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 №2

по курсу «Конструирование компиляторов»

на тему: «Преобразования грамматик»

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 наиболее важных (но не всех) видов преобразований грамматик, чтобы удовлетворить требованиям алгоритмов синтаксического разбора.

Задачи работы

1) Принять к сведению соглашения об обозначениях, принятые в литературе по теории формальных языков и грамматик и кратко описанные в приложении.

2) Познакомиться с основными понятиями и определениями теории формальных языков и грамматик.

3) Детально разобраться в алгоритме устранения левой рекурсии.

4) Разработать, тестировать и отладить программу устранения левой рекурсии.

5) Разработать, тестировать и отладить программу преобразования грамматики в соответствии с предложенным вариантом.

Содержание работы (Вариант 4)

1. Постройте программу, которая в качестве входа принимает приведенную КС-грамматику $G = (N, \Sigma, P, S)$ и преобразует ее в эквивалентную КС-грамматику G' без левой рекурсии.

2. Постройте программу, которая в качестве входа принимает приведенную КС-грамматику $G = (N, \Sigma, P, S)$ без ε-правил и преобразует ее в эквивалентную КС-грамматику G' без ε-правил и цепных правил.

РЕЗУЛЬТАТЫ ВЫПОЛНЕНИЯ ПРОГРАММЫ

Листинг 1 – Устранение левой рекурсии

```
-- in

2
S A
4
a b c d
5
S -> Aa
S -> b
A -> Ac
A -> Sd
A -> eps
S

-- out
2
A S
4
a b c d
4
A -> Ac
A -> Sd
S -> Aa
S -> b
S
```

Листинг 2 – Устранение цепных правил

```
-- in

3
S A B
2
a b
5
S -> A
A -> B
B -> a
B -> b
A -> a
```

```
S


-- out
3
A B S
2
a b
6
A -> a
A -> b
B -> a
B -> b
S -> a
S -> b
S
```

ЗАКЛЮЧЕНИЕ

В ходе выполнения лабораторной работы была реализована программа на языке Go, позволяющая преобразовывать грамматики.

В программе было реализовано устранение левой рекурсии совместно с устранением непосредственной и косвенной рекурсии, после чего применялась левая факторизация. Также было реализовано устранение цепных правил.

Таким образом, в результате выполнения лабораторной работы были приобретены практические навыки реализации наиболее важных (но не всех) видов преобразований грамматик, чтобы удовлетворить требованиям алгоритмов синтаксического разбора.

ПРИЛОЖЕНИЕ А

Листинг А.1 – Модуль grammar

```go
package grammar

import (
"fmt"
"sort"
"strconv"
"strings"
)

type Grammar struct {
  NonTerminals    []string
  Terminals       []string
  ProductionRules []ProductionRule
  StartSymbol     string
  productionMap   map[string][]string
}

type ProductionRule struct {
  Left  string
  Right string
}

func NewGrammarFromString(input string) (*Grammar, error) {
  lines := strings.Split(strings.TrimSpace(input), "\n")
  if len(lines) < 7 {
    return nil, fmt.Errorf("%d)", len(lines))
  }

  var cleanLines []string
  for _, line := range lines {
    trimmed := strings.TrimSpace(line)
    if trimmed != "" {
      cleanLines = append(cleanLines, trimmed)
    }
  }

  g := &Grammar{productionMap: make(map[string][]string)}
```

```
38
39    nonTerminalCount, err := strconv.Atoi(cleanLines[0])
40    if err != nil {
41      return nil, fmt.Errorf("", err)
42    }
43
44    g.NonTerminals = strings.Fields(cleanLines[1])
45    if len(g.NonTerminals) != nonTerminalCount {
46      return nil, fmt.Errorf("",
47      nonTerminalCount, len(g.NonTerminals))
48    }
49
50    terminalCount, err := strconv.Atoi(cleanLines[2])
51    if err != nil {
52      return nil, fmt.Errorf("", err)
53    }
54
55    g.Terminals = strings.Fields(cleanLines[3])
56    if len(g.Terminals) != terminalCount {
57      return nil, fmt.Errorf("",
58      terminalCount, len(g.Terminals))
59    }
60
61    ruleCount, err := strconv.Atoi(cleanLines[4])
62    if err != nil {
63      return nil, fmt.Errorf("%v", err)
64    }
65
66    ruleStart := 5
67    ruleEnd := ruleStart + ruleCount
68    if ruleEnd > len(cleanLines)-1 {
69      return nil, fmt.Errorf("",
70      ruleCount, len(cleanLines)-ruleStart-1)
71    }
72
73    for i := ruleStart; i < ruleEnd; i++ {
74      parts := strings.Split(cleanLines[i], "->")
75      if len(parts) != 2 {
76        return nil, fmt.Errorf("", cleanLines[i])
77      }
78      rule := ProductionRule{
```

```
79          Left:  strings.TrimSpace(parts[0]),
80          Right: strings.TrimSpace(parts[1]),
81        }
82        g.ProductionRules = append(g.ProductionRules, rule)
83        g.productionMap[rule.Left] = append(g.productionMap[rule.Left],
              rule.Right)
84      }
85
86      g.StartSymbol = strings.TrimSpace(cleanLines[ruleEnd])
87      found := false
88      for _, nt := range g.NonTerminals {
89        if nt == g.StartSymbol {
90          found = true
91          break
92        }
93      }
94      if !found {
95        return nil, fmt.Errorf("", g.StartSymbol)
96      }
97
98      return g, nil
99  }
100
101 func (g *Grammar) ToString() string {
102     var builder strings.Builder
103
104     sortedNonTerminals := append([]string{}, g.NonTerminals...)
105     sort.Strings(sortedNonTerminals)
106     builder.WriteString(fmt.Sprintf("%d\n", len(sortedNonTerminals)))
107     builder.WriteString(strings.Join(sortedNonTerminals, " ") + "\n")
108
109     sortedTerminals := append([]string{}, g.Terminals...)
110     sort.Strings(sortedTerminals)
111     builder.WriteString(fmt.Sprintf("%d\n", len(sortedTerminals)))
112     builder.WriteString(strings.Join(sortedTerminals, " ") + "\n")
113
114     rulesCopy := append([]ProductionRule{}, g.ProductionRules...)
115     sort.Slice(rulesCopy, func(i, j int) bool {
116       if rulesCopy[i].Left == rulesCopy[j].Left {
117         return rulesCopy[i].Right < rulesCopy[j].Right
118       }
```

```
119     return rulesCopy[i].Left < rulesCopy[j].Left
120   })
121
122   builder.WriteString(fmt.Sprintf("%d\n", len(rulesCopy)))
123   for _, rule := range rulesCopy {
124     builder.WriteString(fmt.Sprintf("%s -> %s\n", rule.Left, rule.
          Right))
125   }
126
127   builder.WriteString(g.StartSymbol + "\n")
128
129   return builder.String()
130 }
131
132 func (g *Grammar) GetProductionsFor(nonTerminal string) []string {
133   return g.productionMap[nonTerminal]
134 }
135
136 func (g *Grammar) IsNonTerminal(symbol string) bool {
137   for _, nt := range g.NonTerminals {
138     if nt == symbol {
139       return true
140     }
141   }
142   return false
143 }
144
145 func (g *Grammar) IsTerminal(symbol string) bool {
146   for _, t := range g.Terminals {
147     if t == symbol {
148       return true
149     }
150   }
151   return false
152 }
153
154 func (g *Grammar) EliminateLeftRecursion() error {
155   orderedNT := g.NonTerminals
156
157   for i, Ai := range orderedNT {
158     var productionsToAdd []ProductionRule
```

```
159     var productionsToRemove []ProductionRule
160
161     for _, rule := range g.ProductionRules {
162       if rule.Left != Ai {
163         continue
164       }
165       for j := 0; j < i; j++ {
166         Aj := orderedNT[j]
167         if strings.HasPrefix(rule.Right, Aj) {
168           gamma := rule.Right[len(Aj):]
169           for _, ajRule := range g.GetProductionsFor(Aj) {
170             newRight := ajRule + gamma
171             productionsToAdd = append(productionsToAdd,
                    ProductionRule{
172               Left:  Ai,
173               Right: newRight,
174             })
175           }
176           productionsToRemove = append(productionsToRemove, rule)
177           break
178         }
179       }
180     }
181
182     g.removeProductions(productionsToRemove)
183     g.ProductionRules = append(g.ProductionRules, productionsToAdd
          ...)
184     g.rebuildProductionMap()
185
186     if err := g.eliminateImmediateLeftRecursion(Ai); err != nil {
187       return fmt.Errorf("", Ai, err)
188     }
189   }
190
191   return nil
192 }
193
194 func (g *Grammar) eliminateImmediateLeftRecursion(A string) error {
195   var recursiveRules []ProductionRule
196   var nonRecursiveRules []ProductionRule
197
```

```
198   for _, rule := range g.GetProductionsFor(A) {
199     if strings.HasPrefix(rule, A) {
200       recursiveRules = append(recursiveRules, ProductionRule{
201         Left:  A,
202         Right: rule,
203       })
204     } else {
205       nonRecursiveRules = append(nonRecursiveRules, ProductionRule{
206         Left:  A,
207         Right: rule,
208       })
209     }
210   }
211
212   if len(recursiveRules) == 0 {
213     return nil
214   }
215
216   newNonTerminal := A + "'"
217   g.NonTerminals = append(g.NonTerminals, newNonTerminal)
218
219   var newARules []ProductionRule
220   for _, rule := range nonRecursiveRules {
221     if rule.Right == "" {
222       newARules = append(newARules, ProductionRule{
223         Left:  A,
224         Right: newNonTerminal,
225       })
226     } else {
227       newARules = append(newARules, ProductionRule{
228         Left:  A,
229         Right: rule.Right + newNonTerminal,
230       })
231     }
232   }
233
234   var newAPrimeRules []ProductionRule
235   for _, rule := range recursiveRules {
236     alpha := rule.Right[len(A):]
237     newAPrimeRules = append(newAPrimeRules, ProductionRule{
238       Left:  newNonTerminal,
```

```
239         Right: alpha + newNonTerminal,
240       })
241     }
242     newAPrimeRules = append(newAPrimeRules, ProductionRule{
243       Left:  newNonTerminal,
244       Right: "",
245     })
246
247     g.removeProductionsFor(A)
248     g.ProductionRules = append(g.ProductionRules, newARules...)
249     g.ProductionRules = append(g.ProductionRules, newAPrimeRules...)
250     g.rebuildProductionMap()
251
252     return nil
253 }
254
255 func (g *Grammar) removeProductions(rules []ProductionRule) {
256     for _, rule := range rules {
257       for i, r := range g.ProductionRules {
258         if r.Left == rule.Left && r.Right == rule.Right {
259           g.ProductionRules = append(g.ProductionRules[:i], g.
                ProductionRules[i+1:]...)
260           break
261         }
262       }
263     }
264 }
265
266 func (g *Grammar) removeProductionsFor(nt string) {
267     var newRules []ProductionRule
268     for _, rule := range g.ProductionRules {
269       if rule.Left != nt {
270         newRules = append(newRules, rule)
271       }
272     }
273     g.ProductionRules = newRules
274 }
275
276 func (g *Grammar) rebuildProductionMap() {
277     g.productionMap = make(map[string][]string)
278     for _, rule := range g.ProductionRules {
```

```
279     g.productionMap[rule.Left] = append(g.productionMap[rule.Left],
            rule.Right)
280   }
281 }
282 func (g *Grammar) LeftFactor() error {
283   changed := true
284
285   for changed {
286     changed = false
287
288     for _, nt := range g.NonTerminals {
289       productions := g.GetProductionsFor(nt)
290       if len(productions) < 2 {
291         continue
292       }
293
294       prefixGroups := groupByPrefix(productions)
295       for prefix, group := range prefixGroups {
296         if prefix == "" || len(group) < 2 {
297           continue
298         }
299
300         newNT := nt + "'"
301         for contains(g.NonTerminals, newNT) {
302           newNT += "'"
303         }
304         g.NonTerminals = append(g.NonTerminals, newNT)
305
306         g.removeProductionsFor(nt)
307
308         g.ProductionRules = append(g.ProductionRules,
              ProductionRule{
309           Left:  nt,
310           Right: prefix + newNT,
311         })
312
313         for _, prod := range group {
314           suffix := prod[len(prefix):]
315           if suffix == "" {
316             suffix = ""
317           }
```

```
318             g.ProductionRules = append(g.ProductionRules,
                    ProductionRule{
319               Left:  newNT,
320               Right: suffix,
321             })
322           }
323
324           for _, prod := range productions {
325             if !strings.HasPrefix(prod, prefix) {
326               g.ProductionRules = append(g.ProductionRules,
                      ProductionRule{
327                 Left:  nt,
328                 Right: prod,
329               })
330             }
331           }
332
333           changed = true
334           break
335         }
336         if changed {
337           g.rebuildProductionMap()
338           break
339         }
340       }
341     }
342
343     return nil
344 }
345
346 func groupByPrefix(productions []string) map[string][]string {
347     prefixGroups := make(map[string][]string)
348     used := make(map[string]bool)
349
350     for i := 0; i < len(productions); i++ {
351       for j := i + 1; j < len(productions); j++ {
352         p1 := productions[i]
353         p2 := productions[j]
354         prefix := commonPrefix(p1, p2)
355         if len(prefix) > 0 && !used[prefix] {
356           prefixGroups[prefix] = append(prefixGroups[prefix], p1)
```

```
357           prefixGroups[prefix] = append(prefixGroups[prefix], p2)
358           used[prefix] = true
359         }
360       }
361     }
362
363     for k, v := range prefixGroups {
364       prefixGroups[k] = unique(v)
365       sort.Strings(prefixGroups[k])
366     }
367
368     return prefixGroups
369 }
370
371 func commonPrefix(a, b string) string {
372     i := 0
373     for i < len(a) && i < len(b) && a[i] == b[i] {
374       i++
375     }
376     return a[:i]
377 }
378
379 func unique(slice []string) []string {
380     seen := make(map[string]struct{})
381     var result []string
382     for _, val := range slice {
383       if _, ok := seen[val]; !ok {
384         seen[val] = struct{}{}
385         result = append(result, val)
386       }
387     }
388     return result
389 }
390
391 func contains(slice []string, s string) bool {
392     for _, item := range slice {
393       if item == s {
394         return true
395       }
396     }
397     return false
```

```
398 }
399
400 func (g *Grammar) EliminateChainRules() error {
401   Nsets := make(map[string]map[string]bool)
402
403   for _, A := range g.NonTerminals {
404     Nprev := make(map[string]bool)
405     Nprev[A] = true
406     changed := true
407
408     for changed {
409       changed = false
410       Ncurrent := make(map[string]bool)
411       for B := range Nprev {
412         Ncurrent[B] = true
413         for _, rule := range g.ProductionRules {
414           if rule.Left == B && g.IsNonTerminal(rule.Right) && len(
                rule.Right) == 1 {
415             C := rule.Right
416             if !Nprev[C] {
417               Ncurrent[C] = true
418               changed = true
419             }
420           }
421         }
422       }
423       Nprev = Ncurrent
424     }
425
426     Nsets[A] = Nprev
427   }
428
429   newRulesMap := make(map[ProductionRule]bool)
430
431   for _, rule := range g.ProductionRules {
432     if rule.Right == "" || (g.IsNonTerminal(rule.Right) && len(rule
          .Right) == 1) {
433       continue
434     }
435
436     for A, N_A := range Nsets {
```

```
437         if N_A[rule.Left] {
438           newRule := ProductionRule{Left: A, Right: rule.Right}
439           newRulesMap[newRule] = true
440         }
441       }
442     }
443
444     var newRules []ProductionRule
445     for rule := range newRulesMap {
446       newRules = append(newRules, rule)
447     }
448
449     g.ProductionRules = newRules
450     g.rebuildProductionMap()
451
452     return nil
453 }
```

Листинг A.2 – Тестирование модуля grammar

```
1  package grammar
2
3  import (
4  "testing"
5  )
6
7  func TestEliminateLeftRecursion(t *testing.T) {
8    tests := []struct {
9      name     string
10     input    string
11     expected string
12   }{
13     {
14       name: "",
15       input: `1
16       A
17       2
18       a b
19       2
20       A -> Aa
21       A -> b
22       A
23       `,
```

```
      expected: `2
      A A'
      2
      a b
      3
      A -> bA'
      A' -> aA'
      A' ->
      A
      `,
    },
    {
      name: "",
      input: `2
      A B
      2
      a b
      3
      A -> Ba
      B -> Ab
      B -> c
      A
      `,
      expected: `3
      A B B'
      2
      a b
      4
      A -> Ba
      B -> cB'
      B' -> abB'
      B' ->
      A
      `,
    },
    {
      name: "",
      input: `2
      S A
      3
      a b c
```

```
      6
      S -> Sa
      S -> Ab
      S -> c
      A -> Ac
      A -> Sd
      A ->
      S
      `,
      expected: `4
      A A' S S'
      3
      a b c
      9
      A -> A'
      A -> cS'dA'
      A' -> bS'dA'
      A' -> cA'
      A' ->
      S -> AbS'
      S -> cS'
      S' -> aS'
      S' ->
      S
      `,
    },
    {
      name: "",
      input: `2
      S A
      2
      a b
      3
      S -> Ab
      A -> a
      A -> b
      S
      `,
      expected: `2
      A S
      2
```

```
106         a b
107         3
108         A -> a
109         A -> b
110         S -> Ab
111         S
112         `,
113       },
114     }
115
116     for _, tt := range tests {
117       t.Run(tt.name, func(t *testing.T) {
118         g, err := NewGrammarFromString(tt.input)
119         if err != nil {
120           t.Fatal(err)
121         }
122
123         if err := g.EliminateLeftRecursion(); err != nil {
124           t.Fatal(err)
125         }
126
127         result := g.ToString()
128
129         if result != tt.expected {
130           t.Errorf("expected: %s, actual: %s", tt.expected, result)
131         }
132       })
133     }
134 }
135
136 func TestLeftFactor(t *testing.T) {
137     tests := []struct {
138       name     string
139       input    string
140       expected string
141     }{
142       {
143         name: "",
144         input: `1
145         S
146         2
```

```
147         a b
148         3
149         S -> ab
150         S -> ac
151         S -> d
152         S
153         `,
154         expected: `2
155         S S'
156         2
157         a b
158         4
159         S -> aS'
160         S -> d
161         S' -> b
162         S' -> c
163         S
164         `,
165       },
166       {
167         name: "",
168         input: `1
169         A
170         2
171         a b
172         3
173         A -> abc
174         A -> ab
175         A -> a
176         A
177         `,
178         expected: `3
179         A A' A''
180         2
181         a b
182         5
183         A -> aA''
184         A' -> c
185         A' ->
186         A'' -> bA'
187         A'' ->
```

```
188         A
189         `,
190       },
191       {
192         name: "",
193         input: `1
194         S
195         3
196         a b c
197         4
198         S -> abc
199         S -> abd
200         S -> ac
201         S -> bd
202         S
203         `,
204         expected: `3
205         S S' S''
206         3
207         a b c
208         6
209         S -> aS''
210         S -> bd
211         S' -> c
212         S' -> d
213         S'' -> bS'
214         S'' -> c
215         S
216         `,
217       },
218       {
219         name: "",
220         input: `1
221         A
222         2
223         a b
224         2
225         A -> a
226         A -> b
227         A
228         `,
```

```
229         expected: `1
230         A
231         2
232         a b
233         2
234         A -> a
235         A -> b
236         A
237         `,
238       },
239     }
240
241     for _, tt := range tests {
242       t.Run(tt.name, func(t *testing.T) {
243         g, err := NewGrammarFromString(tt.input)
244         if err != nil {
245           t.Fatal(err)
246         }
247
248         if err := g.LeftFactor(); err != nil {
249           t.Fatal(err)
250         }
251
252         result := g.ToString()
253         if result != tt.expected {
254           t.Errorf("expected: %s, actual: %s", tt.expected, result)
255         }
256
257       })
258     }
259 }
260
261 func TestEliminateChainRules(t *testing.T) {
262   tests := []struct {
263     name     string
264     input    string
265     expected string
266   }{
267     {
268       name: "",
269       input: `2
```

```
      S A
      2
      a b
      3
      S -> A
      A -> a
      A -> b
      S
      `,
      expected: `2
      A S
      2
      a b
      4
      A -> a
      A -> b
      S -> a
      S -> b
      S
      `,
    },
    {
      name: "",
      input: `3
      S A B
      2
      a b
      5
      S -> A
      A -> B
      B -> a
      B -> b
      A -> a
      S
      `,
      expected: `3
      A B S
      2
      a b
      6
      A -> a
```

```
311         A -> b
312         B -> a
313         B -> b
314         S -> a
315         S -> b
316         S
317         `,
318       },
319       {
320         name: "",
321         input: `2
322         S A
323         2
324         a b
325         3
326         S -> ab
327         A -> a
328         A -> b
329         S
330         `,
331         expected: `2
332         A S
333         2
334         a b
335         3
336         A -> a
337         A -> b
338         S -> ab
339         S
340         `,
341       },
342     }
343
344     for _, tt := range tests {
345       t.Run(tt.name, func(t *testing.T) {
346         g, err := NewGrammarFromString(tt.input)
347         if err != nil {
348           t.Fatal(err)
349         }
350
351         if err := g.EliminateChainRules(); err != nil {
```

```
352           t.Fatal(err)
353         }
354
355         result := g.ToString()
356         if result != tt.expected {
357           t.Errorf("ecpected: %s, actual: %s", tt.expected, result)
358         }
359       })
360     }
361 }
```