifier classParamClause classTemplateOpt
        | fullID classParamClause classTemplateOpt
        | fullID classTemplateOpt
        ;

enumDef: fullID accessModifier classParamClause enumTemplate
       | fullID classParamClause enumTemplate
       | fullID enumTemplate
       ;

classTemplateOpt: /* empty */ %prec LOW_PREC
                | EXTENDS classTemplate
                | templateBody
                ;

traitTemplateOpt: /* empty */ %prec LOW_PREC
                | EXTENDS traitTemplate
                | templateBody
                ;

enumTemplate: EXTENDS classParents enumBody
            | enumBody
            ;

classTemplate: classParents templateBody
             | classParents %prec END_TEMPLATE
             ;

classParents: stableId argumentExprs simpleTypes
          | stableId simpleTypes
          ;

traitTemplate: simpleType simpleTypes templateBody
             | simpleType simpleTypes %prec END_TEMPLATE
             ;

enumBody: nlo '{' enumStats '}' ;

enumStats: enumStat
       | enumStats semi enumStat
       ;

enumStat: templateStat
      | modifiers enumCase
      ;

enumCase: CASE fullID classParamClause EXTENDS classParents
      | CASE fullID classParamClause
      | CASE fullID %prec END_TEMPLATE
        | CASE ids ',' fullID
      ;

topStatSeq: topStat
```

```
        | topStatSeq semi topStat
          ;

topStat: modifiers tmplDef ;

simpleTypes: /* empty */
           | simpleTypes WITH simpleType
           ;

literal: DECIMAL_LITERAL
       | CHAR_LITERAL
       | DOUBLE_LITERAL
       | STRING_LITERAL
       | TRUE_LITERAL
       | FALSE_LITERAL
       | NULL_LITERAL
       ;

nls: /* empty */
   | nls NL
   ;

nlo: /* empty */
   | NL
   ;

semi: ';'
    | NL nlo
    ;

semio: /* empty */
     | semi
     ;

fullID: '+'
      | '!'
      | '#'
      | '%'
      | '&'
      | '*'
      | '-'
      | '/'
      | '<'
      | '>'
      | '?'
      | '@'
      | '\\'
      | '^'
      | '~'
      | PLUS_ASSIGNMENT
      | MINUS_ASSIGNMENT
      | MUL_ASSIGNMENT
      | DIV_ASSIGNMENT
```

```
      | MOD_ASSIGNMENT
      | EQUAL
      | NOT_EQUAL
      | GREATER_OR_EQUAL
      | LESS_OR_EQUAL
      | ID
      | ID_EQUALITY
      | ID_VERTICAL_SIGN
      | ID_AMPERSAND
      | ID_CIRCUMFLEX
      | ID_LESS
      | ID_GREAT
      | ID_MINUS
      | ID_PLUS
      | ID_ASTERISK
      | ID_SLASH
      | ID_PERCENT
      | ID_EXCLAMATION
      | ID_TILDE
      | ID_COLON
      ;

%%
```

6) Перечень таблиц, генерируемых на этапе семантического анализа

```
#ifndef SCALA_LEXER_TABLES_H
#define SCALA_LEXER_TABLES_H
#include <inttypes.h>
#include <list>
#include <optional>
#include <unordered_map>
#include <string>

#include "tables.hpp"
#include "nodes/generator/GeneratorNode.h"
#include "semantic/scopes/Scope.h"
#include "semantic/tools/datatype.h"
#include "semantic/tools/tools.h"
#include "semantic/tools/NameTransformer.h"

extern const std::string CONSTRUCTOR_NAME;
extern const std::string JVM_CONSTRUCTOR_NAME;
extern const std::string BASE_SCALA_CLASS;

class Scope;
class FunDefNode;
class DclNode;
class VarDefsNode;
class ClassDefNode;
class BlockStatsNode;
class ExprNode;
using namespace std;

class MetaInfo;
class BytesMetaInfo;
class Variable;
class FieldMetaInfo;
class LocalVarMetaInfo;
class MethodMetaInfo;
```

```
class ClassMetaInfo;
class SemanticContext;

class MetaInfo {
public:
   virtual ~MetaInfo();
};

class BytesMetaInfo : public MetaInfo {
public:
   BytesMetaInfo() = default;

   virtual bytearray_t toBytes() {
      return bytearray_t();
   }
};

class JvmDescriptorOwner {
public:
   virtual ~JvmDescriptorOwner() = default;

   virtual std::string jvmDescriptor() = 0;
};

class VarMetaInfo : public BytesMetaInfo {
public:
   string name;
   string jvmName;
   DataType dataType;
   ExprNode *value;
   bool isInit;
   bool isVal;

   VarMetaInfo(string name, const DataType &type, ExprNode *value, bool val)
      : name(name),
        jvmName(NameTransformer::encode(name)),
        dataType(type),
```

```
        value(value),
        isVal(val),
        isInit(false) {
    };

    VarMetaInfo() : isVal(true) {
       value = nullptr;
    }

    std::string toString();

    void setDataType(DataType type);

    bool operator==(const VarMetaInfo &);
};

class FieldMetaInfo : public VarMetaInfo {
public:
    Modifiers modifiers;
    ClassMetaInfo *classMetaInfo = nullptr;

    FieldMetaInfo() : modifiers({}) {
    };

    bool isPrivate() const { return modifiers.hasModifier(_PRIVATE); }
    bool isProtected() const { return modifiers.hasModifier(_PROTECTED); }
    bool isFinal() const { return modifiers.hasModifier(_FINAL); }
    bool isOverride() const { return modifiers.hasModifier(_OVERRIDE); }
};

class MethodVarMetaInfo : public VarMetaInfo {
public:
    MethodMetaInfo *methodMetaInfo = nullptr;
    uint16_t number = 0;

    MethodVarMetaInfo() = default;
};
```

```
class LocalVarMetaInfo : public MethodVarMetaInfo {
public:
    Scope *scope = nullptr;

    LocalVarMetaInfo();
};

class ArgMetaInfo : public MethodVarMetaInfo {
public:
    ArgMetaInfo() : MethodVarMetaInfo() {
    };
};

class MethodMetaInfo : public BytesMetaInfo, public JvmDescriptorOwner {
public:
    string name;
    string jvmName;
    ClassMetaInfo *classMetaInfo = nullptr;
    Modifiers modifiers;
    DataType returnType;
    ExprNode *body = nullptr;
    bool isPrimaryConstructor = false;
    vector<ArgMetaInfo *> args;

    unordered_map<string, unordered_map<unsigned int, LocalVarMetaInfo *>
> localVars;

    uint16_t localVarCounter = 0;

    optional<MethodVarMetaInfo *> resolveLocal(string varName, Scope
*scope);

    vector<DataType *> getArgsTypes();

    MethodMetaInfo() : modifiers({}) {};
```

```
    bool isPrivate() const { return modifiers.hasModifier(_PRIVATE); }
    bool isProtected() const { return modifiers.hasModifier(_PROTECTED); }
    bool isFinal() const { return modifiers.hasModifier(_FINAL); }
    bool isOverride() const { return modifiers.hasModifier(_OVERRIDE); }

    std::string jvmDescriptor() override;

    virtual optional<LocalVarMetaInfo *> addLocalVar(VarDefsNode
*varDefsNode, Scope *scope);
    virtual optional<LocalVarMetaInfo *> addGeneratorVar(GeneratorNode
*generatorNode, Scope *scope);
    virtual optional<LocalVarMetaInfo *> executeAssign(AssignmentNode
*assignNode, Scope* scope);
};

class ClassMetaInfo : public BytesMetaInfo, public JvmDescriptorOwner {
public:
    string name;
    string jvmName;
    Modifiers modifiers;
    ClassDefNode *classNode = nullptr;
    ClassMetaInfo *parent = nullptr;

    unordered_map<string, FieldMetaInfo *> fields;
    unordered_map<string, vector<MethodMetaInfo *> > methods;
    class ConstantPoolBuilder* constantPool = nullptr;

    ClassMetaInfo(string name, Modifiers modifiers, ClassDefNode *classNode)
: name(name),

jvmName(NameTransformer::encode(name)),
                                                modifiers(modifiers),
                                                classNode(classNode) {
    };
    ClassMetaInfo(string name, Modifiers modifiers) :
                                    name(name),
                                    jvmName(NameTransformer::encode(name)),
```

```
                                    modifiers(modifiers) {
    };

    virtual optional<FieldMetaInfo *> addField(VarDefsNode *varDefNode,
Modifiers modifiers);

    virtual optional<FieldMetaInfo *> addField(DclNode *varDclNode);

    virtual optional<MethodMetaInfo *> addMethod(FunDefNode *funDefNode,
Modifiers modifiers);

    virtual optional<MethodMetaInfo *> addMethod(DclNode *funDclNode);

    int subtypeDistance(const DataType* declared, const DataType* actual)
const;

    DataType* asDataType() {
        DataType* currentClassType = new DataType(DataType::Kind::Class);
        currentClassType->className = name;
        return currentClassType;
    };

    bool amSubclassOf(const ClassMetaInfo *other) const;

    virtual optional<FieldMetaInfo *> resolveField(
        const string &fieldName,
        const ClassMetaInfo *accessFrom,
        bool lookupPrivate = false
    );

    virtual optional<MethodMetaInfo *> resolveMethod(
        const string &methodName,
        const vector<DataType *> &argTypes,
        const ClassMetaInfo *accessFrom,
        int leftParents = PARENTS_CONSIDER,
        bool lookupPrivate = false,
        bool exactMatch = false
```

```
    );

    string jvmDescriptor() {
        return "Lvstu/scala/code" + this->name + ";";
    }

    optional<string> getParentName();

    uint16_t getConstantCounter();

    bool isPrivate() const { return modifiers.hasModifier(_PRIVATE); }
    bool isProtected() const { return modifiers.hasModifier(_PROTECTED); }
    bool isFinal() const { return modifiers.hasModifier(_FINAL); }
    bool isAbstract() const { return modifiers.hasModifier(_ABSTRACT); }
    bool isOverride() const { return modifiers.hasModifier(_OVERRIDE); }

    bool isRTL() const;

private:
    bool isSubtypeOrEqual(const DataType *declared, const DataType *actual);
};

class RtlClassMetaInfo : public ClassMetaInfo {
public:
    static RtlClassMetaInfo* Any;
    static RtlClassMetaInfo* String;
    static RtlClassMetaInfo* Integer;
    static RtlClassMetaInfo* StdIn;
    static RtlClassMetaInfo* Predef;
    static RtlClassMetaInfo* Unit;
    static RtlClassMetaInfo* Char;
    static RtlClassMetaInfo* Double;
    static RtlClassMetaInfo* Boolean;
    static RtlClassMetaInfo* Iterator;
    static RtlClassMetaInfo* Array;

    std::string javaDescriptor = "java/lang/";
```

```
    std::string scalaDescriptor = "Lscala/runtime/";

    enum Lang {
        _SCALA,
        _JAVA
    };

    Lang langOfDescriptor;

    string jvmDescriptor() {
        return langOfDescriptor == _SCALA ? scalaDescriptor : javaDescriptor
                + this->name + ";";
    }

    RtlClassMetaInfo(std::string name, Lang langOfDescriptor):
ClassMetaInfo(name, Modifiers({})) {
        this->langOfDescriptor = langOfDescriptor;
    }

    static RtlClassMetaInfo* getRtlClassInfo(const std::string& typeName);

public:
    static void initializeRtlClasses();

private:
    static RtlClassMetaInfo* initAny();
    static RtlClassMetaInfo* initString();
    static RtlClassMetaInfo* initInteger();
    static RtlClassMetaInfo* initStdIn();
    static RtlClassMetaInfo* initPredef();
    static RtlClassMetaInfo* initUnit();
    static RtlClassMetaInfo* initChar();
    static RtlClassMetaInfo* initDouble();
    static RtlClassMetaInfo* initBoolean();
    static RtlClassMetaInfo* initIterator();
    static RtlClassMetaInfo* initArray();
};
```

```
#endif //SCALA_LEXER_TABLES_H
```

7) Перечень ошибок, определяемых на этапе семантического анализа

1. Пустое синтаксическое дерево (программа не содержит кода)
2. Повторное объявление класса
3. Повторное объявление метода
4. Повторное объявление конструктора
5. Повторное объявление поля класса
6. Одинаковые имена параметров метода
7. Наследование от несуществующего класса
8. Наследование от финального класса
9. Наследование от приватного класса
10. Некорректное наследование
11. Попытка переопределения без наличия суперкласса
12. Отсутствие элемента для переопределения в супер классах
13. Попытка переопределить финальный элемент
14. Элемент должен быть переопределен, но override отсутствует
15. Ослабление модификатора доступа при переопределении
16. Несовместимые типы при переопределении элемента
17. Нарушение согласованности (consistency) при переопределении
18. Переопределение метода с несовместимой/несуществующей сигнатурой
19. Недопустимый модификатор у функции
20. Недопустимый модификатор у поля
21. Несовместимые модификаторы
22. Необъявленный тип
23. Несоответствие типов при присваивании
24. Несоответствие типов возвращаемого значения
25. Возврат значения из void-функции
26. Отсутствие возвращаемого значения в функции
27. Несовместимые типы при объявлении переменной
28. Использование необъявленной переменной
29. Использование неинициализированной переменной
30. Повторное присваивание val-переменной
31. Недопустимое присваивание
32. Некорректное присваивание массива

33. Неопределенный тип элементов массива
34. Разные типы элементов при создании массива
35. Несовместимые типы массива и его элементов
36. Индекс массива не имеет целочисленного типа
37. Использование нескольких переменных в for
38. Цикл for не по массиву или генератору
39. Пустой диапазон
40. Недопустимый тип диапазона
41. Некорректные границы диапазона
42. Некорректное значение шага диапазона
43. Несовместимый тип итератора и элементов диапазона
44. Условие не имеет логического типа
45. Метод не найден в классе
46. Класс не найден при вызове метода
47. Поле не инициализировано.
48. Переопределение поля с модификатором final
49. Переопределение метода с модификатором final
50. Абстрактное поле не переопределено
51. Абстрактный метод не переопределен
52. Main класс не найден
53. Main метод не найден
54. Несколько Main классов
55. Несколько main методов внутри Main класса
56. Конструктор содержит некорректную инструкцию (к примеру, return)

8) Перечень преобразований дерева и дополнительной информации для узлов дерева на этапе семантического анализа

- В случае отсутствия класса-родителя, назначать родителем класс Any;
- Создание и размещение в каждом классе объектов для работы с консолью (ввод/вывод);
- Перемещение переменных из первичных конструкторов в тело класса в качестве полей;

- Создание базового конструктора (инициализация полей класса + вызов конструктора родительского класса с необходимыми параметрами);
- Преобразование узла конструктора к узлу обычного метода;
- Преобразование пустого return к return new Unit();
- Нормализация правоассоциативных операторов внутри узла инфиксных выражений;
- Преобразование инфиксных выражений к вызову методов (пример: замена a + b на a.plus(b));
- Преобразование литералов к созданию объектов RTL (пример: 1 -> new Int(1)).

9) Выводы

В ходе выполнения задания мы разобрались во внутренней архитектуре компилятора и его основных этапах работы. Особое внимание было уделено лексическому и синтаксическому анализу, в рамках которых исходный текст программы преобразуется в структурированное синтаксическое дерево.

Этапы семантического анализа и генерации кода позволили глубже понять принципы работы со сложными структурами данных, в частности с деревьями. Кроме того, генерация кода помогла осмыслить процесс формирования байт-кода и его взаимодействие с исполнительной средой.

Таким образом, изучение процесса компиляции расширило наше понимание внутренней логики языков программирования и укрепило навыки анализа и обработки программного кода. Последовательно проходя все стадии, мы получили более ясное представление о том, как текст программы, состоящий из лексем, преобразуется в исполняемый код, понятный компьютеру.