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 № 7

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

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

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. Преобразует регулярное выражение непосредственно в ДКА.

2. 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=Алгоритм_Бржозовского*

3. Моделирует минимальный КА для входной цепочки из терминалов исходной грамматики.

2 Практическая часть

2.1 Листинг

Листинг 2.1 – Исходный код построения AST

```
type AST struct {
    root        Node
    symbols     []rune
    symbolsMap map[int]*Symbol
}

func NewAST(regexp *regexp.Regexp) (*AST, error) {
    slog.Info("start tokenize")

    tokens, err := regexp.Tokenize()
    if err != nil {
        return nil, fmt.Errorf("tokenize: %w", err)
    }

    symbols := getSymbolsList(tokens)

    slog.Info("start building AST by tokens")

    parser := newParser(tokens)
    root, err := parser.parseAlternation()
    if err != nil {
        return nil, fmt.Errorf("parse: %w", err)
    }

    slog.Info("counting follow pos")

    parser.countFollowPos(root)
    symbolsMap := parser.GetLeavesMap()

    slog.Info("built AST")

    return &AST{
        root:       root,
        symbols:    symbols,
        symbolsMap: symbolsMap,
    }, nil
```

```
37 }
38
39 type parser struct {
40     index      int
41     leavesMap  map[int]*Symbol
42     tokens     []common.Token
43     pos        int
44     current    common.Token
45 }
46
47 func newParser(tokens []common.Token) *parser {
48     return &parser{
49         index:      0,
50         leavesMap: make(map[int]*Symbol, len(tokens)),
51         tokens:     tokens,
52         pos:        0,
53         current:    tokens[0],
54     }
55 }
56
57 func (p *parser) GetLeavesMap() map[int]*Symbol {
58     return p.leavesMap
59 }
60
61 func (p *parser) next() {
62     p.pos++
63     if p.pos < len(p.tokens) {
64         p.current = p.tokens[p.pos]
65     } else {
66         p.current = common.Token{common.EOF, 0}
67     }
68 }
69
70 func (p *parser) countFollowPos(root Node) {
71     switch node := root.(type) {
72     case *Concatenation:
73         for _, v := range node.Left.GetLastPos() {
74             leaf := p.leavesMap[v]
75             leaf.FollowPos =
                    set.New(leaf.FollowPos...).Union(set.New(node.Right.Get
76         }
```

```
77          p.countFollowPos(node.Left)
78          p.countFollowPos(node.Right)
79      case *KleeneStar:
80          for _, v := range node.GetLastPos() {
81              leaf := p.leavesMap[v]
82              leaf.FollowPos =
                    set.New(leaf.FollowPos...).Union(set.New(node.GetFirstP
83          }
84          p.countFollowPos(node.Child)
85      case *Alternation:
86          p.countFollowPos(node.Left)
87          p.countFollowPos(node.Right)
88      default:
89          return
90      }
91  }
92
93  func (p *parser) parseAlternation() (Node, error) {
94      node, err := p.parseConcatenation()
95      if err != nil {
96          return nil, fmt.Errorf("invalid concat: %w", err)
97      }
98
99      for p.current.Type == common.Pipe {
100         slog.Info("found alternation")
101
102         p.next()
103         right, err := p.parseConcatenation()
104         if err != nil {
105             return nil, fmt.Errorf("invalid right concat: %w",
                    err)
106         }
107
108         node = &Alternation{
109             Left:  node,
110             Right: right,
111         }
112     }
113
114     return node, nil
115 }
```

```
116
117 func (p *parser) parseConcatenation() (Node, error) {
118     node, err := p.parseQuantifier()
119     if err != nil {
120         return nil, fmt.Errorf("invalid term: %w", err)
121     }
122
123     for {
124         switch p.current.Type {
125         case common.Symbol, common.LParen:
126             slog.Info("found concatenation")
127
128             right, err := p.parseQuantifier()
129             if err != nil {
130                 return nil, fmt.Errorf("invalid right term: %w",
                        err)
131             }
132
133             node = &Concatenation{
134                 Left:  node,
135                 Right: right,
136             }
137         default:
138             return node, nil
139         }
140     }
141 }
142
143 func (p *parser) parseQuantifier() (Node, error) {
144     node, err := p.parseSymbolOrGroup()
145     if err != nil {
146         return nil, fmt.Errorf("invalid factor: %w", err)
147     }
148
149     for {
150         switch p.current.Type {
151         case common.KleeneStar:
152             slog.Info("found kleene star")
153
154             p.next()
155             node = &KleeneStar{
```

```
156                 Child: node,
157             }
158         default:
159             return node, nil
160         }
161     }
162 }
163
164 func (p *parser) parseSymbolOrGroup() (Node, error) {
165     switch p.current.Type {
166     case common.Symbol:
167         slog.Info("found symbol", slog.String("symbol",
              string(p.current.Value)))
168
169         sym := &Symbol{
170             Index: p.index,
171             Value: p.current.Value,
172         }
173         p.leavesMap[p.index] = sym
174         p.index++
175         p.next()
176         return sym, nil
177     case common.LParen:
178         slog.Info("found left parenthese")
179
180         p.next()
181         node, err := p.parseAlternation()
182         if err != nil {
183             return nil, fmt.Errorf("invalid alt: %w", err)
184         }
185
186         if p.current.Type != common.RParen {
187             return nil, fmt.Errorf("invalid token: %w",
                  common.ErrUnclosedParen)
188         }
189
190         slog.Info("found right parenthese")
191
192         p.next()
193         return node, nil
194     default:
```

```
195             return nil, fmt.Errorf("invalid token: %s",
                    string(p.current.Value))
196         }
197     }
```

Листинг 2.2 – Исходный построения ДКА

```
1   type DFA struct {
2       States []*State
3       Tran   map[string]map[rune]*State
4   }
5
6   func NewDFA(ast *ast.AST) *DFA {
7       return buildDFA(ast)
8   }
9
10  func buildDFA(ast *ast.AST) *DFA {
11      root := ast.GetRoot()
12      first := &State{State: root.GetFirstPos(), Start: true}
13      states := []*State{first}
14      tran := make(map[string]map[rune]*State, 0)
15      symbolsList := ast.GetSymbols()
16      symbolsMap := ast.GetSymbolsMap()
17
18      slog.Info("marking if last", slog.String("state",
            first.String()))
19
20      for _, st := range first.State {
21          v, ok := symbolsMap[st]
22          if !ok {
23              panic("symbols incorrect")
24          }
25
26          if v.Value == '#' {
27              first.Last = true
28              break
29          }
30      }
31
32      slog.Info("starting unmarked states cycle")
33
34      for {
35          var s *State
```

```
36          for i := range states {
37              if !states[i].Marked {
38                  s = states[i]
39                  break
40              }
41          }
42
43          if s == nil {
44              break
45          }
46
47          slog.Info("found unmarked", slog.String("unmarked",
                s.String()))
48
49          slices.Sort(s.State)
50          s.Marked = true
51          for _, a := range symbolsList {
52              u := set.New[int]()
53              for _, p := range s.State {
54                  v, ok := symbolsMap[p]
55                  if !ok {
56                      panic("symbols incorrect")
57                  }
58
59                  if v.Value != a {
60                      continue
61                  }
62
63                  u = u.Union(set.New(v.GetFollowPos()...))
64              }
65
66              if u.Len() == 0 {
67                  continue
68              }
69
70              state := &State{State: u.SortedList()}
71
72              slog.Info("got follow pos union, marking if last",
                    slog.String("union", state.String()))
73
74              for _, st := range state.State {
```

```
75                  v, ok := symbolsMap[st]
76                  if !ok {
77                      panic("symbols incorrect")
78                  }
79
80                  if v.Value == '#' {
81                      state.Last = true
82                      break
83                  }
84              }
85
86              contains := false
87              for _, v := range states {
88                  if v.Equal(state) {
89                      contains = true
90                      break
91                  }
92              }
93              if !contains {
94                  slog.Info("adding new state",
                       slog.String("state", state.String()))
95                  states = append(states, state)
96              }
97
98              tr, ok := tran[s.String()]
99              if !ok {
100                 tr = map[rune]*State{a: state}
101             } else {
102                 tr[a] = state
103             }
104             tran[s.String()] = tr
105
106             slog.Info("adding new tran", slog.String("from",
                   s.String()), slog.String("by", string(a)),
                   slog.String("to", state.String()))
107         }
108     }
109
110     return &DFA{
111         States: states,
112         Tran:   tran,
```

```
113     }
114 }
115
116 func getSymbolsList(tokens []common.Token) []rune {
117     symbols := make([]rune, 0, len(tokens))
118     for _, t := range tokens {
119         if t.Type == common.Symbol {
120             symbols = append(symbols, t.Value)
121         }
122     }
123
124     return symbols
125 }
126
127 func (a *AST) GetRoot() Node {
128     return a.root
129 }
130
131 func (a *AST) GetSymbols() []rune {
132     return a.symbols
133 }
134
135 func (a *AST) GetSymbolsMap() map[int]*Symbol {
136     return a.symbolsMap
137 }
138
139 func (a *AST) Print() {
140     printAST(a.root, "", true)
141 }
142
143 type Node interface {
144     String() string
145     IsNullable() bool
146     GetFirstPos() []int
147     GetLastPos() []int
148 }
149
150 type Concatenation struct {
151     Nullable *bool
152     FirstPos []int
153     LastPos  []int
```

```
154
155     Left  Node
156     Right Node
157 }
158
159 func (c *Concatenation) IsNullable() bool {
160     if c.Nullable != nil {
161         return *c.Nullable
162     }
163
164     nullable := c.Left.IsNullable() && c.Right.IsNullable()
165     c.Nullable = &nullable
166
167     return nullable
168 }
169
170 func (c *Concatenation) GetFirstPos() []int {
171     if c.FirstPos != nil {
172         return c.FirstPos
173     }
174
175     if c.Left.IsNullable() {
176         c.FirstPos =
               set.New(c.Left.GetFirstPos()...).Union(set.New(c.Right.GetF
177     } else {
178         c.FirstPos =
               set.New(c.Left.GetFirstPos()...).SortedList()
179     }
180
181     return c.FirstPos
182 }
183
184 func (c *Concatenation) GetLastPos() []int {
185     if c.LastPos != nil {
186         return c.LastPos
187     }
188
189     if c.Right.IsNullable() {
190         c.LastPos =
               set.New(c.Right.GetLastPos()...).Union(set.New(c.Left.GetLa
191     } else {
```

```
192             c.LastPos = set.New(c.Right.GetLastPos()...).SortedList()
193         }
194
195         return c.LastPos
196     }
197
198     func (c *Concatenation) String() string {
199         return ""
200     }
201
202     type Alternation struct {
203         Nullable *bool
204         FirstPos []int
205         LastPos  []int
206
207         Left  Node
208         Right Node
209     }
210
211     func (a *Alternation) IsNullable() bool {
212         if a.Nullable != nil {
213             return *a.Nullable
214         }
215
216         nullable := a.Left.IsNullable() || a.Right.IsNullable()
217         a.Nullable = &nullable
218
219         return nullable
220     }
221
222     func (a *Alternation) GetFirstPos() []int {
223         if a.FirstPos != nil {
224             return a.FirstPos
225         }
226
227         a.FirstPos =
               set.New(a.Left.GetFirstPos()...).Union(set.New(a.Right.GetFirst
228
229         return a.FirstPos
230     }
231
```

```
232 func (a *Alternation) GetLastPos() []int {
233     if a.LastPos != nil {
234         return a.LastPos
235     }
236
237     a.LastPos =
        set.New(a.Left.GetLastPos()...).Union(set.New(a.Right.GetLastPo
238
239     return a.LastPos
240 }
241
242 func (a *Alternation) String() string {
243     return "|"
244 }
245
246 type KleeneStar struct {
247     FirstPos []int
248     LastPos  []int
249
250     Child Node
251 }
252
253 func (k *KleeneStar) IsNullable() bool { return true }
254
255 func (k *KleeneStar) GetFirstPos() []int {
256     if k.FirstPos != nil {
257         return k.FirstPos
258     }
259
260     k.FirstPos = k.Child.GetFirstPos()
261
262     return k.FirstPos
263 }
264
265 func (k *KleeneStar) GetLastPos() []int {
266     if k.LastPos != nil {
267         return k.LastPos
268     }
269
270     k.LastPos = k.Child.GetLastPos()
271
```

```
272     return k.LastPos
273 }
274
275 func (k *KleeneStar) String() string {
276     return "*"
277 }
278
279 type Symbol struct {
280     FirstPos  []int
281     LastPos   []int
282     FollowPos []int
283
284     Index int
285
286     Value rune
287 }
288
289 func (s *Symbol) IsNullable() bool { return false }
290
291 func (s *Symbol) GetFirstPos() []int {
292     if s.FirstPos != nil {
293         return s.FirstPos
294     }
295
296     s.FirstPos = []int{s.Index}
297
298     return s.FirstPos
299 }
300
301 func (s *Symbol) GetLastPos() []int {
302     if s.LastPos != nil {
303         return s.LastPos
304     }
305
306     s.LastPos = []int{s.Index}
307
308     return s.LastPos
309 }
310
311 func (s *Symbol) GetFollowPos() []int {
312     return s.FollowPos
```

```
313 }
314
315 func (s *Symbol) String() string {
316     return string(s.Value)
317 }
```

Листинг 2.3 – Исходный минимизации ДКА

```
1  func (n *NFA) GetStartStates() []*State {
2      states := make([]*State, 0, len(n.States))
3      for _, s := range n.States {
4          if s.Start {
5              states = append(states, s)
6          }
7      }
8
9      return states
10 }
11
12 func (n *NFA) GetByIndexes(s *State) []*State {
13     states := make([]*State, 0, len(s.State))
14     for _, v := range s.State {
15         states = append(states, n.States[v])
16     }
17
18     return states
19 }
20
21 func (n *NFA) Move(states []*State, a rune) []*State {
22     u := make([]*State, 0, len(states))
23     for _, in := range states {
24         tr, ok := n.Tran[in.String()]
25         if !ok {
26             continue
27         }
28
29         end, ok := tr[a]
30         if !ok {
31             continue
32         }
33
34         u = Union(u, end)
35     }
```

```
36
37      return u
38  }
39
40  func (n *NFA) EpsClosure(s []*State) []int {
41      closure := make([]int, 0, len(s))
42      for _, in := range s {
43          for j, st := range n.States {
44              if in.Equal(st) {
45                  closure = append(closure, j)
46                  break
47              }
48          }
49      }
50
51      return closure
52  }
53
54  func NewNFA(d *DFA) *NFA {
55      nfa := &NFA{
56          States: make([]*State, 0, len(d.States)),
57          Tran:   make(map[string]map[rune][]*State, len(d.Tran)),
58      }
59
60      slog.Info("start reversing DFA")
61
62      for _, s := range d.States {
63          last := false
64          start := false
65
66          if s.Last {
67              start = true
68          }
69          if s.Start {
70              last = true
71          }
72          state := &State{
73              State:  slices.Clone(s.State),
74              Marked: s.Marked,
75              Last:   last,
76              Start:  start,
```

```
77          }
78
79          nfa.States = append(nfa.States, state)
80      }
81
82      slog.Info("got NFA states", slog.Int("count",
            len(nfa.States)))
83
84      for stateStart, tran := range d.Tran {
85          var state *State
86          for _, s := range nfa.States {
87              if stateStart == s.String() {
88                  state = &State{
89                      State:  slices.Clone(s.State),
90                      Marked: s.Marked,
91                      Last:   s.Last,
92                      Start:  s.Start,
93                  }
94                  break
95              }
96          }
97
98          if state == nil {
99              continue
100         }
101
102         for sym, stateEnd := range tran {
103             tr, ok := nfa.Tran[stateEnd.String()]
104             if !ok {
105                 tr = map[rune][]*State{sym: []*State{state}}
106             } else {
107                 tr[sym] = append(tr[sym], state)
108             }
109             nfa.Tran[stateEnd.String()] = tr
110         }
111     }
112
113     slog.Info("got NFA transitions", slog.Int("count",
            len(nfa.Tran)))
114
115     return nfa
```

```
116 }
117
118 func (n *NFA) Determine(ast *ast.AST) *DFA {
119     symbolsList := ast.GetSymbols()
120     dfa := &DFA{
121         States: make([]*State, 0, len(n.States)),
122         Tran:   make(map[string]map[rune]*State, len(n.Tran)),
123     }
124
125     slog.Info("start determine NFA")
126
127     s0EpcClosure := n.EpsClosure(n.GetStartStates())
128     first := &State{
129         State:  s0EpcClosure,
130         Marked: false,
131         Last:   false,
132         Start:  true,
133     }
134     dfa.States = append(dfa.States, first)
135
136     slog.Info("got first state (indexes of NFA states)",
        slog.String("state", first.String()))
137
138     nStates := n.GetByIndexes(first)
139     for _, s := range nStates {
140         if s.Last {
141             first.Last = true
142             break
143         }
144     }
145
146     slog.Info("starting unmarked states cycle")
147
148     for {
149         var s *State
150         for i := range dfa.States {
151             if !dfa.States[i].Marked {
152                 s = dfa.States[i]
153                 break
154             }
155         }
```

```
156
157         if s == nil {
158             break
159         }
160
161         slog.Info("found unmarked", slog.String("unmarked",
              s.String()))
162
163         slices.Sort(s.State)
164         s.Marked = true
165         for _, a := range symbolsList {
166             u := n.EpsClosure(n.Move(n.GetByIndexes(s), a))
167             if len(u) == 0 {
168                 continue
169             }
170
171             slog.Info("calculated union eps-closure",
                  slog.String("ec", fmt.Sprintf("%v", u)))
172
173             contains := false
174             for _, v := range dfa.States {
175                 slices.Sort(v.State)
176                 slices.Sort(u)
177                 if slices.Equal(v.State, u) {
178                     contains = true
179                     break
180                 }
181             }
182             uState := &State{
183                 State:  u,
184                 Marked: false,
185                 Last:   false,
186                 Start:  false,
187             }
188
189             nStates := n.GetByIndexes(uState)
190             for _, s := range nStates {
191                 if s.Last {
192                     uState.Last = true
193                     break
194                 }
```

```
195             }
196
197             if !contains {
198                 slog.Info("adding new state",
                       slog.String("state", uState.String()))
199                 dfa.States = append(dfa.States, uState)
200             }
201
202             tr, ok := dfa.Tran[s.String()]
203             if !ok {
204                 tr = map[rune]*State{a: uState}
205             } else {
206                 tr[a] = uState
207             }
208             dfa.Tran[s.String()] = tr
209
210             slog.Info("adding new tran", slog.String("from",
                   s.String()), slog.String("by", string(a)),
                   slog.String("to", uState.String()))
211         }
212     }
213
214     return dfa
215 }
```

Листинг 2.4 – Исходный код моделирования ДКА

```
1  func (d *DFA) Model(in string) bool {
2      fmt.Println(commentC, "START MODELING FOR", in, endC)
3
4      curState := d.GetStart()
5
6      way := &way{
7          steps: []*step{
8              {
9                  symbol: "",
10                 dst:    curState.String(),
11                 border: true,
12             },
13         },
14     }
15
16     for _, s := range in {
```

```
17          slog.Info("current state", slog.String("state",
                curState.String()))
18
19          tr, ok := d.Tran[curState.String()]
20          if !ok {
21              slog.Error("can't find transition starting with such
                    state", slog.Any("valid options",
                    maps.Keys(d.Tran)))
22              way.Show()
23              return false
24          }
25
26          slog.Info("found transition with such start state",
                slog.String("from", curState.String()))
27
28          next, ok := tr[s]
29          if !ok {
30              slog.Error("can't find transition by such symbol",
                    slog.String("symbol", string(s)),
31                   slog.Any("valid options", maps.Keys(tr)))
32              way.Show()
33              return false
34          }
35
36          slog.Info("found transition by such symbol",
                slog.String("from", curState.String()),
37              slog.String("by", string(s)), slog.String("to",
                    next.String()))
38
39          way.steps = append(way.steps, &step{
40              symbol: string(s),
41              dst:    next.String(),
42          })
43
44          curState = next
45      }
46
47      if !curState.Last {
48          slog.Error("end state isn't last", slog.String("state",
                curState.String()),
49              slog.String("last", d.GetLast().String()))
```

```
50          way.Show()
51          return false
52      }
53
54      way.steps[len(way.steps)-1].border = true
55      way.Show()
56
57      return true
58  }
```

2.2 Результаты выполнения программы

Таблица 2.1 – Результаты выполнения программы

Регулярное выражение	**Входная строка**	**Результат**
a	пустая строка	FAIL
	a	OK
	aa	FAIL
	b	FAIL
ab(ab|b*)*	пустая строка	FAIL
	abbb	OK
	ab	OK
	bbb	FAIL
a*|(bb*|c*a)	b	OK
	a	OK
	ccccca	OK
	cccab	FAIL
	пустая строка	OK

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

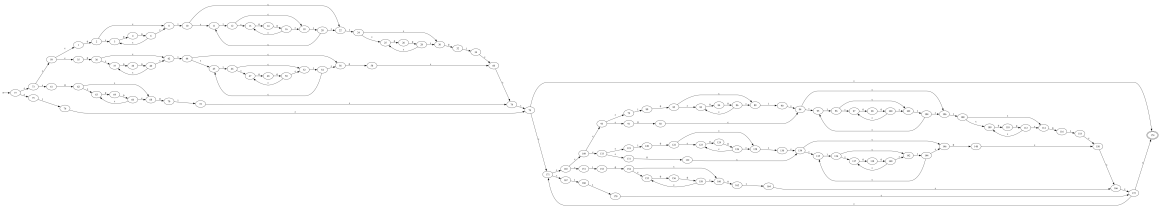

Рисунок 3.1 – НКА

   (b) ДКА

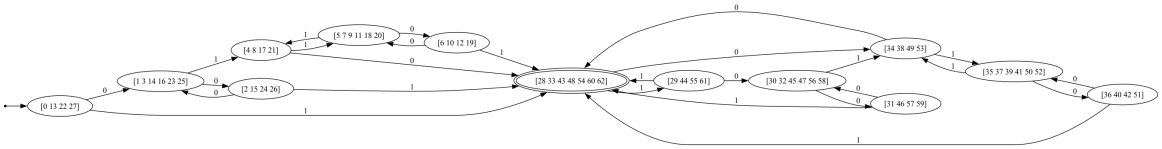

Рисунок 3.2 – ДКА

3. Множество цепочек из 0, 1*, длины которых делятся на 3

   (a) НКА

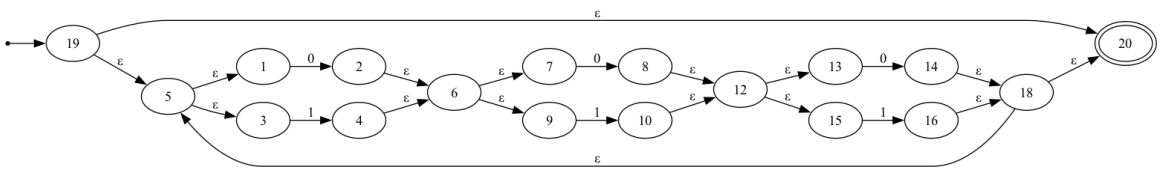

Рисунок 3.3 – НКА

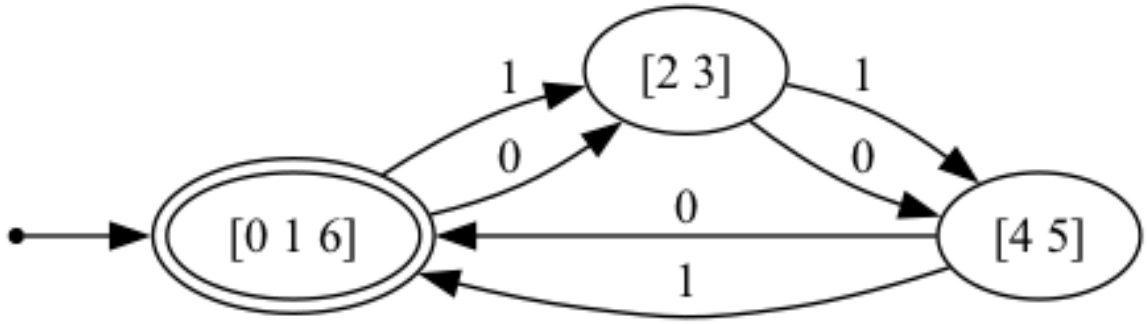

Рисунок 3.4 – ДКА

(b) ДКА

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

(a) НКА

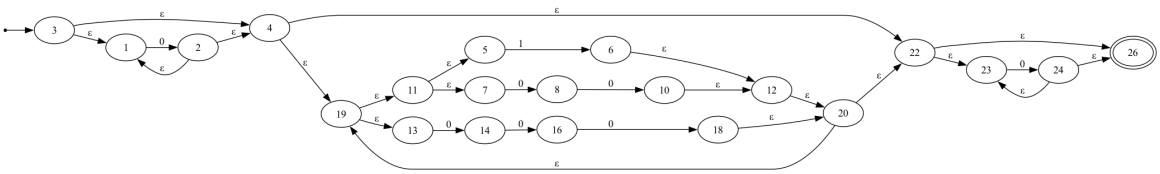

Рисунок 3.5 – НКА

(b) ДКА

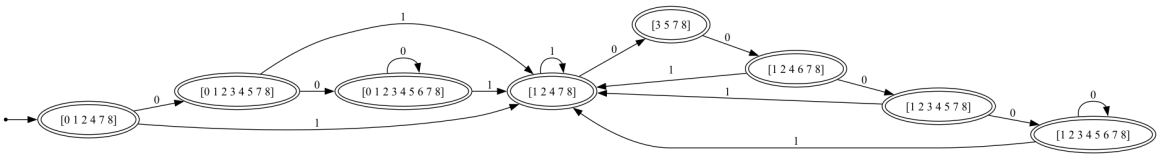

Рисунок 3.6 – ДКА

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:

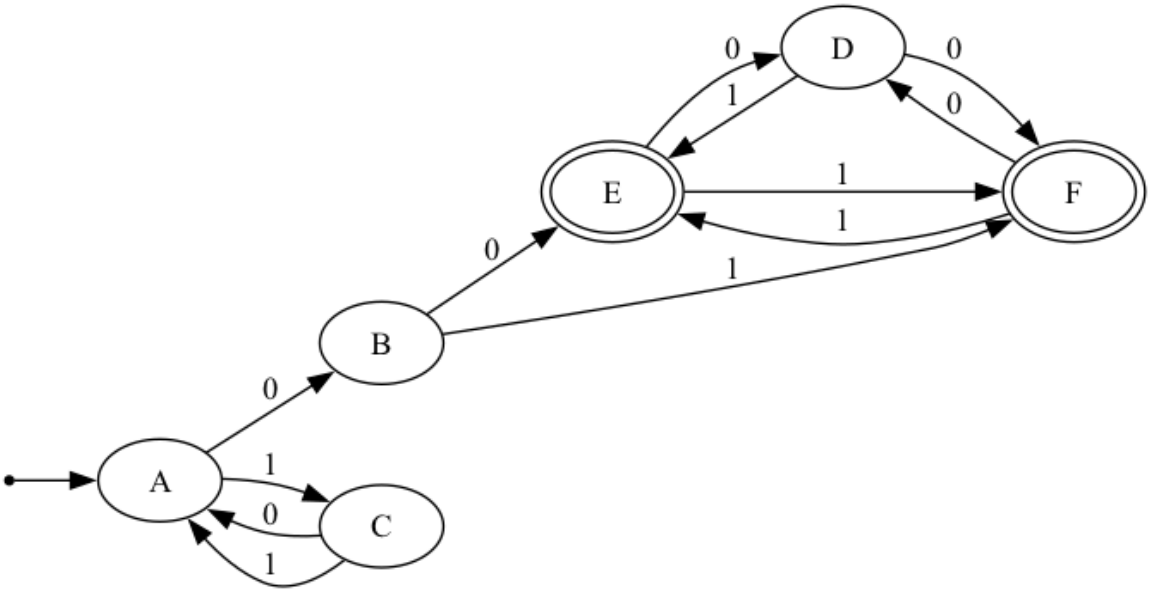

Рисунок 3.7 – Конечный автомат

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:

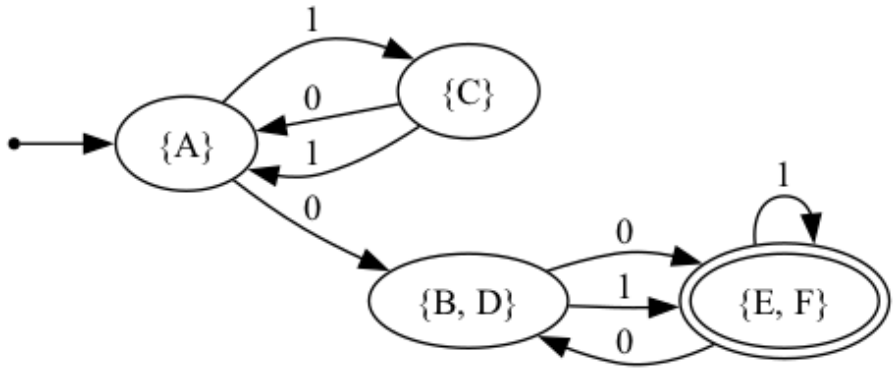

Рисунок 3.8 – Минимизированный конечный автомат