МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НИЖЕГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ИМ. Р.Е. АЛЕКСЕЕВА»

Институт радиоэлектроники и информационных технологий

Кафедра «Прикладная математика»

Лабораторная работа №1

Выполнил:

Студент группы 23-ПМ-2

Корноухов Илья

Нижний Новгород

2024

**Содержание**

[**Содержание** 2](#_Toc74168366)

[**Постановка задачи** 3](#_Toc74168367)

[**Теоретический материал** 4](#_Toc74168368)

[**Структура программы** 5](#_Toc74168368)

[**Пример работы программы** 7](#_Toc74168368)

[**Исходный код** 8](#_Toc74168368)

[**Заключение** 17](#_Toc74168368)

Постановка задачи

1. Написать программу на любом языке программирования
2. Считать грамматику из консоли или файла
3. Построить на основе полученной грамматики конечный автомат
4. Преобразовать конечный автомат к детерминированному конечному автомату
5. Считать произвольную строку и проверить ее на принадлежность к грамматике

Теоретический материал

Грамматика – кортеж вида <N, T, S, R>, где

1. N – множество нетерминалов
2. T – множество терминалов
3. S – начальный символ грамматики
4. R – правила перехода

Автоматная грамматика – грамматика, все правила которой имеют вид:

A -> aB или A -> a для праволинейной грамматики, где A, B – нетерминалы, a - терминал

A -> Ba или A -> a для леволинейной грамматики, где A, B – нетерминалы, a – терминал

Конечный автомат – кортеж вида <Q, A, D, S, E>, где

1. Q – множество состояний автомата
2. A – алфавит
3. D – множество переходов
4. S – множество начальных состояний
5. E – множество конечных состояний

Детерминированный конечный автомат (ДКА) – конечный автомат, переходы которого однозначны и имеют вид <q0, a> -> q1, где q0, q1 – состояния автомата, a – символ алфавита

Структура программы

Программа написана на языке программирования Kotlin с использованием системы сборки Gradle.

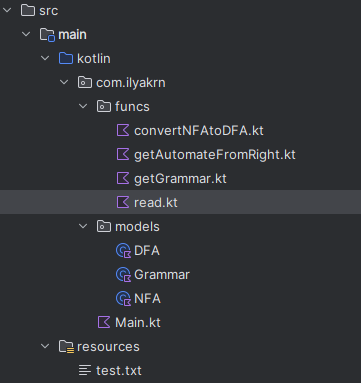


Рисунок 1

На рисунке 1 представлена структура программы.

В пакете funcs находятся файлы, содержащие функции для чтения грамматики, построения конечного автомата по грамматике, преобразования КА в ДКА, проверки строки на принадлежность к грамматике.

В пакете models находятся файлы с моделями ДКА, КА, Грамматики.

В файле Main.kt находится функция main для запуска программы.

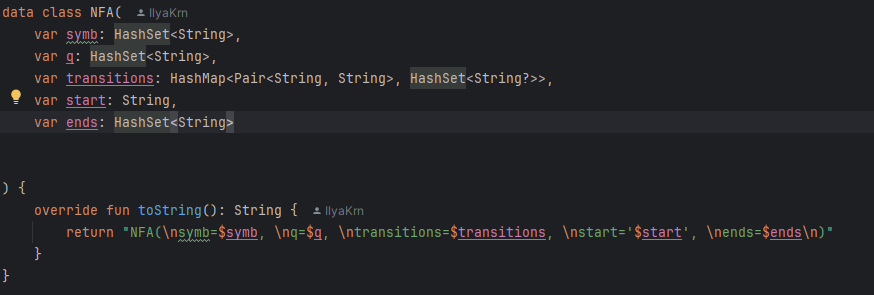


Рисунок 2

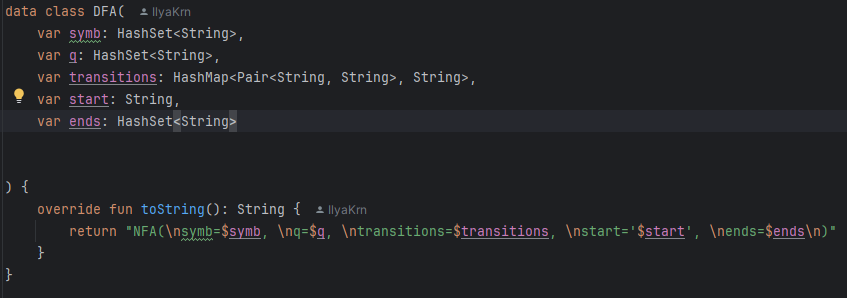


Рисунок 3

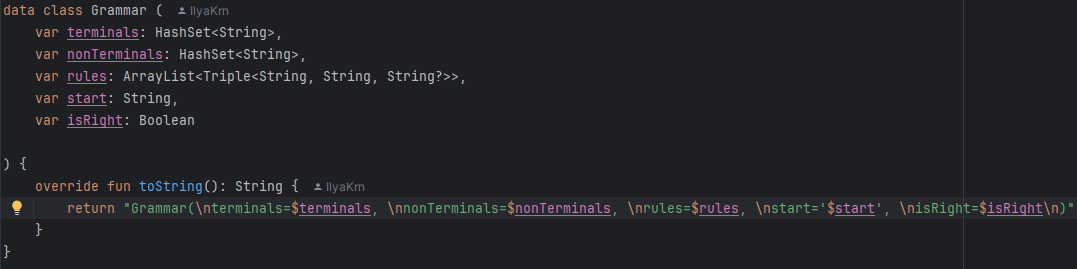


Рисунок 4

На рисунках 2-4 представлены классы КА, ДКА и грамматики соответственно

Пример работы программы

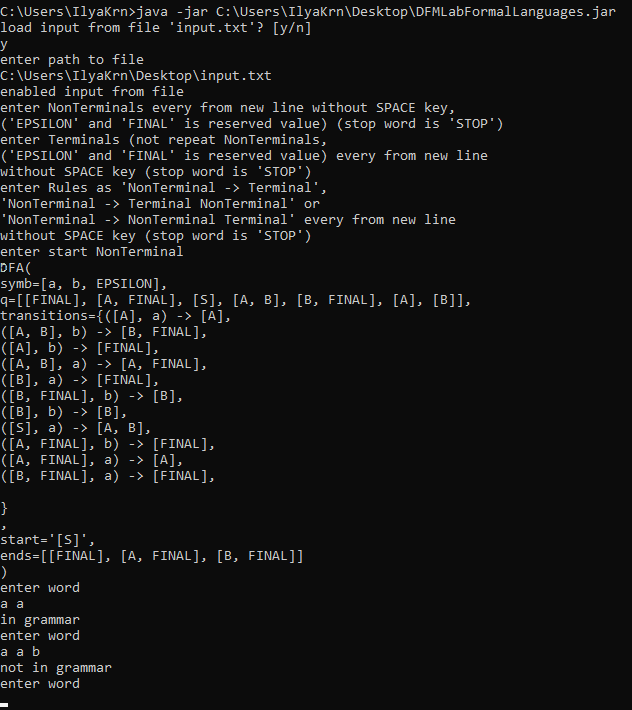


Рисунок 5

На рисунке 5 представлена работа программы: чтение грамматики из файла, вывод ДКА, построенного по введенной грамматике и проверка слов «aa» и «aab» на принадлежность к грамматике

Исходный код

Main.kt:

package com.ilyakrn  
  
import com.ilyakrn.funcs.convertNFAtoDFA  
import com.ilyakrn.funcs.getAutomateFromRight  
import com.ilyakrn.funcs.getGrammar  
import com.ilyakrn.funcs.read  
import java.io.File  
import java.io.FileInputStream  
import java.util.\*  
import kotlin.collections.ArrayList  
  
fun main() {  
  
 val scanner = Scanner(System.`in`)  
 var inputStream = System.`in`  
 println("load input from file 'input.txt'? [y/n]")  
 val ans = scanner.nextLine()  
 when(ans){  
 "y" -> {  
 println("enter path to file")  
 val file = File(scanner.nextLine())  
 if (!file.exists()){  
 println("file does not exist")  
 return  
 }  
 println("enabled input from file")  
 inputStream = FileInputStream(file)  
 }  
 "n" -> {  
 println("enabled input from terminal")  
 }  
 else -> {  
 println("incorrect answer, enabled input from terminal")  
 }  
 }  
  
  
  
 val g = getGrammar(inputStream, System.out)  
 if (g.isRight) {  
 val a = getAutomateFromRight(g)  
 val da = convertNFAtoDFA(a)  
 println(da)  
 while (true) {  
 println("enter word")  
 val input = ArrayList<String>()  
 while (true) {  
 val inp = scanner.nextLine().split(" ")  
 var fl = false  
 for (i in inp) {  
 if (i.isEmpty()) {  
 println("try again")  
 fl = true  
 continue  
 }  
 }  
 if (fl) {  
 continue  
 }  
 input.addAll(inp)  
 break  
 }  
 read(input, da)  
 }  
 }  
 else {  
 println("grammar is left!")  
 }  
  
  
  
}

NFA.kt:

package com.ilyakrn.models  
  
data class NFA(  
 var symb: HashSet<String>,  
 var q: HashSet<String>,  
 var transitions: HashMap<Pair<String, String>, HashSet<String?>>,  
 var start: String,  
 var ends: HashSet<String>  
  
  
) {  
 override fun toString(): String {  
 return "NFA(\nsymb=$symb, \nq=$q, \ntransitions=$transitions, \nstart='$start', \nends=$ends\n)"  
 }  
}

DFA.kt:

package com.ilyakrn.models  
  
data class DFA(  
 var symb: HashSet<String>,  
 var q: HashSet<String>,  
 var transitions: HashMap<Pair<String, String>, String>,  
 var start: String,  
 var ends: HashSet<String>  
  
  
) {  
 override fun toString(): String {  
 var transitionsString = "{"  
 transitions.forEach {  
 transitionsString += "${it.key} -> ${it.value},\n"  
 }  
 transitionsString += "\n}\n"  
 return "DFA(\nsymb=$symb, \nq=$q, \ntransitions=$transitionsString, \nstart='$start', \nends=$ends\n)"  
 }  
}

Grammar.kt:

package com.ilyakrn.models;  
  
data class Grammar (  
 var terminals: HashSet<String>,  
 var nonTerminals: HashSet<String>,  
 var rules: ArrayList<Triple<String, String, String?>>,  
 var start: String,  
 var isRight: Boolean  
  
) {  
 override fun toString(): String {  
 return "Grammar(\nterminals=$terminals, \nnonTerminals=$nonTerminals, \nrules=$rules, \nstart='$start', \nisRight=$isRight\n)"  
 }  
}

read.kt:

package com.ilyakrn.funcs  
  
import com.ilyakrn.models.DFA  
  
fun read(str: ArrayList<String>, a: DFA) {  
 var state = a.start  
 for (i in str) {  
 if (a.transitions[Pair(state, i)] != null) {  
 state = a.transitions[Pair(state, i)]!!  
 if (a.ends.contains(state))  
 state = a.start  
 }  
 else{  
 println("not in grammar")  
 return  
 }  
 }  
 println("in grammar")  
}

getGrammar.kt:

package com.ilyakrn.funcs  
  
import com.ilyakrn.models.Grammar  
import java.io.InputStream  
import java.io.PrintStream  
import java.util.\*  
import kotlin.collections.ArrayList  
import kotlin.collections.HashMap  
import kotlin.collections.HashSet  
  
  
fun getGrammar(stream: InputStream, out: PrintStream?): Grammar {  
 // Scanner of input stream  
 val scanner = Scanner(stream)  
 // initial values of grammar  
 val terminals = HashSet<String>()  
 val nonTerminals = HashSet<String>()  
 val rules = ArrayList<Triple<String, String, String?>>()  
 var start = ""  
 var linearly = 0 // 0 none; 1 right; -1 left;  
  
  
 // reading NonTerminals  
 out?.println("enter NonTerminals every from new line without SPACE key,\n('EPSILON' and 'FINAL' is reserved value) (stop word is 'STOP')")  
 var inputLine = scanner.nextLine()  
 while (inputLine != "STOP") {  
 if (!inputLine.contains(" ") && !nonTerminals.contains(inputLine) && inputLine != "EPSILON" && inputLine != "FINAL")  
 nonTerminals.add(inputLine)  
 else  
 out?.println("wrong NonTerminal $inputLine skipped, try again")  
 inputLine = scanner.nextLine()  
 }  
 // checking empty input  
 if (nonTerminals.size == 0)  
 throw RuntimeException("NonTerminals not found")  
  
  
 // reading Terminals  
 out?.println("enter Terminals (not repeat NonTerminals,\n('EPSILON' and 'FINAL' is reserved value) every from new line\nwithout SPACE key (stop word is 'STOP')")  
 inputLine = scanner.nextLine()  
 while (inputLine != "STOP") {  
 if (!inputLine.contains(" ") && !nonTerminals.contains(inputLine) && !terminals.contains(inputLine) && inputLine != "EPSILON" && inputLine != "FINAL")  
 terminals.add(inputLine)  
 else  
 out?.println("wrong Terminal $inputLine skipped, try again")  
 inputLine = scanner.nextLine()  
 }  
 // checking empty input  
 if (terminals.size == 0)  
 throw RuntimeException("NonTerminals not found")  
 // empty word  
 terminals.add("EPSILON")  
  
  
 // reading Rules  
 out?.println("enter Rules as 'NonTerminal -> Terminal',\n'NonTerminal -> Terminal NonTerminal' or\n'NonTerminal -> NonTerminal Terminal' every from new line\nwithout SPACE key (stop word is 'STOP')")  
 inputLine = scanner.nextLine()  
 while (inputLine != "STOP") {  
 val lineSep = inputLine.split(" ")  
 if (lineSep.size == 3){  
 if (nonTerminals.contains(lineSep[0]) && terminals.contains(lineSep[2])) {  
 rules.add(Triple(lineSep[0], lineSep[2], null))  
 inputLine = scanner.nextLine()  
 continue  
 }  
 out?.println("wrong Rule $inputLine skipped, try again")  
 inputLine = scanner.nextLine()  
 continue  
 }  
 else if (lineSep.size == 4){  
 // right grammar  
 if (nonTerminals.contains(lineSep[0]) && terminals.contains(lineSep[2]) && nonTerminals.contains(lineSep[3])){  
 if (linearly == -1) {  
 out?.println("wrong Rule $inputLine skipped (different linearly), try again")  
 inputLine = scanner.nextLine()  
 continue  
 }  
 rules.add(Triple(lineSep[0], lineSep[2], lineSep[3]))  
 linearly = 1  
 }  
 // left grammar  
 else if (nonTerminals.contains(lineSep[0]) && nonTerminals.contains(lineSep[2]) && terminals.contains(lineSep[3])){  
 if (linearly == 1) {  
 out?.println("wrong Rule $inputLine skipped (different linearly), try again")  
 inputLine = scanner.nextLine()  
 continue  
 }  
 rules.add(Triple(lineSep[0], lineSep[2], lineSep[3]))  
 linearly = -1  
 }  
 else  
 out?.println("wrong Rule $inputLine skipped, try again")  
 }  
 else  
 out?.println("wrong Rule $inputLine skipped, try again")  
 inputLine = scanner.nextLine()  
 }  
 // checking empty input  
 if (rules.size == 0)  
 throw RuntimeException("Rules not found")  
  
  
 // reading starting NonTerminal  
 out?.println("enter start NonTerminal")  
 inputLine = scanner.nextLine()  
 while (true) {  
 if (nonTerminals.contains(inputLine))  
 start = inputLine  
 if (start != "")  
 break  
 out?.println("wrong start NonTerminal $inputLine skipped, try again")  
 if (scanner.hasNext())  
 inputLine = scanner.nextLine()  
 }  
  
 if (nonTerminals.size == 0 || terminals.size == 0 || rules.size == 0)  
 throw RuntimeException("Input is empty")  
  
 return Grammar(terminals, nonTerminals, rules, start, linearly == 1)  
}

getAutomateFromRight.kt:

package com.ilyakrn.funcs  
  
import com.ilyakrn.models.NFA  
import com.ilyakrn.models.Grammar  
import kotlin.collections.ArrayList  
  
fun getAutomateFromRight(grammar: Grammar): NFA {  
 // check grammar linearly  
 if (!grammar.isRight)  
 throw RuntimeException("grammar is not right")  
 grammar.nonTerminals.add("FINAL")  
  
 // rules with FINAL state  
 val newRules = ArrayList<Triple<String, String, String>>()  
 for (r in grammar.rules) {  
 if (r.third == null){  
 var fl = false  
 for (i in grammar.rules){  
 if (i.third != null)  
 fl = true  
 }  
 if (fl){  
 newRules.add(Triple(r.first, r.second, "FINAL"))  
 }  
 }  
 }  
 grammar.rules.addAll(newRules)  
  
 // automate transitions  
 var transitions = HashMap<Pair<String, String>, HashSet<String?>>()  
 for (r in grammar.rules){  
 if (r.third != null){  
 if (transitions[Pair(r.first, r.second)] == null)  
 transitions[Pair(r.first, r.second)] = HashSet()  
 transitions[Pair(r.first, r.second)]!!.add(r.third)  
 }  
 }  
 // end states  
 val ends = HashSet<String>()  
 for (r in grammar.rules){  
 if (r.third == null){  
 for (i in grammar.rules){  
 if (i.first == r.first && i.second == r.second && i.third != null)  
 ends.add(i.third!!)  
 }  
 }  
 }  
 // adding start to end states  
 if (grammar.rules.contains(Triple(grammar.start, "EPSILON", null)))  
 ends.add(grammar.start)  
 return NFA(grammar.terminals, grammar.nonTerminals, transitions, grammar.start, ends)  
  
}

convertNFAtoDFA.kt:

package com.ilyakrn.funcs  
  
import com.ilyakrn.models.DFA  
import com.ilyakrn.models.NFA  
  
fun convertNFAtoDFA(nfa: NFA): DFA {  
 val dfaStates = HashSet<HashSet<String>>() // Новые состояния DFA  
 val dfaTransitions = HashMap<Pair<HashSet<String>, String>, HashSet<String>>() // Переходы DFA  
 val dfaEnds = HashSet<HashSet<String>>() // Конечные состояния DFA  
  
 // Start in DFA  
 val startState = HashSet<String>()  
 startState.add(nfa.start)  
 val queue = ArrayDeque<HashSet<String>>()  
 queue.add(startState)  
 dfaStates.add(startState)  
  
 while (queue.isNotEmpty()) {  
 val currentState = queue.removeFirst()  
  
 for (symbol in nfa.symb) {  
 val newState = HashSet<String>()  
  
 // Set of states from current state  
 for (state in currentState) {  
 val transitionKey = Pair(state, symbol)  
 if (nfa.transitions.containsKey(transitionKey)) {  
 for (ii in nfa.transitions[transitionKey]!!){  
 newState.add(ii!!)  
 }  
 }  
 }  
  
 if (newState.isNotEmpty()) {  
 dfaTransitions[Pair(currentState, symbol)] = newState  
  
 if (!dfaStates.contains(newState)) {  
 dfaStates.add(newState)  
 queue.add(newState)  
 }  
  
 if (newState.any { nfa.ends.contains(it) }) {  
 dfaEnds.add(newState)  
 }  
 }  
 }  
 }  
  
 val dfaStatesForm = HashSet<String>()  
 dfaStates.forEach { dfaStatesForm.add(it.toString()) }  
 val dfaTransitionsForm = HashMap<Pair<String, String>, String>()  
 dfaTransitions.forEach { dfaTransitionsForm[Pair(it.key.first.toString(), it.key.second)] = it.value.toString() }  
 val dfaEndsForm = HashSet<String>()  
 dfaEnds.forEach { dfaEndsForm.add(it.toString()) }  
  
 return DFA(  
 symb = nfa.symb,  
 q = dfaStatesForm,  
 transitions = dfaTransitionsForm,  
 start = startState.toString(),  
 ends = dfaEndsForm  
 )  
}

Заключение

В ходе выполнения лабораторной работы был изучен необходимый материал и написана программа, считывающая грамматику и строящая на ее основе детерминированный конечный автомат.