Лабораторна робота №3, ІП-31, Кахерський О.І.

**Построение конечного автомата по регулярной грамматике**

**Цель:** закрепить понятия «регулярная грамматика», «недетерминированный и детерминированный конечный автомат», сформировать умения и навыки построения конечного автомата по регулярной грамматике и преобразования недетерминированного конечного автомата к детерминированному конечному автомату.

Вариант – 8.

G = ({S,A,B,C,D},{a,b,c,d,~},{S=>aA|bB; A=>cC|~; C=>cC|cA; B=>dD|~; D=>dD|dB;},S)

**Результат работы программы:**

grammarParts = [{S,A,B,C,D}, {a,b,c,d,~}, {S=>aA|bB; A=>cC|~; C=>cC|cA; B=>dD|~; D=>dD|dB;}, S]

grammar = RegularGrammar{terminals={a=a, b=b, c=c, d=d, ~=~}, nonTerminals={A=A, B=B, S=S, C=C, D=D}, transitionRules={A=[cC, ~], B=[dD, ~], S=[aA, bB], C=[cC, cA], D=[dD, dB]}, startSymbol=S}

Множество вершин: [A, B, S, C, D, Z];

Входной алфавит: [a, b, c, d, ~];

Множество команд:

F(A,c)=C

F(A,~)=Z

F(B,d)=D

F(B,~)=Z

F(S,b)=B

F(S,a)=A

F(C,c)=A

F(C,c)=C

F(D,d)=B

F(D,d)=D;

Множество начальных состояний: [S];

Множество конечных состояний: [Z]

DkaModel{states=[[A], [B], [S], [C], [D], [A, C], [B, D], [Z]],

alphabet={a=a, b=b, c=c, d=d, ~=~},

transitions={[A]={c=[C], ~=[Z]}, [B]={d=[D], ~=[Z]}, [S]={a=[A], b=[B]}, [C]={c=[A, C]}, [D]={d=[B, D]}, [A, C]={c=[A, C], ~=[Z]}, [B, D]={d=[B, D], ~=[Z]}},

endStates=[[Z]],

startStates=[S]}

Листинг кода

**import** grammar.NkaBuilder;  
**import** grammar.NkaModel;  
**import** grammar.RegularGrammar;  
  
**import** java.io.BufferedReader;  
**import** java.io.FileNotFoundException;  
**import** java.io.FileReader;  
**import** java.util.Arrays;  
  
*/\*\*  
 \** ***@author*** *Oleh Kakherskyi (olehkakherskiy@gmail.com)  
 \*/***public class** Main {  
  
 **public static void** main(String[] args) {  
 RegularGrammarValidator validator = **new** RegularGrammarValidator();  
 RegularGrammar grammar = **null**;  
 **try** {  
 AbstractGrammarLoader grammarLoader = **new** FileGrammarLoader(**new** BufferedReader(**new** FileReader(**"D:\\Документы\\FICT\\6\_семестр\\TOLSA\\lab3\\grammar.txt"**)));  
 String[] grammarParts = grammarLoader.getRegularGrammar();  
 System.***out***.println(**"grammarParts = "** + Arrays.*toString*(grammarParts));  
 grammar = validator.validateAndBuildRegularGrammar(grammarParts);  
 } **catch** (FileNotFoundException e) {  
 e.printStackTrace();  
 }  
*// String regGrammar = "{S,A,B,C,D}\n{a,b,c,d,~}\n{S=>aA|bB; A=>cC|~; C=>cC|cA; B=>dD|~; D=>dD|dB;}\nS";  
// String regGrammar = "{K,L,M,N,P}\n{0,1,&,%,a,b}\n{K=>1M|ε; M=>0L|&N|&P; L=>1L|0L|%P; N=>aN|bN|%P; P=>1P|aP|0}\nK";  
// String regGrammar = "{S,A,B}\n{a,b}\n{S=>aB|aA; B=>bB|a; A=>aA|b}\nS";  
// System.out.println("regGrammar.split() = " + Arrays.toString(regGrammar.split("\n")));  
// RegularGrammar grammar = validator.validateAndBuildRegularGrammar(regGrammar.split("\n"));* System.***out***.println(**"grammar = "** + grammar.toString());  
 System.***out***.println();  
 System.***out***.println();  
 NkaBuilder builder = **new** NkaBuilder();  
 NkaModel machine = builder.buildNonDeterministicFiniteStateMachine(grammar);  
 System.***out***.println(machine.toString());  
 DkaBuilder dkaBuilder = **new** DkaBuilder();  
 System.***out***.println(dkaBuilder.buildDkaFromNka(machine));  
 }  
}

**import** grammar.RegularGrammar;  
**import** grammar.RegularGrammarBuilder;  
  
**import** java.util.Set;  
**import** java.util.regex.Pattern;  
  
*/\*\*  
 \** ***@author*** *Oleh Kakherskyi (olehkakherskiy@gmail.com)  
 \*/***public class** RegularGrammarValidator {  
  
 **private static final** Pattern ***terminalsValidationRule*** = Pattern.*compile*(**"^\\{([a-z0-9!@#$%^&\*~ε], \*)\*[a-z0-9!@#$%^&\*~ε]?\\}$"**);  
  
 **private static final** Pattern ***nonTerminalsValidationRule*** = Pattern.*compile*(**"^\\{([A-Z], \*)\*[A-Z]?\\}$"**);  
  
 *//"^([A-Z]=>(([a-z][A-Z]|[a-z])\\|?)+;? \*)+$"* **private static final** String ***transitionRuleValidationRule*** = **"^\\{(%s=>((%s%s|%s)\\|?)+;? \*)+}$"**;  
  
 **public** RegularGrammar validateAndBuildRegularGrammar(String[] regularGrammarString) {  
 RegularGrammarBuilder builder = **new** RegularGrammarBuilder();  
  
 validateGrammarStringArray(regularGrammarString);  
  
 validateNonTerminals(regularGrammarString[0], builder);  
  
 validateTerminals(regularGrammarString[1], builder);  
  
 validateTransitionRules(regularGrammarString[2], builder);  
  
 validateStartSymbol(regularGrammarString[3], builder);  
  
 **return** builder.getResultGrammar();  
 }  
  
 **private void** validateTerminals(String terminalsString, RegularGrammarBuilder grammarBuilder) {  
 **if** (!***terminalsValidationRule***.matcher(terminalsString).matches()) {  
 **throw new** IllegalArgumentException(**"Множество терминальных символов не прошло валидацию. "** +  
 **"Проверьте правильность формата: {терм\_символ(маленькая лат. буква),терм\_символ...}"**);  
  
 } **else** {  
 grammarBuilder.addTerminals(terminalsString.trim());  
 }  
 }  
  
 **private void** validateNonTerminals(String nonTerminalString, RegularGrammarBuilder grammarBuilder) {  
 **if** (!***nonTerminalsValidationRule***.matcher(nonTerminalString).matches()) {  
 **throw new** IllegalArgumentException(**"Множество нетерминальных символов не прошло валидацию. "** +  
 **"Проверьте правильность формата: {неТерм\_символ(большая лат. буква),неТерм\_символ...}"**);  
  
 } **else** {  
 grammarBuilder.addNonTerminals(nonTerminalString);  
 }  
 }  
  
 **private void** validateTransitionRules(String transitionRules, RegularGrammarBuilder grammarBuilder) {  
 RegularGrammar grammar = grammarBuilder.getResultGrammar();  
 String nonTerminalsRegexpClass = buildTerminalOrNonTerminalRegexpClass(grammar.getNonTerminals());  
 String terminalsRegexpClass = buildTerminalOrNonTerminalRegexpClass(grammar.getTerminals());  
  
 String transitionRulesRegexp = String.*format*(***transitionRuleValidationRule***, nonTerminalsRegexpClass,  
 terminalsRegexpClass, nonTerminalsRegexpClass, terminalsRegexpClass);  
  
 **if** (!Pattern.*matches*(transitionRulesRegexp, transitionRules)) {  
 **throw new** IllegalArgumentException(**"Правила грамматики не соответствуют регулярному "** +  
 **"выражению: "** + transitionRulesRegexp);  
 }  
 grammarBuilder.addTransitionRules(transitionRules);  
 }  
  
 **private void** validateStartSymbol(String startSymbol, RegularGrammarBuilder builder) {  
 **if** (startSymbol.length() > 1) {  
 **throw new** IllegalArgumentException(**"Начальный символ грамматики должен состоять из 1 буквы. "** +  
 **"Полученный результат: "** + startSymbol);  
 }  
  
 String regexp = buildTerminalOrNonTerminalRegexpClass(builder.getResultGrammar().getNonTerminals());  
 **if** (!Pattern.*matches*(regexp, startSymbol)) {  
 **throw new** IllegalArgumentException(**"Начальный символ не входит в множество нетерминальных символов.\n"** +  
 **"Множество нетерминальных символов: "** + regexp + **"\nНачальный символ: "** + startSymbol);  
 }  
 builder.addStartSymbol(startSymbol);  
 }  
  
 **private** String buildTerminalOrNonTerminalRegexpClass(Set<Character> grammarSymbols) {  
 StringBuilder regexpClassBuilder = **new** StringBuilder(**"["**);  
 **for** (Character key : grammarSymbols)  
 regexpClassBuilder.append(key);  
 **return** regexpClassBuilder.append(**"ε]"**).toString();  
 }  
  
 **private void** validateGrammarStringArray(String[] regularGrammarString) {  
 **if** (regularGrammarString == **null** || regularGrammarString.**length** < 4) {  
 **throw new** IllegalArgumentException(**"Количество элементов грамматики должно быть >=4"**);  
 }  
 **for** (**int** i = 0; i < regularGrammarString.**length**; i++) {  
 **if** (regularGrammarString[i] == **null**)  
 **throw new** IllegalArgumentException((i + 1) + **" элемент грамматики == null"**);  
  
 regularGrammarString[i] = regularGrammarString[i].trim();  
 **if** (regularGrammarString[i].equals(**""**))  
 **throw new** IllegalArgumentException((i + 1) + **" элемент грамматики - пустой"**);  
 }  
 }  
  
}

**package** grammar;  
  
**import** java.util.HashMap;  
**import** java.util.HashSet;  
**import** java.util.Map;  
**import** java.util.Set;  
  
*/\*\*  
 \** ***@author*** *Oleh Kakherskyi (olehkakherskiy@gmail.com)  
 \*/***public class** RegularGrammar {  
  
 **private** Set<Character> **terminals** = **new** HashSet<>();  
  
 **private** Set<Character> **nonTerminals** = **new** HashSet<>();  
  
 **private** Map<Character, Set<String>> **transitionRules** = **new** HashMap<>();  
  
 **private char startSymbol**;  
  
 **public** Set<String> getTransitionRules(Character key) {  
 **return transitionRules**.get(key);  
 }  
  
 **boolean** addTransitionRule(**char** key, String rule) {  
 **if** (rule == **null** || rule.equals(**""**))  
 **return false**;  
 **if** (!**transitionRules**.containsKey(key)) { *//у данного нетерминального символа пока нет правил* Set<String> keyRules = **new** HashSet<>();  
 keyRules.add(rule);  
 **transitionRules**.put(key, keyRules);  
 **return true**; *// 100% что правило добавиться в пустое множество правил* } **else** { *//у данного нетерминального символа уже есть правила.* **return transitionRules**.get(key).add(rule); *//false, если правило уже есть* }  
 }  
  
  
 **public** Map<Character, Set<String>> getTransitionRules() {  
 **return transitionRules**;  
 }  
  
 **public char** getStartSymbol() {  
 **return startSymbol**;  
 }  
  
 **boolean** setStartSymbol(**char** startSymbol) {  
 **if** (**nonTerminals**.contains(startSymbol)) {  
 **this**.**startSymbol** = startSymbol;  
 **return true**;  
 } **else return false**;  
 }  
  
 **public** Set<Character> getTerminals() {  
 **return terminals**;  
 }  
  
 **public void** setTerminals(Set<Character> terminals) {  
 **this**.**terminals** = terminals;  
 }  
  
 **public** Set<Character> getNonTerminals() {  
 **return nonTerminals**;  
 }  
  
 **public void** setNonTerminals(Set<Character> nonTerminals) {  
 **this**.**nonTerminals** = nonTerminals;  
 }  
  
 @Override  
 **public** String toString() {  
 **return "RegularGrammar{"** +  
 **"terminals="** + **terminals** +  
 **", nonTerminals="** + **nonTerminals** +  
 **", transitionRules="** + **transitionRules** +  
 **", startSymbol="** + **startSymbol** +  
 **'}'**;  
 }  
}

**package** grammar;  
  
**import** java.util.HashSet;  
**import** java.util.Set;  
  
*/\*\*  
 \** ***@author*** *Oleh Kakherskyi (olehkakherskiy@gmail.com)  
 \*/***public class** RegularGrammarBuilder {  
  
 **private static** String *transitionRuleSplitter* = **"=>"**;  
  
 **private** RegularGrammar **grammar**;  
  
 **public** RegularGrammarBuilder() {  
 **grammar** = **new** RegularGrammar();  
 }  
  
 **public void** addTerminals(String terminalString) { *//параметр в формате {termSymbol1,termSymbol2,termSymbol3}* **grammar**.setTerminals(buildGrammarSymbolsSet(terminalString));  
 }  
  
 **public void** addNonTerminals(String nonTerminalString) { *//параметр в формате {NonTermSymbol1,NonTermSymbol2,NonTermSymbol3}* **grammar**.setNonTerminals(buildGrammarSymbolsSet(nonTerminalString));  
 }  
  
 *//разбивает строку правил на правила. Разграничитель - символ <;>. Последнее правило может быть пустым - игнорируется* **public void** addTransitionRules(String rulesRepresentation) {  
 *//удаляем символы фигурных скобок и разбиваем на правила по символу <;>* String[] rules = rulesRepresentation.substring(1, rulesRepresentation.length() - 1).split(**";"**);  
 **int** lastRuleIndex = rules[rules.**length** - 1].equals(**""**) ? rules.**length** - 1 : rules.**length**;  
 **for** (**int** i = 0; i < lastRuleIndex; i++) {  
 **if** (!addTransitionRule(rules[i].trim()))  
 **throw new** IllegalArgumentException(**"Правило "** + rules[i].trim() + **"продублировано"**);  
 }  
 }  
  
 **private boolean** addTransitionRule(String ruleRepresentation) { *//ruleRepresentation формат: NonTermSymbol=>Rule  
  
 //1 часть - нетерминальный символ, 2 - правило вида <termSymbol><NonTermSymbol>|<termSymbol> (или <NonTermSymbol><termSymbol>|<termSymbol>)* String[] ruleParts = ruleRepresentation.split(*transitionRuleSplitter*);  
  
 **char** nonTermSymbol = ruleParts[0].trim().charAt(0);  
*// System.out.println("ruleParts = " + Arrays.toString(ruleParts));* **for** (**int** i = 1; i < ruleParts.**length**; i++) {  
 String[] simpleRules = ruleParts[i].split(**"\\|"**); *//разбитие правила на простые правила вида <NonTermSymbol><termSymbol> или <termSymbol>  
// System.out.println("simpleRules = " + Arrays.toString(simpleRules));* **if** (!addSimpleRulesToGrammar(nonTermSymbol, simpleRules))  
 **return false**;  
 }  
 **return true**;  
 }  
  
 *//добавляем все простые правила в грамматику* **private boolean** addSimpleRulesToGrammar(**char** nonTerminal, String[] simpleRules) {  
 **for** (String simpleRule : simpleRules) {  
 **if** (!**grammar**.addTransitionRule(nonTerminal, simpleRule.trim())) {  
 **return false**;  
 }  
 }  
 **return true**;  
 }  
  
 **public boolean** addStartSymbol(String startSymbol) {  
 **return grammar**.setStartSymbol(startSymbol.trim().charAt(0));  
 }  
  
 **public** RegularGrammar getResultGrammar() {  
 **return grammar**;  
 }  
  
 **private** Set<Character> buildGrammarSymbolsSet(String grammarSymbolSet) **throws** IllegalArgumentException { *//параметр в формате {Symbol1,Symbol2,Symbol3}* String terminals = grammarSymbolSet.substring(1, grammarSymbolSet.length() - 1); *//убираем {} => termSymbol1,termSymbol1,termSymbol1* String[] splittedTerminals = terminals.split(**","**);  
 Set<Character> preparedTerminals = **new** HashSet<>();  
  
 **for** (String terminalSymbol : splittedTerminals) {  
 **char** preparedTerminal = terminalSymbol.trim().charAt(0);  
 **if** (preparedTerminals.contains(preparedTerminal)) {  
 **throw new** IllegalArgumentException(**"Дублирование символа грамматики: "** + preparedTerminal);  
 }  
 preparedTerminals.add(preparedTerminal);  
 }  
 **return** preparedTerminals;  
 }  
}

**package** grammar;  
  
**import** java.util.\*;  
  
*/\*\*  
 \** ***@author*** *Oleh Kakherskyi (olehkakherskiy@gmail.com)  
 \*/***public class** NkaModel {  
  
 **private** Set<Character> **startStates**;  
  
 **private** Set<Character> **endStates**;  
  
 **private** Set<Character> **allStates**;  
  
 **private** Set<Character> **terminals**;  
  
 **private** Map<Character, TransitionFunction> **transitions**;  
  
 **public class** TransitionFunction {  
  
 **private char state**; *//для которого определена данная функция (нетерминальный символ)* **public** TransitionFunction(**char** state) {  
 **this**.**state** = state;  
 }  
  
 **private** Map<Character, Character> **transitionFunction** = **new** IdentityHashMap<>(); *//терминальный символ - ключ, нетерминальный - значение.  
 //Хранит в себе все возможные переходы с вершины state в другие.* **void** addFunction(**char** terminalSymbol, **char** nonTerminalSymbol) {  
 **transitionFunction**.put(**new** Character(terminalSymbol), nonTerminalSymbol); *//не менять на автоупаковку -  
 // будут одинаковые объекты, а нужны разные при одном и том же символе* }  
  
 **public char** getState() {  
 **return state**;  
 }  
 }  
  
 **public** NkaModel() {  
 **startStates** = **new** HashSet<>();  
 **endStates** = **new** HashSet<>();  
 **allStates** = **new** HashSet<>();  
 **terminals** = **new** HashSet<>();  
 **transitions** = **new** HashMap<>();  
 }  
  
 **public** Map<Character, Character> getFullFunctionOf(**char** state) {  
 **return transitions**.get(state) == **null** ? **new** HashMap<>() : **transitions**.get(state).**transitionFunction**;  
 }  
  
 **public void** addFunctionPair(**char** nonTerminalArgument, **char** terminalArgument, **char** nonTerminalValue) {  
 **if** (!**transitions**.containsKey(nonTerminalArgument)) {  
 **transitions**.put(nonTerminalArgument, **new** TransitionFunction(nonTerminalArgument));  
 }  
 **transitions**.get(nonTerminalArgument).addFunction(terminalArgument, nonTerminalValue);  
 }  
  
 **public void** setAllStates(Set<Character> allStates) {  
 **for** (Character state : allStates)  
 **this**.**allStates**.add(state);  
 }  
  
 **public** Map<Character, TransitionFunction> getTransitions() {  
 **return transitions**;  
 }  
  
 **public void** setTransitions(Map<Character, TransitionFunction> transitions) {  
 **this**.**transitions** = transitions;  
 }  
  
 @Override  
 **public** String toString() {  
 String resultTemplate = **"Множество вершин: %s; \nВходной алфавит: %s; "** +  
 **"\nМножество команд:\n%s; \nМножество начальных состояний: %s; \nМножество конечных состояний: %s"**;  
 String states = **allStates**.toString();  
 String inputAlphabet = **terminals**.toString();  
 String transitionFunction = buildTransitionFunction();  
 String startStates = **this**.**startStates**.toString();  
 String endStates = **this**.**endStates**.toString();  
 **return** String.*format*(resultTemplate, states, inputAlphabet, transitionFunction, startStates, endStates);  
 }  
  
 **private** String buildTransitionFunction() {  
 String functionTemplate = **"F(%c,%c)=%c"**;  
 StringBuilder result = **new** StringBuilder();  
 **for** (**char** nonTerminalArgument : **transitions**.keySet()) {  
 TransitionFunction transitionFunction = **transitions**.get(nonTerminalArgument);  
 **for** (Character terminalArgument : transitionFunction.**transitionFunction**.keySet()) {  
 result.append(String.*format*(functionTemplate, nonTerminalArgument, terminalArgument,  
 transitionFunction.**transitionFunction**.get(terminalArgument)));  
 result.append(**"\n"**);  
 }  
 }  
 **return** result.deleteCharAt(result.length() - 1).toString();  
 }  
  
  
 **public** Set<Character> getStartStates() {  
 **return startStates**;  
 }  
  
 **public void** setStartStates(Set<Character> startStates) {  
 **this**.**startStates** = startStates;  
 }  
  
 **public** Set<Character> getEndStates() {  
 **return endStates**;  
 }  
  
 **public void** setEndStates(Set<Character> endStates) {  
 **this**.**endStates** = endStates;  
 }  
  
 **public** Set<Character> getAllStates() {  
 **return allStates**;  
 }  
  
 **public** Set<Character> getTerminals() {  
 **return terminals**;  
 }  
  
 **public void** setTerminals(Set<Character> terminals) {  
 **this**.**terminals** = terminals;  
 }  
}

**package** grammar;  
  
**import** java.util.Set;  
  
*/\*\*  
 \** ***@author*** *Oleh Kakherskyi (olehkakherskiy@gmail.com)  
 \*/***public class** NkaBuilder {  
  
 **private static final char *finalMachineState*** = **'Z'**;  
  
 **public** NkaModel buildNonDeterministicFiniteStateMachine(RegularGrammar grammar) {  
 NkaModel stateMachine = **new** NkaModel();  
  
 stateMachine.setAllStates(grammar.getNonTerminals()); *//нетерминальные символы - множество вершин НКА* stateMachine.getEndStates().add(***finalMachineState***); *//конечное состояние НКА* stateMachine.getAllStates().add(***finalMachineState***); *//конечное состояние - символ множества состояний* stateMachine.getStartStates().add(grammar.getStartSymbol()); *//начальное состояние НКА - стартовый символ грамматики* stateMachine.setTerminals(grammar.getTerminals()); *//терминальные символы - аргументы функции переходов.* Set<Character> nonTerminalSymbols = grammar.getNonTerminals();  
 **for** (**char** nonTerminal : nonTerminalSymbols) {  
 addFunctionPairForOneState(stateMachine, nonTerminal, grammar.getTransitionRules(nonTerminal), grammar);  
 }  
 **return** stateMachine;  
 }  
  
 **private void** addFunctionPairForOneState(NkaModel stateMachine, **char** nonTerminal,  
 Set<String> stateRules, RegularGrammar grammar) {  
 **for** (String rule : stateRules) { *//правило вида <termSymbol><NonTermSymbol>(может быть местами поменяно) или <termSymbol>* **if** (rule.length() == 1) { *//терминальный символ или эпсилон. Если эпсилон - добавляем вершину в конечные.* **char** terminal = rule.charAt(0);  
 **if** (terminal == **'ε'**) {  
 stateMachine.getEndStates().add(nonTerminal); *//добавили вершину в множество конечных вершин* } **else** { *//просто терминальный символ - добавляем функцию вида F(nonTerminal, terminal) = finalMachineState* stateMachine.addFunctionPair(nonTerminal, terminal, ***finalMachineState***);  
 }  
  
 } **else** { *//правило состоит из терминала и нетерминала. Обрабатываем праволинейную грамматику (терминал\_нетерминал)* stateMachine.addFunctionPair(nonTerminal, rule.charAt(0), rule.charAt(1));  
 }  
 }  
 }  
  
 **private void** addFunctionPair(NkaModel stateMachine, **char** nonTerminalArg, **char** terminalArg, **char** nonTerminalValue) {  
 stateMachine.addFunctionPair(nonTerminalArg, terminalArg, nonTerminalValue);  
 }  
  
}