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

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

ОТЧЕТ

по лабораторной работе №1

по курсу «Конструирование компиляторов»

на тему: «Распознавание цепочек регулярного языка»

Вариант № 4

Студент ИУ7-22М (Группа) _____ (Подпись, дата) И. А. Глотов (И. О. Фамилия)

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

2025 г.

ОПИСАНИЕ ЗАДАНИЯ

Цель работы: приобретение практических навыков реализации важнейших элементов лексических анализаторов на примере распознавания цепочек регулярного языка.

Задачи работы

1) ознакомиться с основными понятиями и определениями, лежащими в основе построения лексических анализаторов;

2) прояснить связь между регулярным множеством, регулярным выражением, праволинейным языком, конечно-автоматным языком и недетерминированным конечно-автоматным языком;

3) разработать, протестировать и отладить программу распознавания цепочек регулярного языка в соответствии с предложенным вариантом грамматики.

Содержание работы (Вариант 4)

Напишите программу, которая в качестве входа принимает произвольное регулярное выражение и выполняет следующие преобразования.

1) По регулярному выражению строит НКА.

2) По НКА строит эквивалентный ему ДКА.

3) По ДКА строит эквивалентный ему КА, имеющий наименьшее возможное количество состояний (алгоритм Бржозовского).

4) Моделирует минимальный КА для входной цепочки из терминалов исходной грамматики.

РЕЗУЛЬТАТЫ ВЫПОЛНЕНИЯ ПРОГРАММЫ

Введенное регулярное выражение: **(bd)*b(b|e)**

Построенный по нему НКА изображен на рисунке 1.

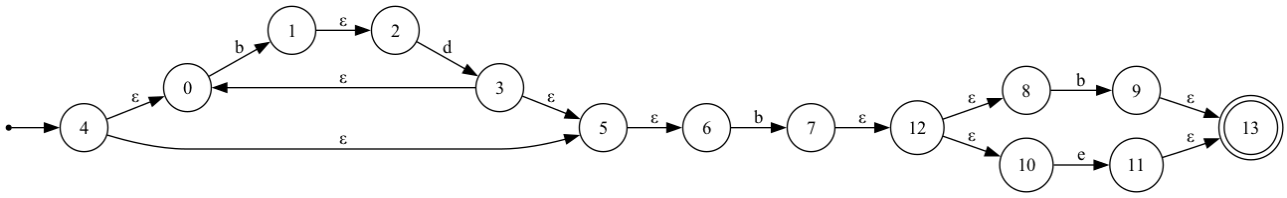

Рисунок 1 – Построенный НКА

ДКА, эквивалентный построенному НКА, изображен на рисунке 2.

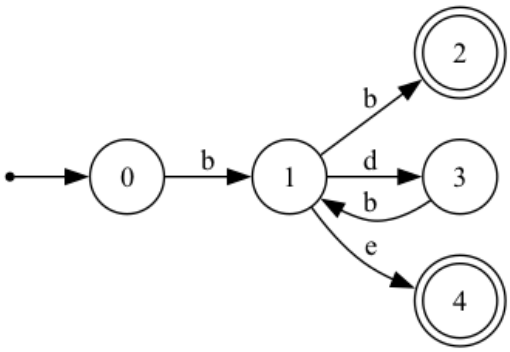

Рисунок 2 – Построенный ДКА

КА, эквивалентный построенному ДКА и имеющий наименьшее возможное количество состояний (алгоритм Бржозовского), изображен на рисунке 3

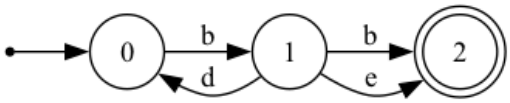

Рисунок 3 – Минимальный ДКА

ЗАКЛЮЧЕНИЕ

В ходе выполнения лабораторной работы была реализована программа на языке Go, позволяющая распознавать цепочки регулярного языка.

В программе было реализовано построение НКА по произвольно введённому регулярному выражению, по НКА был построен эквивалентный ему ДКА, по ДКА был построен эквивалентный ему КА, имеющий наименьшее возможное количество состояний (в соответствии с алгоритмом Бржозовского), а также был смоделирован минимальный КА для входной цепочки из терминалов исходной грамматики.

Таким образом, в результате выполнения лабораторной работы были приобретены практические навыки реализации важнейших элементов лексических анализаторов на примере распознавания цепочек регулярного языка.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. *Шолин И. М.*, *Чубырь Н. О.* АЛГОРИТМ ПЕРЕНОСНОЙ ШИФРОВАЛЬНОЙ МАШИНЫ ЭНИГМА. // Форум молодых ученых. — 2018.
2. Программная реализация шифровальной машины «Энигма» на языке Си [Электронный ресурс]. — Режим доступа: `https://habr.com/ru/articles/721790/` (дата обращения: 28.09.2023).

ПРИЛОЖЕНИЕ А

Листинг А.1 – Модуль infixToPostfix

```
package infixToPostix

import "strings"

const (
maxPriority = 4
)

func fillConcatenateChars(infix string) string {
  var result strings.Builder
  n := len(infix)

  for i := 0; i < n; i++ {
    result.WriteByte(infix[i])

    if i+1 < n && shouldAddConcatenateChar(infix[i], infix[i+1]) {
      result.WriteByte('.')
    }
  }

  return result.String()
}

func shouldAddConcatenateChar(a, b byte) bool {
  return (isLetterOrDigit(a) && isLetterOrDigit(b)) ||
  (isLetterOrDigit(a) && b == '(') ||
  (a == ')' && isLetterOrDigit(b)) ||
  ((a == '*' || a == '+') && (isLetterOrDigit(b) || b == '('))
}

func isLetterOrDigit(c byte) bool {
  return (c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z') || (c >=
     '0' && c <= '9')
}

var specialCharsPriorityMap = map[rune]int{
  '(': 1,
```

```
37    '|': 2,
38    '.': 3,
39    '?': 4,
40    '*': 4,
41    '+': 4,
42  }
43
44  func priorityOf(r rune) int {
45    if priority, ok := specialCharsPriorityMap[r]; ok {
46      return priority
47    }
48    return maxPriority + 1
49  }
50
51  func Transform(infix string) string {
52    infix = fillConcatenateChars(infix)
53
54    postfix := []rune{}
55    stack := []rune{}
56
57    for _, r := range infix {
58      switch r {
59        case '(':
60        stack = append(stack, r)
61        case ')':
62        for len(stack) > 0 && stack[len(stack)-1] != '(' {
63          postfix = append(postfix, stack[len(stack)-1])
64          stack = stack[:len(stack)-1]
65        }
66        if len(stack) > 0 {
67          stack = stack[:len(stack)-1]
68        }
69        default:
70        for len(stack) > 0 && priorityOf(stack[len(stack)-1]) >=
             priorityOf(r) {
71          postfix = append(postfix, stack[len(stack)-1])
72          stack = stack[:len(stack)-1]
73        }
74        stack = append(stack, r)
75      }
76
```

```
77    }
78
79    for len(stack) > 0 {
80      postfix = append(postfix, stack[len(stack)-1])
81      stack = stack[:len(stack)-1]
82    }
83
84    return string(postfix)
85  }
```

Листинг А.2 – Модуль NFA

```
1  package nfa
2
3  import (
4  "fmt"
5  "sort"
6  )
7
8  const EPS = 'eps'
9
10 type State struct {
11   ID          int
12   Transitions map[rune][]*State
13   IsFinal     bool
14 }
15
16 type NFA struct {
17   Start       *State
18   End         *State
19   StartStates []*State
20 }
21
22 func (a *NFA) ExtractAlphabet() []rune {
23   alphabetMap := make(map[rune]bool)
24
25   var traverse func(state *State)
26   traverse = func(state *State) {
27     for symbol := range state.Transitions {
28       if symbol != EPS {
29         alphabetMap[symbol] = true
30       }
31     }
```

```
32    }
33
34    visited := make(map[int]bool)
35    stack := []*State{a.Start}
36    stack = append(stack, a.StartStates...)
37
38    for len(stack) > 0 {
39      state := stack[len(stack)-1]
40      stack = stack[:len(stack)-1]
41
42      if state == nil {
43        continue
44      }
45
46      if visited[state.ID] {
47        continue
48      }
49      visited[state.ID] = true
50
51      traverse(state)
52
53      for _, nextStates := range state.Transitions {
54        for _, nextState := range nextStates {
55          stack = append(stack, nextState)
56        }
57      }
58    }
59
60    alphabet := make([]rune, 0, len(alphabetMap))
61    for symbol := range alphabetMap {
62      alphabet = append(alphabet, symbol)
63    }
64
65    sort.Slice(alphabet, func(i, j int) bool {
66      return alphabet[i] < alphabet[j]
67    })
68
69    return alphabet
70  }
71
72  func NewState(id int) *State {
```

```
73    return &State{
74      ID:          id,
75      Transitions: make(map[rune][]*State),
76    }
77  }
78
79  func New(start, end *State) *NFA {
80    return &NFA{Start: start, End: end, StartStates: []*State{}}
81  }
82
83  func Build(postfix string) *NFA {
84    stack := []*NFA{}
85    stateID := 0
86
87    for _, char := range postfix {
88      switch char {
89        case '.':
90        nfa2 := stack[len(stack)-1]
91        nfa1 := stack[len(stack)-2]
92        stack = stack[:len(stack)-2]
93
94        nfa1.End.Transitions[EPS] = append(nfa1.End.Transitions[EPS],
              nfa2.Start)
95
96        stack = append(stack, New(nfa1.Start, nfa2.End))
97        case '|':
98        nfa2 := stack[len(stack)-1]
99        nfa1 := stack[len(stack)-2]
100       stack = stack[:len(stack)-2]
101
102       start := NewState(stateID)
103       stateID++
104       end := NewState(stateID)
105       stateID++
106
107       start.Transitions[EPS] = append(start.Transitions[EPS], nfa1.
             Start, nfa2.Start)
108
109       nfa1.End.Transitions[EPS] = append(nfa1.End.Transitions[EPS],
              end)
110       nfa2.End.Transitions[EPS] = append(nfa2.End.Transitions[EPS],
```

```
            end)
111
112         stack = append(stack, New(start, end))
113         case '?':
114         nfa := stack[len(stack)-1]
115         stack = stack[:len(stack)-1]
116
117         start := NewState(stateID)
118         stateID++
119         end := NewState(stateID)
120         stateID++
121
122         start.Transitions[EPS] = append(start.Transitions[EPS], nfa.
               Start, end)
123         nfa.End.Transitions[EPS] = append(nfa.End.Transitions[EPS],
               end)
124
125         stack = append(stack, New(start, end))
126         case '*':
127         nfa := stack[len(stack)-1]
128         stack = stack[:len(stack)-1]
129
130         start := NewState(stateID)
131         stateID++
132         end := NewState(stateID)
133         stateID++
134
135         start.Transitions[EPS] = append(start.Transitions[EPS], nfa.
               Start, end)
136         nfa.End.Transitions[EPS] = append(nfa.End.Transitions[EPS],
               nfa.Start)
137         nfa.End.Transitions[EPS] = append(nfa.End.Transitions[EPS],
               end)
138
139         stack = append(stack, New(start, end))
140         case '+':
141         nfa := stack[len(stack)-1]
142         stack = stack[:len(stack)-1]
143
144         start := NewState(stateID)
145         stateID++
```

```
146         end := NewState(stateID)
147         stateID++
148 
149         start.Transitions[EPS] = append(start.Transitions[EPS], nfa.
                Start)
150         nfa.End.Transitions[EPS] = append(nfa.End.Transitions[EPS],
                nfa.Start)
151         nfa.End.Transitions[EPS] = append(nfa.End.Transitions[EPS],
                end)
152 
153         stack = append(stack, New(start, end))
154         default:
155         start := NewState(stateID)
156         stateID++
157         end := NewState(stateID)
158         stateID++
159 
160         start.Transitions[char] = append(start.Transitions[char], end
                )
161         stack = append(stack, New(start, end))
162       }
163     }
164 
165     stack[0].StartStates = append(stack[0].StartStates, stack[0].
          Start)
166     stack[0].End.IsFinal = true
167     return stack[0]
168 }
169 
170 func (a *NFA) ToGraphviz() string {
171     graph := "digraph NFA {\n"
172       graph += "  rankdir=LR;\n"
173       graph += "  node [shape = circle];\n"
174 
175       graph += "  start [shape = point];\n"
176       graph += fmt.Sprintf("  start -> %d;\n", a.Start.ID)
177 
178       graph += fmt.Sprintf("  %d [shape = doublecircle];\n", a.End.ID
              )
179 
180       visited := make(map[*State]bool)
```

```
181     stack := []*State{a.Start}
182
183     for len(stack) > 0 {
184       state := stack[len(stack)-1]
185       stack = stack[:len(stack)-1]
186
187       if visited[state] {
188         continue
189       }
190       visited[state] = true
191
192       graph += fmt.Sprintf("  %d [label=\"%d\"];\n", state.ID,
            state.ID)
193
194       for char, nextStates := range state.Transitions {
195         for _, nextState := range nextStates {
196           graph += fmt.Sprintf("  %d -> %d [label=\"%c\"];\n",
                state.ID, nextState.ID, char)
197           stack = append(stack, nextState)
198         }
199       }
200     }
201
202     graph += "}\n"
203   return graph
204 }
205
206 func (a *NFA) StateByID(stateID int) *State {
207   visited := make(map[int]bool)
208   stack := []*State{a.Start}
209   stack = append(stack, a.StartStates...)
210
211   for len(stack) > 0 {
212     state := stack[len(stack)-1]
213     stack = stack[:len(stack)-1]
214
215     if visited[state.ID] {
216       continue
217     }
218     visited[state.ID] = true
219
```

```
220     if state.ID == stateID {
221       return state
222     }
223
224     for _, nextStates := range state.Transitions {
225       for _, nextState := range nextStates {
226         stack = append(stack, nextState)
227       }
228     }
229   }
230
231   return nil
232 }
233
234 func (a *NFA) IsFinalState(states map[int]bool) bool {
235   for stateID := range states {
236     state := a.StateByID(stateID)
237     if state.IsFinal {
238       return true
239     }
240   }
241   return false
242 }
243
244 func (a *NFA) EpsilonClosure(states map[int]bool) map[int]bool {
245   closure := make(map[int]bool)
246   for stateID := range states {
247     closure[stateID] = true
248   }
249
250   stack := make([]int, 0, len(states))
251   for stateID := range states {
252     stack = append(stack, stateID)
253   }
254
255   for len(stack) > 0 {
256     currentStateID := stack[len(stack)-1]
257     stack = stack[:len(stack)-1]
258
259     state := a.StateByID(currentStateID)
260     for _, nextState := range state.Transitions[EPS] {
```

```
261         if !closure[nextState.ID] {
262           closure[nextState.ID] = true
263           stack = append(stack, nextState.ID)
264         }
265       }
266     }
267
268     return closure
269 }
```

Листинг А.3 – Модуль DFA

```
1  package dfa
2
3  import (
4  "fmt"
5
6  nfa_pkg "github.com/Erlendum/BMSTU_CC/internal/nfa"
7  )
8
9  type State struct {
10   id           int
11   nfaStates    map[int]bool
12   transitions map[rune]int
13   isFinal      bool
14 }
15
16 type DFA struct {
17   start    int
18   states   map[int]*State
19   alphabet []rune
20 }
21
22 func NewState(id int, nfaStates map[int]bool, isFinal bool) *State
       {
23   return &State{
24     id:           id,
25     nfaStates:    nfaStates,
26     transitions: make(map[rune]int),
27     isFinal:      isFinal,
28   }
29 }
30
```

```
31 func Build(nfa *nfa_pkg.NFA) *DFA {
32   alphabet := nfa.ExtractAlphabet()
33
34   dfa := &DFA{
35     states:   make(map[int]*State),
36     alphabet: alphabet,
37   }
38
39   startedStates := make(map[int]bool)
40   for _, state := range nfa.StartStates {
41     startedStates[state.ID] = true
42   }
43
44   startNFAStates := nfa.EpsilonClosure(startedStates)
45   dfa.start = 0
46   dfa.states[0] = NewState(0, startNFAStates, nfa.IsFinalState(
       startNFAStates))
47
48   queue := []int{0}
49   processed := make(map[int]bool)
50
51   stateID := 1
52
53   for len(queue) > 0 {
54     currentStateID := queue[0]
55     queue = queue[1:]
56
57     if processed[currentStateID] {
58       continue
59     }
60     processed[currentStateID] = true
61
62     currentState := dfa.states[currentStateID]
63
64     for _, symbol := range alphabet {
65       nextNFAStates := make(map[int]bool)
66       for nfaStateID := range currentState.nfaStates {
67         state := nfa.StateByID(nfaStateID)
68         for _, nextState := range state.Transitions[symbol] {
69           nextNFAStates[nextState.ID] = true
70         }
```

```
71        }
72
73        nextNFAStates = nfa.EpsilonClosure(nextNFAStates)
74
75        if len(nextNFAStates) == 0 {
76          continue
77        }
78
79        found := false
80        var nextStateID int
81        for id, state := range dfa.states {
82          if statesEqual(state.nfaStates, nextNFAStates) {
83            found = true
84            nextStateID = id
85            break
86          }
87        }
88
89        if !found {
90          nextStateID = stateID
91          dfa.states[nextStateID] = NewState(nextStateID,
               nextNFAStates, nfa.IsFinalState(nextNFAStates))
92          queue = append(queue, nextStateID)
93          stateID++
94        }
95
96        currentState.transitions[symbol] = nextStateID
97      }
98    }
99
100   return dfa
101 }
102
103 func statesEqual(a, b map[int]bool) bool {
104   if len(a) != len(b) {
105     return false
106   }
107   for stateID := range a {
108     if !b[stateID] {
109       return false
110     }
```

```
111     }
112     return true
113 }
114 
115 func (dfa *DFA) ToGraphviz() string {
116     graph := "digraph DFA {\n"
117         graph += "  rankdir=LR;\n"
118         graph += "  node [shape = circle];\n"
119 
120         graph += fmt.Sprintf("  start [shape = point];\n")
121         graph += fmt.Sprintf("  start -> %d;\n", dfa.start)
122 
123         for _, state := range dfa.states {
124             if state.isFinal {
125                 graph += fmt.Sprintf("  %d [shape = doublecircle];\n",
                        state.id)
126             } else {
127                 graph += fmt.Sprintf("  %d [shape = circle];\n", state.id)
128             }
129         }
130 
131         for _, state := range dfa.states {
132             for symbol, nextStateID := range state.transitions {
133                 graph += fmt.Sprintf("  %d -> %d [label=\"%c\"];\n", state.
                        id, nextStateID, symbol)
134             }
135         }
136 
137         graph += "}\n"
138     return graph
139 }
140 
141 func (dfa *DFA) Minimize() *DFA {
142     invertedNFA := dfa.invert()
143     intermediateDFA := Build(invertedNFA)
144     invertedNFA2 := intermediateDFA.invert()
145     minimizedDFA := Build(invertedNFA2)
146     return minimizedDFA
147 }
148 
149 func (dfa *DFA) invert() *nfa_pkg.NFA {
```

```
150   stateMap := make(map[int]*nfa_pkg.State)
151   startStates := make([]*nfa_pkg.State, 0)
152   startRandomID := 0
153   for id, state := range dfa.states {
154     stateMap[id] = &nfa_pkg.State{ID: id, Transitions: map[rune][]*
          nfa_pkg.State{}}
155     if state.isFinal {
156       startRandomID = state.id
157       startStates = append(startStates, stateMap[id])
158     }
159   }
160
161   stateMap[dfa.start].IsFinal = true
162
163   for fromID, state := range dfa.states {
164     for symbol, toID := range state.transitions {
165       if _, exists := stateMap[toID]; !exists {
166         stateMap[toID] = &nfa_pkg.State{ID: toID, Transitions: map[
              rune][]*nfa_pkg.State{}}
167       }
168       stateMap[toID].Transitions[symbol] = append(stateMap[toID].
            Transitions[symbol], stateMap[fromID])
169     }
170   }
171
172   nfa := &nfa_pkg.NFA{
173     Start:       stateMap[startRandomID],
174     End:         stateMap[dfa.start],
175     StartStates: startStates,
176   }
177
178   return nfa
179 }
180
181 func (dfa *DFA) SimulateDFA(input string) ([]string, bool) {
182   var steps []string
183   currentStateID := dfa.start
184   currentState := dfa.states[currentStateID]
185
186   steps = append(steps, dfa.ToGraphvizWithHighlight(currentStateID,
        "Start"))
```

```
187
188   for i, symbol := range input {
189     if nextStateID, exists := currentState.transitions[symbol];
          exists {
190       currentStateID = nextStateID
191       currentState = dfa.states[currentStateID]
192       steps = append(steps, dfa.ToGraphvizWithHighlight(
            currentStateID, fmt.Sprintf("Step %d: Symbol '%c'", i+1,
            symbol)))
193     } else {
194       steps = append(steps, dfa.ToGraphvizWithError(currentStateID,
            symbol))
195       return steps, false
196     }
197   }
198
199   isAccepted := currentState.isFinal
200   if isAccepted {
201     steps = append(steps, dfa.ToGraphvizWithHighlight(
        currentStateID, "Accepted"))
202   } else {
203     steps = append(steps, dfa.ToGraphvizWithHighlight(
        currentStateID, "Rejected"))
204   }
205
206   return steps, isAccepted
207 }
208
209 func (dfa *DFA) ToGraphvizWithHighlight(currentStateID int,
     description string) string {
210   graph := "digraph DFA {\n"
211     graph += "  rankdir=LR;\n"
212     graph += "  node [shape = circle];\n"
213
214     graph += fmt.Sprintf("  start [shape = point];\n")
215     graph += fmt.Sprintf("  start -> %d;\n", dfa.start)
216
217     for _, state := range dfa.states {
218       if state.isFinal {
219         graph += fmt.Sprintf("  %d [shape = doublecircle];\n",
              state.id)
```

```
220         } else {
221           graph += fmt.Sprintf("  %d [shape = circle];\n", state.id)
222         }
223       }
224
225       graph += fmt.Sprintf("  %d [color=red, fontcolor=red];\n",
              currentStateID)
226
227       graph += fmt.Sprintf("  labelloc=\"t\";\n")
228       graph += fmt.Sprintf("  label=\"%s\";\n", description)
229
230       for _, state := range dfa.states {
231         for symbol, nextStateID := range state.transitions {
232           graph += fmt.Sprintf("  %d -> %d [label=\"%c\"];\n", state.
                id, nextStateID, symbol)
233         }
234       }
235
236       graph += "}\n"
237     return graph
238 }
239
240 func (dfa *DFA) ToGraphvizWithError(currentStateID int, symbol rune
       ) string {
241     graph := "digraph DFA {\n"
242       graph += "  rankdir=LR;\n"
243       graph += "  node [shape = circle];\n"
244
245       graph += fmt.Sprintf("  start [shape = point];\n")
246       graph += fmt.Sprintf("  start -> %d;\n", dfa.start)
247
248       for _, state := range dfa.states {
249         if state.isFinal {
250           graph += fmt.Sprintf("  %d [shape = doublecircle];\n",
                state.id)
251         } else {
252           graph += fmt.Sprintf("  %d [shape = circle];\n", state.id)
253         }
254       }
255
256       graph += fmt.Sprintf("  %d [color=red, fontcolor=red];\n",
```

```
        currentStateID)
257     graph += fmt.Sprintf("  %d -> error [label=\"%c\"];\n",
        currentStateID, symbol)
258     graph += "  error [shape=box, color=red, fontcolor=red];\n"
259
260     graph += fmt.Sprintf("  labelloc=\"t\";\n")
261     graph += fmt.Sprintf("  label=\"Error: No transition for symbol
        '%c'\";\n", symbol)
262
263     for _, state := range dfa.states {
264       for symbol, nextStateID := range state.transitions {
265         graph += fmt.Sprintf("  %d -> %d [label=\"%c\"];\n", state.
            id, nextStateID, symbol)
266       }
267     }
268
269     graph += "}\n"
270   return graph
271 }
```