Министерство науки и высшего образования Российской Федерации
Федеральное государственное автомное 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»

Отчёт по лабораторной работе №1
по курсу «Конструирование компиляторов»

Вариант №2

Тема Распознование цепочек регулярного языка

Студент Аскарян К.А.

Группа ИУ7-21М

Преподаватель Ступников А. А.

Москва — 2025 г.

1 Теоретическая часть

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, тестировать и отладить программу распознавания цепочек регулярного или праволинейного языка в соответствии с предложенным вариантом грамматики.

1.1 Задание

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. Указание. Воспользоваться алгоритмом, приведенным по адресу *http://neerc.ifmo.ru/wiki/index.php?title=Минимизация_ДКА,_алгоритм_за_O(n%5E2)_с_построением_пар_раз*
4. Моделирует минимальный КА для входной цепочки из терминалов исходной грамматики.

2 Практическая часть

2.1 Результат выполнения работы

В таблице 2.1 приведены результаты работы программы.

Таблица 2.1 – Результаты расчетов для задачи из индивидуального варианта

Регулярное выражение	Входная строка	Результат
(ab)*c	a	НЕТ
	abc	ДА
	c	ДА
	ababc	ДА
(a|b)*c	a	НЕТ
	ac	ДА
	abc	ДА
	aaaaaaabc	ДА
ab(ab|b*)*	ab	ДА
	ababb	ДА
	c	НЕТ
	ababbab	ДА

2.2 Код программы

В листингах 2.1 — 2.6 приведен код программы на языке Go.

Листинг 2.1 – Код модуля *commands*

```go
package commands

import (
    "bytes"
    "fmt"
    "github.com/AskaryanKarine/BMSTU-CC/lab_01/internal/dfa"
    "github.com/AskaryanKarine/BMSTU-CC/lab_01/internal/fs"
    "github.com/AskaryanKarine/BMSTU-CC/lab_01/internal/nfa"
    "github.com/AskaryanKarine/BMSTU-CC/lab_01/internal/transform"
    "os"
    "os/exec"
)

const (
    nfaFileDot = "graphs/nfa.dot"
```

```
16      nfaFilePng = "graphs/nfa.png"
17      dfaFileDot = "graphs/dfa.dot"
18      dfaFilePng = "graphs/dfa.png"
19      minFileDot = "graphs/min.dot"
20      minFilePng = "graphs/min.png"
21      stepsDir   = "graphs/emulate/"
22  )
23
24  type Commands struct {
25      outputBuffer bytes.Buffer
26      dfa          *dfa.DFA
27  }
28
29  func (c *Commands) RegularToFAs(input string) (string, error) {
30      c.outputBuffer.Reset()
31      postfixForm := transform.InfixToPostfix(input)
32
33      initialNFA := nfa.Build(postfixForm)
34      if err := c.saveAutomaton("NFA", initialNFA.ToGraphviz(), nfaFileDot,
            nfaFilePng); err != nil {
35          return c.output(), err
36      }
37
38      builtDFA := dfa.Build(initialNFA)
39      if err := c.saveAutomaton("DFA", builtDFA.ToGraphviz(), dfaFileDot,
            dfaFilePng); err != nil {
40          return c.output(), err
41      }
42
43      minimizeDFA := builtDFA.Minimize()
44      if err := c.saveAutomaton("Minimized␣DFA", minimizeDFA.ToGraphviz(),
            minFileDot, minFilePng); err != nil {
45          return c.output(), err
46      }
47
48      c.dfa = minimizeDFA
49
50      return c.output(), nil
51  }
52
53  func (c *Commands) saveAutomaton(name, dotData, dotPath, pngPath string)
        error {
54      if err := os.WriteFile(dotPath, []byte(dotData), 0644); err != nil {
55          msg := fmt.Sprintf("create␣file␣%s␣error:␣%v\n", name, err)
56          c.outputBuffer.WriteString(msg)
57
58          return fmt.Errorf(msg)
59      }
```

```
60
61      cmd := exec.Command("dot", "-Tpng", "-o", pngPath, dotPath)
62      if output, err := cmd.CombinedOutput(); err != nil {
63          % msg := fmt.Sprintf("Graphviz␣error␣for␣%s:␣%v\nOutput:␣%s\n",
                name, err, string(output))
64          c.outputBuffer.WriteString(msg)
65
66          return fmt.Errorf(msg)
67      }
68
69      return nil
70  }
71
72  func (c *Commands) Emulate(input string) (string, error) {
73      c.outputBuffer.Reset()
74
75      err := fs.RecreateEmulateDir()
76      if err != nil {
77          % msg := fmt.Sprintf("error␣for␣create␣folder␣emulate:␣%v\n", err)
78          c.outputBuffer.WriteString(msg)
79
80          return c.output(), err
81      }
82      steps, ok := c.dfa.SimulateDFA(input)
83      for i, step := range steps {
84          name := fmt.Sprintf("step_%d", i)
85          err := c.saveAutomaton(name, step, stepsDir+name+".dot",
                stepsDir+name+".png")
86          if err != nil {
87              return c.output(), err
88          }
89      }
90
91      if ok {
92          % msg := fmt.Sprintf("String␣%s␣is␣OK", input)
93          c.outputBuffer.WriteString(msg)
94      } else {
95          % msg := fmt.Sprintf("String␣%s␣isn't␣OK", input)
96          c.outputBuffer.WriteString(msg)
97      }
98
99      return c.output(), nil
100 }
101
102 func (c *Commands) output() string {
103     return c.outputBuffer.String()
104 }
```

Листинг 2.2 – Код модуля *dfa*

```go
package dfa

import (
    "fmt"
    "github.com/AskaryanKarine/BMSTU-CC/lab_01/internal/nfa"
)

type State struct {
    ID          int
    NFAStates   map[int]bool
    Transitions map[rune]int
    IsFinal     bool
}

type DFA struct {
    Start    int
    States   map[int]*State
    Alphabet []rune
}

func NewState(id int, nfaStates map[int]bool, isFinal bool) *State {
    return &State{
        ID:          id,
        NFAStates:   nfaStates,
        Transitions: make(map[rune]int),
        IsFinal:     isFinal,
    }
}

func Build(nfa *nfa.NFA) *DFA {
    alphabet := nfa.ExtractAlphabet()

    dfa := &DFA{
        States:   make(map[int]*State),
        Alphabet: alphabet,
    }

    startedStates := make(map[int]bool)
    startedStates[nfa.Start.ID] = true

    startNFAStates := nfa.EpsilonClosure(startedStates)
    dfa.Start = 0
    dfa.States[0] = NewState(0, startNFAStates,
       nfa.IsFinalState(startNFAStates))

    queue := []int{0}
    processed := make(map[int]bool)
```

```
47
48      stateID := 1
49
50      for len(queue) > 0 {
51          currentStateID := queue[0]
52          queue = queue[1:]
53
54          if processed[currentStateID] {
55              continue
56          }
57          processed[currentStateID] = true
58
59          currentState := dfa.States[currentStateID]
60
61          for _, symbol := range alphabet {
62              nextNFAStates := make(map[int]bool)
63              for nfaStateID := range currentState.NFAStates {
64                  state := nfa.StateByID(nfaStateID)
65                  for _, nextState := range state.Transitions[symbol] {
66                      nextNFAStates[nextState.ID] = true
67                  }
68              }
69
70              nextNFAStates = nfa.EpsilonClosure(nextNFAStates)
71
72              if len(nextNFAStates) == 0 {
73                  continue
74              }
75
76              found := false
77              var nextStateID int
78              for id, state := range dfa.States {
79                  if statesEqual(state.NFAStates, nextNFAStates) {
80                      found = true
81                      nextStateID = id
82                      break
83                  }
84              }
85
86              if !found {
87                  nextStateID = stateID
88                  dfa.States[nextStateID] = NewState(nextStateID,
                        nextNFAStates, nfa.IsFinalState(nextNFAStates))
89                  queue = append(queue, nextStateID)
90                  stateID++
91              }
92
93              currentState.Transitions[symbol] = nextStateID
```

```
94          }
95      }
96
97      return dfa
98  }
99
100 func statesEqual(a, b map[int]bool) bool {
101     if len(a) != len(b) {
102         return false
103     }
104     for stateID := range a {
105         if !b[stateID] {
106             return false
107         }
108     }
109     return true
110 }
111
112 func (dfa *DFA) ToGraphviz() string {
113     graph := "digraph␣DFA␣{\n"
114     graph += "␣␣rankdir=LR;\n"
115     graph += "␣␣node␣[shape␣=␣circle];\n"
116
117     graph += fmt.Sprintf("␣␣start␣[shape␣=␣point];\n")
118     graph += fmt.Sprintf("␣␣start␣->␣%d;\n", dfa.Start)
119
120     for _, state := range dfa.States {
121         if state.IsFinal {
122             graph += fmt.Sprintf("␣␣%d␣[shape␣=␣doublecircle];\n",
                    state.ID)
123         } else {
124             graph += fmt.Sprintf("␣␣%d␣[shape␣=␣circle];\n", state.ID)
125         }
126     }
127
128     for _, state := range dfa.States {
129         for symbol, nextStateID := range state.Transitions {
130             graph += fmt.Sprintf("␣␣%d␣->␣%d␣[label=\"%c\"];\n", state.ID,
                    nextStateID, symbol)
131         }
132     }
133
134     graph += "}\n"
135     return graph
136 }
137
138 func (dfa *DFA) SimulateDFA(input string) ([]string, bool) {
139     var steps []string
```

```
140     currentStateID := dfa.Start
141     currentState := dfa.States[currentStateID]
142 
143     steps = append(steps, dfa.ToGraphvizWithHighlight(currentStateID,
            "Начало"))
144 
145     for i, symbol := range input {
146         if nextStateID, exists := currentState.Transitions[symbol]; exists
                {
147             currentStateID = nextStateID
148             currentState = dfa.States[currentStateID]
149             steps = append(steps,
                    dfa.ToGraphvizWithHighlight(currentStateID,
                    fmt.Sprintf("Шаг %d -- символ '%c'", i+1, symbol)))
150         } else {
151             steps = append(steps, dfa.ToGraphvizWithError(currentStateID,
                    symbol))
152             return steps, false
153         }
154     }
155 
156     isAccepted := currentState.IsFinal
157     if isAccepted {
158         steps = append(steps, dfa.ToGraphvizWithHighlight(currentStateID,
                "Допускается"))
159     } else {
160         steps = append(steps, dfa.ToGraphvizWithHighlight(currentStateID,
                "НЕ допускается"))
161     }
162 
163     return steps, isAccepted
164 }
165 
166 func (dfa *DFA) ToGraphvizWithHighlight(currentStateID int, description
        string) string {
167     graph := "digraph DFA {\n"
168     graph += "  rankdir=LR;\n"
169     graph += "  node [shape = circle];\n"
170 
171     graph += fmt.Sprintf("  start [shape = point];\n")
172     graph += fmt.Sprintf("  start -> %d;\n", dfa.Start)
173 
174     for _, state := range dfa.States {
175         if state.IsFinal {
176             graph += fmt.Sprintf("  %d [shape = doublecircle];\n",
                    state.ID)
177         } else {
178             graph += fmt.Sprintf("  %d [shape = circle];\n", state.ID)
```

```
179         }
180     }
181
182     graph += fmt.Sprintf("␣␣%d␣[color=red,␣fontcolor=red];\n",
            currentStateID)
183
184     graph += fmt.Sprintf("␣␣labelloc=\"t\";\n")
185     graph += fmt.Sprintf("␣␣label=\"%s\";\n", description)
186
187     for _, state := range dfa.States {
188         for symbol, nextStateID := range state.Transitions {
189             graph += fmt.Sprintf("␣␣%d␣->␣%d␣[label=\"%c\"];\n", state.ID,
                    nextStateID, symbol)
190         }
191     }
192
193     graph += "}\n"
194     return graph
195 }
196
197 func (dfa *DFA) ToGraphvizWithError(currentStateID int, symbol rune)
        string {
198     graph := "digraph␣DFA␣{\n"
199     graph += "␣␣rankdir=LR;\n"
200     graph += "␣␣node␣[shape␣=␣circle];\n"
201
202     graph += fmt.Sprintf("␣␣start␣[shape␣=␣point];\n")
203     graph += fmt.Sprintf("␣␣start␣->␣%d;\n", dfa.Start)
204
205     for _, state := range dfa.States {
206         if state.IsFinal {
207             graph += fmt.Sprintf("␣␣%d␣[shape␣=␣doublecircle];\n",
                    state.ID)
208         } else {
209             graph += fmt.Sprintf("␣␣%d␣[shape␣=␣circle];\n", state.ID)
210         }
211     }
212
213     graph += fmt.Sprintf("␣␣%d␣[color=red,␣fontcolor=red];\n",
            currentStateID)
214     graph += fmt.Sprintf("␣␣%d␣->␣ошибка␣[label=\"%c\"];\n",
            currentStateID, symbol)
215     graph += "␣␣ошибка␣[shape=box,␣color=red,␣fontcolor=red];\n"
216
217     graph += fmt.Sprintf("␣␣labelloc=\"t\";\n")
218     graph += fmt.Sprintf("␣␣labelОшибка=\":␣нет␣перехода␣для␣символа␣
            '%c'\";\n", symbol)
219
```

```
    for _, state := range dfa.States {
        for symbol, nextStateID := range state.Transitions {
            graph += fmt.Sprintf("␣␣%d␣->␣%d␣[label=\"%c\"];\n", state.ID,
                nextStateID, symbol)
        }
    }

    graph += "}\n"
    return graph
}
```

Листинг 2.3 – Код модуля *dfa*

```
package dfa

func (dfa *DFA) buildReverseTransitions() map[int]map[rune][]int {
    reverse := make(map[int]map[rune][]int)
    for id := range dfa.States {
        reverse[id] = make(map[rune][]int)
    }
    for fromID, state := range dfa.States {
        for symbol, toID := range state.Transitions {
            reverse[toID][symbol] = append(reverse[toID][symbol], fromID)
        }
    }
    return reverse
}

func (dfa *DFA) findReachable() map[int]bool {
    reachable := make(map[int]bool)
    queue := []int{dfa.Start}
    reachable[dfa.Start] = true

    for len(queue) > 0 {
        current := queue[0]
        queue = queue[1:]

        for _, next := range dfa.States[current].Transitions {
            if !reachable[next] {
                reachable[next] = true
                queue = append(queue, next)
            }
        }
    }
    return reachable
}

func (dfa *DFA) buildMarkedTable(reverseTransitions
    map[int]map[rune][]int) [][]bool {
```

```
36      maxID := 0
37      for id := range dfa.States {
38          if id > maxID {
39              maxID = id
40          }
41      }
42      marked := make([][]bool, maxID+1)
43      for i := range marked {
44          marked[i] = make([]bool, maxID+1)
45      }
46
47      var queue [][2]int
48      for i := range maxID + 1 {
49          for j := range maxID + 1 {
50              if !marked[i][j] && dfa.States[i].IsFinal !=
                    dfa.States[j].IsFinal {
51                  marked[i][j] = true
52                  marked[j][i] = true
53                  queue = append(queue, [2]int{i, j})
54              }
55          }
56      }
57
58      for len(queue) > 0 {
59          pair := queue[0]
60          queue = queue[1:]
61          u, v := pair[0], pair[1]
62
63          for _, symbol := range dfa.Alphabet {
64              for _, r := range reverseTransitions[u][symbol] {
65                  for _, s := range reverseTransitions[v][symbol] {
66                      if !marked[r][s] {
67                          marked[r][s] = true
68                          marked[s][r] = true
69                          queue = append(queue, [2]int{r, s})
70                      }
71                  }
72              }
73          }
74      }
75
76      return marked
77  }
78
79  func (dfa *DFA) buildComponents(marked [][]bool, reachable map[int]bool)
        []int {
80      component := make([]int, len(dfa.States))
81      currentComp := 0
```

```
82
83      for id := range dfa.States {
84          component[id] = -1
85      }
86
87      for i := range len(dfa.States) {
88          if !marked[0][i] {
89              component[i] = currentComp
90          }
91      }
92
93      for id := 1; id < len(component); id++ {
94          if !reachable[id] {
95              continue
96          }
97          if component[id] == -1 {
98              currentComp++
99              component[id] = currentComp
100             for j := id + 1; j < len(component); j++ {
101                 if !marked[id][j] {
102                     component[j] = currentComp
103                 }
104             }
105         }
106     }
107
108     return component
109 }
110
111 func (dfa *DFA) buildMinimizedDFA(component []int) *DFA {
112     minimized := &DFA{
113         States:   make(map[int]*State),
114         Alphabet: dfa.Alphabet,
115     }
116
117     compToState := make(map[int]*State)
118     for _, comp := range component {
119         if comp == -1 {
120             continue // Игнорируем недостижимые
121         }
122         if _, exists := compToState[comp]; !exists {
123             isFinal := false
124             for stateID, c := range component {
125                 if c == comp && dfa.States[stateID].IsFinal {
126                     isFinal = true
127                     break
128                 }
129             }
```

```
130             newState := &State{
131                 ID:          comp,
132                 IsFinal:     isFinal,
133                 Transitions: make(map[rune]int),
134             }
135             compToState[comp] = newState
136             minimized.States[comp] = newState
137         }
138     }
139
140     for comp, state := range compToState {
141         var sampleStateID int
142         for id, c := range component {
143             if c == comp {
144                 sampleStateID = id
145                 break
146             }
147         }
148         for symbol, targetID := range
              dfa.States[sampleStateID].Transitions {
149             targetComp := component[targetID]
150             if targetComp != -1 {
151                 state.Transitions[symbol] = targetComp
152             }
153         }
154     }
155
156     minimized.Start = component[dfa.Start]
157     return minimized
158 }
159
160 func (dfa *DFA) Minimize() *DFA {
161     extendedDFA := dfa.addZeroState()
162
163     reverseTransitions := extendedDFA.buildReverseTransitions()
164
165     reachable := extendedDFA.findReachable()
166
167     marked := extendedDFA.buildMarkedTable(reverseTransitions)
168
169     component := extendedDFA.buildComponents(marked, reachable)
170
171     minimized := extendedDFA.buildMinimizedDFA(component)
172
173     return minimized.removeZeroState()
174 }
175
176 func (dfa *DFA) addZeroState() *DFA {
```

```
177     newDFA := &DFA{
178         Start:    dfa.Start + 1, // Стартовое состояние теперь ID+1
179         Alphabet: dfa.Alphabet,
180         States:   make(map[int]*State),
181     }
182 
183     for oldID, state := range dfa.States {
184         newState := &State{
185             ID:          oldID + 1,
186             Transitions: make(map[rune]int),
187             IsFinal:     state.IsFinal,
188         }
189         for symbol, target := range state.Transitions {
190             newState.Transitions[symbol] = target + 1
191         }
192         newDFA.States[newState.ID] = newState
193     }
194 
195     trap := &State{
196         ID:          0,
197         Transitions: make(map[rune]int),
198         IsFinal:     false,
199     }
200     for _, symbol := range newDFA.Alphabet {
201         trap.Transitions[symbol] = 0
202     }
203     newDFA.States[0] = trap
204 
205     for _, state := range newDFA.States {
206         if state.ID == 0 {
207             continue
208         }
209         for _, symbol := range newDFA.Alphabet {
210             if _, exists := state.Transitions[symbol]; !exists {
211                 state.Transitions[symbol] = 0
212             }
213         }
214     }
215 
216     return newDFA
217 }
218 
219 func (dfa *DFA) removeZeroState() *DFA {
220     cleaned := &DFA{
221         Start:    dfa.Start,
222         Alphabet: dfa.Alphabet,
223         States:   make(map[int]*State),
224     }
```

```
225
226     for id, state := range dfa.States {
227         if id == 0 {
228             continue
229         }
230         newState := &State{
231             ID:          state.ID,
232             IsFinal:     state.IsFinal,
233             Transitions: make(map[rune]int),
234         }
235
236         for symbol, target := range state.Transitions {
237             if target != 0 {
238                 newState.Transitions[symbol] = target
239             }
240         }
241
242         cleaned.States[id] = newState
243     }
244
245     return cleaned
246 }
```

Листинг 2.4 – Код модуля *fs*

```
1  package fs
2
3  import (
4      "fmt"
5      "os"
6      "path/filepath"
7  )
8
9  func CreateGraphsDir() error {
10     err := os.MkdirAll("graphs/emulate", 0755)
11     if err != nil {
12         return fmt.Errorf("error creating dirs: %v", err)
13     }
14     return nil
15 }
16
17 func DeleteGraphsDir() error {
18     err := os.RemoveAll("graphs")
19     if err != nil {
20         return fmt.Errorf("error deleting dir: %v", err)
21     }
22     return nil
23 }
24
```

```
func RecreateEmulateDir() error {
    emulatePath := filepath.Join("graphs", "emulate")

    if err := os.RemoveAll(emulatePath); err != nil {
        return fmt.Errorf("can't delete folder: %v", err)
    }

    if err := os.MkdirAll(emulatePath, 0755); err != nil {
        return fmt.Errorf("can't create folder: %v", err)
    }

    return nil
}
```

Листинг 2.5 – Код модуля *nfa*

```
package nfa

import (
    "fmt"
    "sort"
)

const EPS = 'eps'

type State struct {
    ID          int
    Transitions map[rune][]*State
    IsFinal     bool
}

type NFA struct {
    Start *State
    End   *State
}

func (a *NFA) ExtractAlphabet() []rune {
    alphabetMap := make(map[rune]bool)

    var traverse func(state *State)
    traverse = func(state *State) {
        for symbol := range state.Transitions {
            if symbol != EPS {
                alphabetMap[symbol] = true
            }
        }
    }

    visited := make(map[int]bool)
```

```
34      stack := []*State{a.Start}
35
36      for len(stack) > 0 {
37          state := stack[len(stack)-1]
38          stack = stack[:len(stack)-1]
39
40          if state == nil {
41              continue
42          }
43
44          if visited[state.ID] {
45              continue
46          }
47          visited[state.ID] = true
48
49          traverse(state)
50
51          for _, nextStates := range state.Transitions {
52              for _, nextState := range nextStates {
53                  stack = append(stack, nextState)
54              }
55          }
56      }
57
58      alphabet := make([]rune, 0, len(alphabetMap))
59      for symbol := range alphabetMap {
60          alphabet = append(alphabet, symbol)
61      }
62
63      sort.Slice(alphabet, func(i, j int) bool {
64          return alphabet[i] < alphabet[j]
65      })
66
67      return alphabet
68  }
69
70  func NewState(id int) *State {
71      return &State{
72          ID:          id,
73          Transitions: make(map[rune][]*State),
74      }
75  }
76
77  func New(start, end *State) *NFA {
78      return &NFA{Start: start, End: end}
79  }
80
81  func Build(postfix string) *NFA {
```

```
82      var stack []*NFA
83      stateID := 0
84
85      for _, char := range postfix {
86          switch char {
87          case '.':
88              nfa2 := stack[len(stack)-1]
89              nfa1 := stack[len(stack)-2]
90              stack = stack[:len(stack)-2]
91
92              nfa1.End.Transitions[EPS] = append(nfa1.End.Transitions[EPS],
                    nfa2.Start)
93
94              stack = append(stack, New(nfa1.Start, nfa2.End))
95          case '|':
96              nfa2 := stack[len(stack)-1]
97              nfa1 := stack[len(stack)-2]
98              stack = stack[:len(stack)-2]
99
100             start := NewState(stateID)
101             stateID++
102             end := NewState(stateID)
103             stateID++
104
105             start.Transitions[EPS] = append(start.Transitions[EPS],
                    nfa1.Start, nfa2.Start)
106
107             nfa1.End.Transitions[EPS] = append(nfa1.End.Transitions[EPS],
                    end)
108             nfa2.End.Transitions[EPS] = append(nfa2.End.Transitions[EPS],
                    end)
109
110             stack = append(stack, New(start, end))
111         case '?':
112             nfa := stack[len(stack)-1]
113             stack = stack[:len(stack)-1]
114
115             start := NewState(stateID)
116             stateID++
117             end := NewState(stateID)
118             stateID++
119
120             start.Transitions[EPS] = append(start.Transitions[EPS],
                    nfa.Start, end)
121             nfa.End.Transitions[EPS] = append(nfa.End.Transitions[EPS],
                    end)
122
123             stack = append(stack, New(start, end))
```

```
124         case '*':
125             nfa := stack[len(stack)-1]
126             stack = stack[:len(stack)-1]
127 
128             start := NewState(stateID)
129             stateID++
130             end := NewState(stateID)
131             stateID++
132 
133             start.Transitions[EPS] = append(start.Transitions[EPS],
                    nfa.Start, end)
134             nfa.End.Transitions[EPS] = append(nfa.End.Transitions[EPS],
                    nfa.Start)
135             nfa.End.Transitions[EPS] = append(nfa.End.Transitions[EPS],
                    end)
136 
137             stack = append(stack, New(start, end))
138         case '+':
139             nfa := stack[len(stack)-1]
140             stack = stack[:len(stack)-1]
141 
142             start := NewState(stateID)
143             stateID++
144             end := NewState(stateID)
145             stateID++
146 
147             start.Transitions[EPS] = append(start.Transitions[EPS],
                    nfa.Start)
148             nfa.End.Transitions[EPS] = append(nfa.End.Transitions[EPS],
                    nfa.Start)
149             nfa.End.Transitions[EPS] = append(nfa.End.Transitions[EPS],
                    end)
150 
151             stack = append(stack, New(start, end))
152         default:
153             start := NewState(stateID)
154             stateID++
155             end := NewState(stateID)
156             stateID++
157 
158             start.Transitions[char] = append(start.Transitions[char], end)
159             stack = append(stack, New(start, end))
160         }
161     }
162 
163     stack[0].End.IsFinal = true
164     return stack[0]
165 }
```

```
166
167 func (a *NFA) ToGraphviz() string {
168     graph := "digraph␣NFA␣{\n"
169     graph += "␣␣rankdir=LR;\n"
170     graph += "␣␣node␣[shape␣=␣circle];\n"
171
172     graph += "␣␣start␣[shape␣=␣point];\n"
173     graph += fmt.Sprintf("␣␣start␣->␣%d;\n", a.Start.ID)
174
175     graph += fmt.Sprintf("␣␣%d␣[shape␣=␣doublecircle];\n", a.End.ID)
176
177     visited := make(map[*State]bool)
178     stack := []*State{a.Start}
179
180     for len(stack) > 0 {
181         state := stack[len(stack)-1]
182         stack = stack[:len(stack)-1]
183
184         if visited[state] {
185             continue
186         }
187         visited[state] = true
188
189         graph += fmt.Sprintf("␣␣%d␣[label=\"%d\"];\n", state.ID, state.ID)
190
191         for char, nextStates := range state.Transitions {
192             for _, nextState := range nextStates {
193                 graph += fmt.Sprintf("␣␣%d␣->␣%d␣[label=\"%c\"];\n",
                        state.ID, nextState.ID, char)
194                 stack = append(stack, nextState)
195             }
196         }
197     }
198
199     graph += "}\n"
200     return graph
201 }
202
203 func (a *NFA) StateByID(stateID int) *State {
204     visited := make(map[int]bool)
205     stack := []*State{a.Start}
206
207     for len(stack) > 0 {
208         state := stack[len(stack)-1]
209         stack = stack[:len(stack)-1]
210
211         if visited[state.ID] {
212             continue
```

```
213         }
214         visited[state.ID] = true
215
216         if state.ID == stateID {
217             return state
218         }
219
220         for _, nextStates := range state.Transitions {
221             for _, nextState := range nextStates {
222                 stack = append(stack, nextState)
223             }
224         }
225     }
226
227     return nil
228 }
229
230 func (a *NFA) IsFinalState(states map[int]bool) bool {
231     for stateID := range states {
232         state := a.StateByID(stateID)
233         if state.IsFinal {
234             return true
235         }
236     }
237     return false
238 }
239
240 func (a *NFA) EpsilonClosure(states map[int]bool) map[int]bool {
241     closure := make(map[int]bool)
242     for stateID := range states {
243         closure[stateID] = true
244     }
245
246     stack := make([]int, 0, len(states))
247     for stateID := range states {
248         stack = append(stack, stateID)
249     }
250
251     for len(stack) > 0 {
252         currentStateID := stack[len(stack)-1]
253         stack = stack[:len(stack)-1]
254
255         state := a.StateByID(currentStateID)
256         for _, nextState := range state.Transitions[EPS] {
257             if !closure[nextState.ID] {
258                 closure[nextState.ID] = true
259                 stack = append(stack, nextState.ID)
260             }
```

```
        }
    }

    return closure
}
```

Листинг 2.6 – Код модуля *transform*

```
package transform

import (
    "strings"
    "unicode"
)

const (
    maxPriority = 4
)

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

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

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

    return result.String()
}

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

func isAlphanumeric(c byte) bool {
    r := rune(c)
    return unicode.IsLetter(r) || unicode.IsDigit(r)
}

func getOpPrecedence(r rune) int {
    specialCharsPriorityMap := map[rune]int{
```

```
41          '(': 1,
42          '|': 2,
43          '.': 3,
44          '?': 4,
45          '*': 4,
46          '+': 4,
47      }
48
49      if priority, ok := specialCharsPriorityMap[r]; ok {
50          return priority
51      }
52      return maxPriority + 1
53  }
54
55  func InfixToPostfix(infix string) string {
56      infix = insertConcatOperators(infix)
57
58      var (
59          postfix []rune
60          stack   []rune
61      )
62
63      for _, r := range infix {
64          switch r {
65          case '(':
66              stack = append(stack, r)
67          case ')':
68              for len(stack) > 0 && stack[len(stack)-1] != '(' {
69                  postfix = append(postfix, stack[len(stack)-1])
70                  stack = stack[:len(stack)-1]
71              }
72              if len(stack) > 0 {
73                  stack = stack[:len(stack)-1]
74              }
75          default:
76              for len(stack) > 0 && getOpPrecedence(stack[len(stack)-1]) >=
                    getOpPrecedence(r) {
77                  postfix = append(postfix, stack[len(stack)-1])
78                  stack = stack[:len(stack)-1]
79              }
80              stack = append(stack, r)
81          }
82
83      }
84
85      for len(stack) > 0 {
86          postfix = append(postfix, stack[len(stack)-1])
87          stack = stack[:len(stack)-1]
```

```
88      }
89
90      return string(postfix)
91  }
```

3 Контрольные вопросы

3.1 Какие из следующих множеств регулярны? Для тех, которые регулярны, напишите регулярные выражения

1. Множество цепочек с равным числом нулей и единиц
   Не является регулярным.
2. Множество цепочек из 0, 1* с четным числом нулей и нечетным числом единиц
   ((0(00)*1)(1(00)*1)*1(00)*01|(0(00)*1)(1(00)*1)*0|0(00)*01|1)
   (((10)(00)*1|0)(1(00)*1)*1(00)*01|((10)(00)*1|0)(1(00)*1)*0|(10)(00)*01|11)*
3. Множество цепочек из 0, 1*, длины которых делятся на 3
   ((0|1)(0|1)(0|1))*
4. Множество цепочек из 0, 1*, не содержащих подцепочки 101
   0*(1|00|000)*0*

3.2 Найдите праволинейные грамматики для тех множеств из вопроса 1, которые регулярны

2. Множество цепочек из 0, 1* с четным числом нулей и нечетным числом единиц
   S -> 1C|0A
   A -> 0S|1B
   B -> 1A|0C
   C -> 1S|0B|ε
3. Множество цепочек из 0, 1*, длины которых делятся на 3
   S -> 0A|1A|ε
   A -> 0B|1B
   B -> 0S|1S
4. Множество цепочек из 0, 1*, не содержащих подцепочки 101
   S -> 0S|εA
   A -> 1A|00A|000A|εB
   B -> 0B|ε

3.3 Найдите детерминированные и недетерминированные конечные автоматы для тех множеств из вопроса 1, которые регулярны

2. Множество цепочек из 0, 1* с четным числом нулей и нечетным числом единиц
   (a) НКА

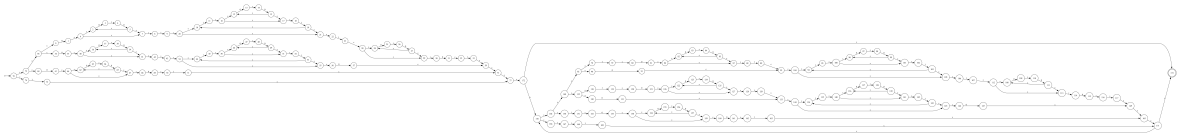

Рисунок 3.1

   (b) ДКА

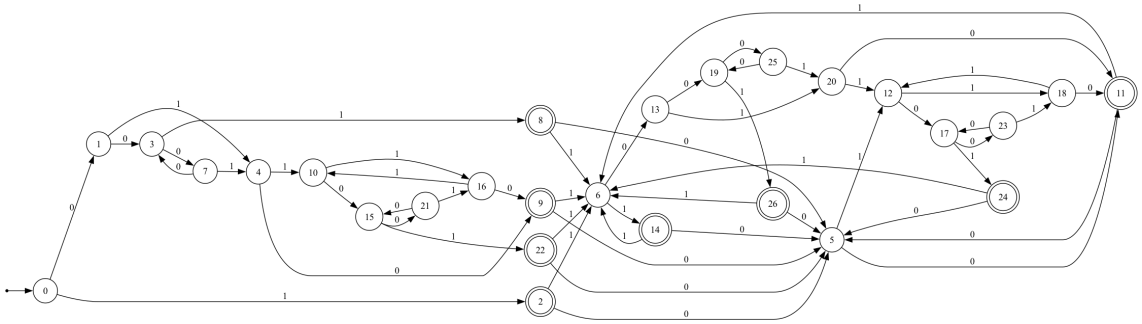

Рисунок 3.2

3. Множество цепочек из 0, 1*, длины которых делятся на 3
   (a) НКА

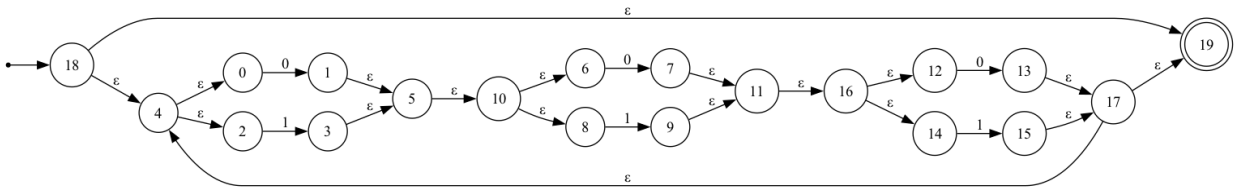

Рисунок 3.3

   (b) ДКА

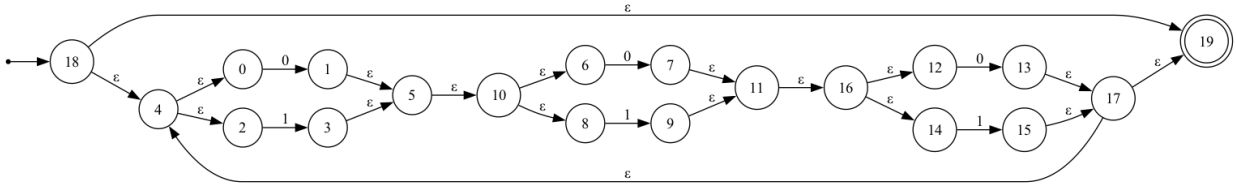

Рисунок 3.4

4. Множество цепочек из 0, 1*, не содержащих подцепочки 101

(a) НКА

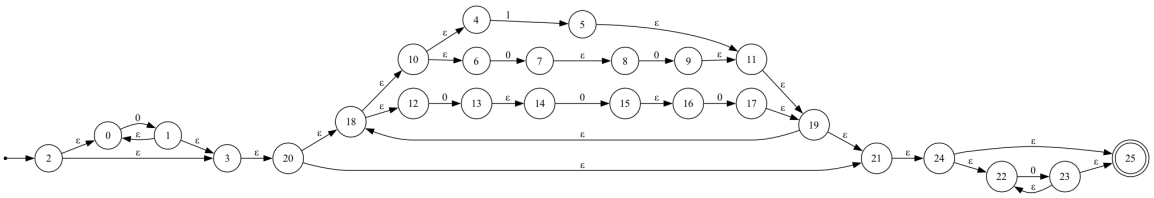

Рисунок 3.5

(b) ДКА

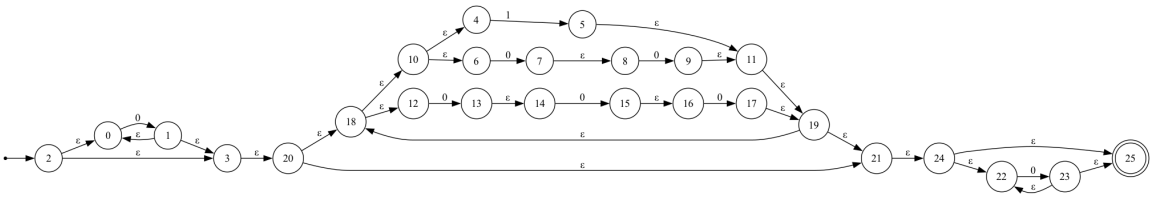

Рисунок 3.6

3.4 Найдите конечный автомат с минимальным числом состояний для языка, определяемого автоматом M = ({A, B, C, D, E}, {0, 1}, d, A, {E, F}), где функция задается таблицей

Таблица 3.1

Состояние	**Вход**	
	0	**1**
A	B	C
B	E	F
C	A	A
D	F	E
E	D	F
F	D	E

Исходный конечный автомат:

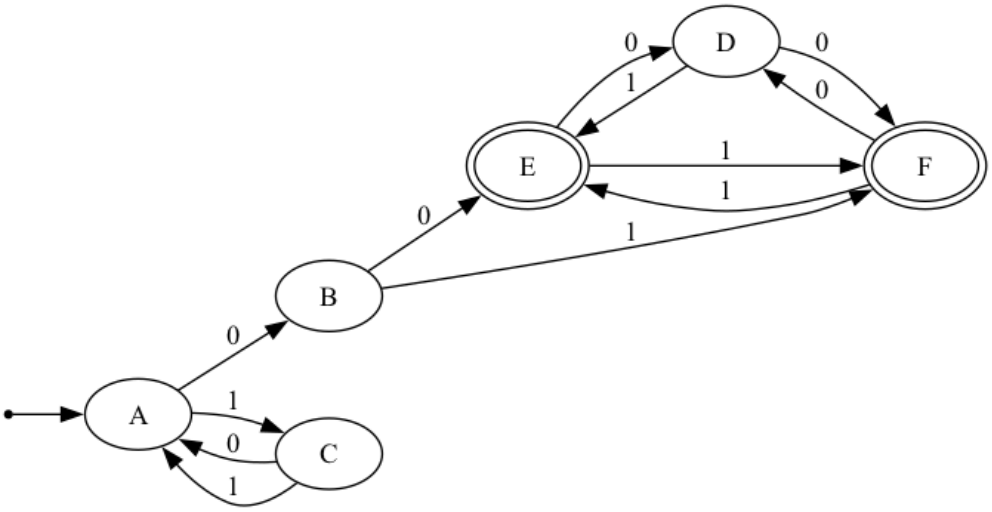

Рисунок 3.7 – Конечный автомат

Классы 0-эквивалентности:

$$\{A, B, C, D\}, \{E, F\}.$$

Классы 1-эквивалентности:

$$\{A, C\}, \{B, D\}, \{E, F\}.$$

Классы 2-эквивалентности:

$$\{A\}, \{C\}, \{B, D\}, \{E, F\}.$$

Классы 3-эквивалентности:

$$\{A\}, \{C\}, \{B, D\}, \{E, F\}.$$

Классы 3-эквивалентности и 2-эквивалентности совпадают. Таким образом, в минимизированном конечном автомате будет 4 состояния:

$$\{A\}, \{C\}, \{B, D\}, \{E, F\}.$$

Таким образом, минимизированный конечный автомат:

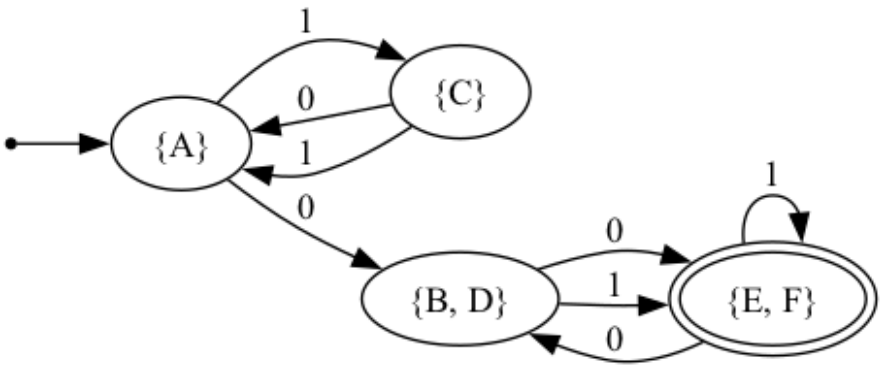

Рисунок 3.8 – Минимизированный конечный автомат