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 №2
по курсу «Конструирование компиляторов»

Вариант №2

Тема Преобразование грамматик

Студент Аскарян К.А.

Группа ИУ7-21М

Преподаватель Ступников А. А.

Москва — 2025 г.

1 Теоретическая часть

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. Разработать, тестировать и отладить программу распознавания цепочек регулярного или праволинейного языка в соответствии с предложенным вариантом грамматики.
4. Детально разобраться в алгоритме устранения левой рекурсии.
5. Разработать, тестировать и отладить программу устранения левой рекурсии.
6. Разработать, тестировать и отладить программу преобразования грамматики в соответствии с предложенным вариантом.

1.1 Задание

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 не леворекурсивную приведенную КС-грамматику $G = (N, \Sigma, P, S)$ и преобразует ее в эквивалентную КС-грамматику G' не содержащую бесполезных символов.

2 Практическая часть

2.1 Результат выполнения работы

В таблицфах 2.1 — 2.2 приведены результаты работы программы.

Таблица 2.1 – Тесты устранения левой рекурсии

Входная грамматика	Результат
3 E T F 5 + * ( ) a 6 E ->E + T E ->T T ->T * F T ->F F ->a F ->(E) E	5 E T F E' T' 5 + * ( ) a 8 E ->T E' E' ->+ T E' E' ->eps T' ->* F T' T' ->eps T ->F T' F ->a F ->(E) E
2 S A 4 a b c d 5 S ->A a S ->b A ->A c A ->S d A ->eps S	3 S A A' 4 a b c d 7 S ->A a S ->b A ->b d A' A ->eps A' A' ->c A' A' ->a d A' A' ->eps S

Таблица 2.2 – Тесты устранения бесполезных символов

Входная грамматика	Результат
8 S A B C D E F G 1 c 5 S ->A B S ->C D A ->E F G ->A D C ->c S	7 A B C D E F S 1 c 4 S ->A B S ->C D A ->E F C ->c S
3 S A B 2 a b 9 S ->A S ->B A ->a B A ->b S A ->b B ->A B B ->B a B ->A S B ->b S	3 A B S 2 a b 9 S ->A S ->B B ->A B B ->B a B ->A S B ->b A ->a B A ->b S A ->b S

2.2 Код программы

В листингах 2.1 — 2.5 приведен код программы на языке Go.

Листинг 2.1 – Код модуля *grammar*

```
1 const (
2     Empty = "eps"
```

```
)

type Grammar struct {
    NonTerminals []string
    Terminals    []string
    Start        string
    Rules        map[string][][]string
}
```

Листинг 2.2 – Код модуля *grammar* — устранение левой рекурсии

```
package grammar

import "slices"

func (g *Grammar) EliminateLeftRecursion() *Grammar {
    orderedNT := make([]string, len(g.NonTerminals))
    copy(orderedNT, g.NonTerminals)

    for i := 0; i < len(orderedNT); i++ {
        currentNT := orderedNT[i]

        for j := 0; j < i; j++ {
            prevNT := orderedNT[j]
            g.replaceProductions(currentNT, prevNT)
        }

        g.eliminateImmediateLR(currentNT)
    }

    return g
}

func (g *Grammar) replaceProductions(ai, aj string) {
    productions, exists := g.Rules[ai]
    if !exists {
        return
    }

    var newProductions [][]string
    for _, prod := range productions {
        if len(prod) > 0 && prod[0] == aj {
            gamma := prod[1:]
            ajProductions := g.Rules[aj]

            for _, sigma := range ajProductions {
                newProd := append(append([]string{}, sigma...), gamma...)
                newProductions = append(newProductions, newProd)
            }
```

```
39          } else {
40              newProductions = append(newProductions, prod)
41          }
42      }
43      g.Rules[ai] = newProductions
44  }
45
46  func (g *Grammar) eliminateImmediateLR(nt string) {
47      productions := g.Rules[nt]
48      var alphas [][]string
49      var betas [][]string
50
51      for _, prod := range productions {
52          if len(prod) > 0 && prod[0] == nt {
53              alphas = append(alphas, prod[1:])
54          } else {
55              betas = append(betas, prod)
56          }
57      }
58
59      if len(alphas) == 0 {
60          return
61      }
62
63      newNT := nt + "'"
64      g.NonTerminals = append(g.NonTerminals, newNT)
65
66      var newBetas [][]string
67      for _, beta := range betas {
68          newBetas = append(newBetas, append(beta, newNT))
69      }
70      g.Rules[nt] = newBetas
71
72      var newAlphaProductions [][]string
73      for _, alpha := range alphas {
74          newAlphaProductions = append(newAlphaProductions, append(alpha,
                newNT))
75      }
76      newAlphaProductions = append(newAlphaProductions, []string{Empty})
77
78      g.Rules[newNT] = newAlphaProductions
79  }
80
81  func (g *Grammar) RemoveCycles() *Grammar {
82      orderedNT := g.NonTerminals
83
84      for i, Ai := range orderedNT {
85          for j := 0; j < i; j++ {
```

```
            Aj := orderedNT[j]

            var newCombs [][]string
            for _, comb := range g.Rules[Ai] {
                if len(comb) > 0 && comb[0] == Aj {
                    for _, jComb := range g.Rules[Aj] {
                        newComb := append(slices.Clone(jComb), comb[1:]...)
                        newCombs = append(newCombs, newComb)
                    }
                } else {
                    newCombs = append(newCombs, comb)
                }
            }
            g.Rules[Ai] = newCombs
        }
    }

    return g
}
```

Листинг 2.3 – Код модуля *grammar* — устранение бесполезных символов

```
package grammar

import (
    "reflect"
    "sort"
)

func (g *Grammar) EliminationUselessSymbols() *Grammar {
    useful := make(map[string]bool)
    newUseful := make(map[string]bool)
    terms := make(map[string]bool)

    newUseful[g.Start] = true
    for _, t := range g.Terminals {
        terms[t] = true
    }

    for !reflect.DeepEqual(useful, newUseful) {
        useful, newUseful = newUseful, useful
        for k := range newUseful {
            delete(newUseful, k)
        }


        for symbol, productions := range g.Rules {
            if useful[symbol] {
                for _, p := range productions {
```

```
                    for _, elem := range p {
                        if terms[elem] {
                            newUseful[elem] = true
                        }
                    }
                }
            }
        }

        for k := range useful {
            newUseful[k] = true
        }
    }

    for symbol, productions := range g.Rules {
        if !useful[symbol] {
            continue
        }
        for _, p := range productions {
            for _, elem := range p {
                if terms[elem] {
                    continue
                }
                useful[elem] = true

            }
        }

    }

    newRules := make(map[string][][]string)
    for symbol, productions := range g.Rules {
        if useful[symbol] {
            newRules[symbol] = productions
        }
    }

    newNonTerminals, newTerminals := g.extractSymbolsFromRules(newRules)

    return &Grammar{
        NonTerminals: newNonTerminals,
        Terminals:    newTerminals,
        Start:        g.Start,
        Rules:        newRules,
    }
}

func (g *Grammar) extractSymbolsFromRules(newRules map[string][][]string)
```

```
    (nonTerminals, terminals []string) {
76      existsT := map[string]struct{}{}
77      existsNT := map[string]struct{}{}
78
79      for _, nt := range g.NonTerminals {
80          existsNT[nt] = struct{}{}
81      }
82
83      for _, t := range g.Terminals {
84          existsT[t] = struct{}{}
85      }
86
87      newNT := make(map[string]struct{})
88      newT := make(map[string]struct{})
89
90      for nt := range newRules {
91          newNT[nt] = struct{}{}
92      }
93
94      for _, productions := range newRules {
95          for _, production := range productions {
96              for _, symbol := range production {
97                  if symbol == Empty {
98                      continue
99                  }
100
101                 if _, ok := existsNT[symbol]; ok {
102                     newNT[symbol] = struct{}{}
103                     continue
104                 }
105
106                 if _, ok := existsT[symbol]; ok {
107                     newT[symbol] = struct{}{}
108                     continue
109                 }
110             }
111         }
112     }
113
114     nonTerminals = mapKeysToSlice(newNT)
115     terminals = mapKeysToSlice(newT)
116
117     return nonTerminals, terminals
118 }
119
120 func mapKeysToSlice(m map[string]struct{}) []string {
121     keys := make([]string, 0, len(m))
122     for k := range m {
```

```
123             keys = append(keys, k)
124         }
125         sort.Strings(keys)
126         return keys
127 }
```

Листинг 2.4 – Код модуля *commands*

```
1  package commands
2
3  import (
4      "bytes"
5      "fmt"
6      "github.com/AskaryanKarine/BMSTU-CC/lab_02/internal/fs"
7      "github.com/AskaryanKarine/BMSTU-CC/lab_02/internal/grammar"
8  )
9
10 const (
11     leftRecSuf      = "_lr"
12     leftIndirectSuf = "_ilr"
13     UnlessSymSuf    = "_us"
14 )
15
16 type Command struct {
17     grammarFileName string
18     grammar         *grammar.Grammar
19     outputBuffer    bytes.Buffer
20 }
21
22 func (c *Command) output() string {
23     return c.outputBuffer.String()
24 }
25
26 func (c *Command) LoadGrammar(input string) (string, error) {
27     c.outputBuffer.Reset()
28     gram, err := grammar.ReadGrammarFromFile(input)
29     if err != nil {
30         msg := fmt.Sprintf("ошибка при чтении файла %s: %v\n", input, err)
31         c.outputBuffer.WriteString(msg)
32         return c.output(), err
33     }
34     msg := fmt.Sprintf("Грамматика %s прочитана успешно", input)
35     c.outputBuffer.WriteString(msg)
36     c.grammar = gram
37     c.grammarFileName = input
38     return c.output(), nil
39 }
40
41 func (c *Command) EliminatingLeftRecursion() (string, error) {
```

```
42      c.outputBuffer.Reset()
43      c.outputBuffer.WriteString("Вызов устранения левой рекурсии\n")
44      out := c.grammar.Copy().EliminateLeftRecursion().String()
45
46      filename := fs.AddSuffixToFilename(c.grammarFileName, leftRecSuf)
47      err := fs.WriteStringToFile(out, filename)
48      if err != nil {
49          msg := fmt.Sprintf("Ошибка при записи файла %s: %v\n", filename, err)
50          c.outputBuffer.WriteString(msg)
51
52          return c.output(), err
53      }
54
55      msg := fmt.Sprintf("Грамматика записана в файл %s\n", filename)
56      c.outputBuffer.WriteString(msg)
57
58      return c.output(), nil
59  }
60
61  func (c *Command) EliminatingLeftIndirectRecursion() (string, error) {
62      c.outputBuffer.Reset()
63      c.outputBuffer.WriteString("Вызов устранения косвенной левой рекурсии\n")
64
65      out :=
            c.grammar.Copy().RemoveCycles().EliminateLeftRecursion().String()
66      filename := fs.AddSuffixToFilename(c.grammarFileName, leftIndirectSuf)
67      err := fs.WriteStringToFile(out, filename)
68      if err != nil {
69          msg := fmt.Sprintf("Ошибка при записи файла %s: %v\n", filename, err)
70          c.outputBuffer.WriteString(msg)
71
72          return c.output(), err
73      }
74
75      msg := fmt.Sprintf("Грамматика записана в файл %s\n", filename)
76      c.outputBuffer.WriteString(msg)
77
78      return c.output(), nil
79  }
80
81  func (c *Command) EliminationUselessSymbols() (string, error) {
82      c.outputBuffer.Reset()
83      c.outputBuffer.WriteString("Вызов устранения бесполезных символов\n")
84      out := c.grammar.Copy().EliminationUselessSymbols().String()
85
86      filename := fs.AddSuffixToFilename(c.grammarFileName, UnlessSymSuf)
87      err := fs.WriteStringToFile(out, filename)
88      if err != nil {
```

```
89          msg := fmt.Sprintf("Ошибка при записи файла %s: %v\n", filename, err)
90          c.outputBuffer.WriteString(msg)
91
92          return c.output(), err
93      }
94
95      msg := fmt.Sprintf("Грамматика записана в файл %s\n", filename)
96      c.outputBuffer.WriteString(msg)
97
98      return c.output(), nil
99  }
```

Листинг 2.5 – Код модуля *fs*

```
1  package fs
2
3  import (
4      "os"
5      "path/filepath"
6  )
7
8  func AddSuffixToFilename(filename, suffix string) string {
9      ext := filepath.Ext(filename)
10     name := filename[:len(filename)-len(ext)]
11     return name + suffix + ext
12 }
13
14 func WriteStringToFile(data, filename string) error {
15     return os.WriteFile(filename, []byte(data), 0644)
16 }
```

3 Контрольные вопросы

3.1 Как может быть определён формальный язык?

Формальный язык может быть определён, например:

1. простым перечислением слов, входящих в данный язык. Этот способ, в основном, применим для определения конечных языков и языков простой структуры;
2. словами, порождёнными некоторой формальной грамматикой;
3. словами, порождёнными регулярным выражением;
4. словами, распознаваемыми некоторым конечным автоматом;
5. словами, порождёнными БНФ-конструкцией.

3.2 Какими характеристиками определяется грамматика?

Грамматика определяется следующими характеристиками:

1. Σ — набор (алфавит) терминальных символов;
2. N — набор (алфавит) нетерминальных символов;
3. P — набор правил вида: «левая часть» $\rightarrow$ «правая часть», где:
   - «левая часть» — непустая последовательность терминалов и нетерминалов, содержащая хотя бы один нетерминал;
   - «правая часть» — любая последовательность терминалов и нетерминалов;
4. S — стартовый (или начальный) символ грамматики из набора нетерминалов.

3.3 Дайте описания грамматик по иерархии Хомского.

Грамматика с фразовой структурой G — это алгебраическая структура, упорядоченная четвёрка (V_T, V_N, P, S), где:

- V_T — алфавит (множество) терминальных символов;
- V_N — алфавит (множество) нетерминальных символов;
- $V = V_T \cup V_N$ — словарь G, причём $V_T \cap V_N = \varnothing$;
- P — конечное множество продукций (правил) грамматики, $P \subseteq V^+ \times V^*$;
- S — начальный символ (источник).

Здесь V^* — множество всех строк над алфавитом V, а V^+ — множество непустых строк над алфавитом V.

По иерархии Хомского, грамматики делятся на 4 типа, каждый последующий является более ограниченным подмножеством предыдущего (но и легче поддающимся анализу).

1. неограниченные грамматики — возможны любые правила;
2. контекстно-зависимые грамматики — левая часть может содержать один нетерминал, окруженный «контекстом» (последовательности символов, в том же виде присутствующие в правой части); сам нетерминал заменяется непустой последовательностью символов в правой части;
3. контекстно-свободные грамматики — левая часть состоит из одного нетерминала;
4. регулярные грамматики — более простые, эквивалентны конечным автоматам.

3.3.1 Неограниченные грамматики

Это все без исключения формальные грамматики. Правила можно записать в виде: $\alpha \to \beta$, где $\alpha \in V^+$ — любая непустая цепочка, содержащая хотя бы один нетерминальный символ, а $\beta \in V^*$ — любая цепочка символов из алфавита.

3.3.2 Контекстно-зависимые грамматики

К этому типу относятся контекстно-зависимые (КЗ) грамматики и неукорачивающие грамматики. Для грамматики $G(V_T, V_N, P, S)$, $V = V_T \cup V_N$ все правила имеют вид:

— $\alpha A \beta \to \alpha \gamma \beta$, где $\alpha, \beta \in V^*$, $\gamma \in V^+$, $A \in V_N$. Такие грамматики относят к контекстно-зависимым.

— $\alpha \to \beta$, где $\alpha, \beta \in V^+$, $1 \leq |\alpha| \leq |\beta|$. Такие грамматики относят к неукорачивающим.

3.3.3 Контекстно-свободные грамматики

Для грамматики $G(V_T, V_N, P, S)$, $V = V_T \cup V_N$ все правила имеют вид: $A \to \beta$, где $\beta \in V^+$ (для неукорачивающих КС-грамматик) или $\beta \in V^*$ (для укорачивающих), $A \in V_N$. То есть грамматика допускает появление в левой части правила только нетерминального символа.

3.3.4 Регулярные грамматики

К третьему типу относятся регулярные грамматики (автоматные) — самые простые из формальных грамматик. Они являются контекстно-свободными, но с ограниченными возможностями.

Все регулярные грамматики могут быть разделены на два эквивалентных класса, которые для грамматики вида III будут иметь правила следующего вида:

— $A \to B\gamma$ или $A \to \gamma$, где $\gamma \in V_T^*$, $A, B \in V_N$ (для леволинейных грамматик).

— $A \to \gamma B$ или $A \to \gamma$, где $\gamma \in V_T^*$, $A, B \in V_N$ (для праволинейных грамматик).

3.4 Какие абстрактные устройства используются для разбора грамматик?

1. Для разбора слов из регулярных языков подходят формальные автоматы самого простого устройства, т. н. конечные автоматы. Их функция перехода задаёт только смену состояний и, возможно, сдвиг (чтение) входного символа.
2. Для разбора слова из контекстно-свободных языков в автомат приходится добавлять «магазинную ленту» или «стек», в который при каждом переходе записывается цепочка на основе соответствующего алфавита магазина. Такие автоматы называют «магазинные автоматы».
3. Для контекстно-зависимых языков разработаны ещё более сложные линейно-ограниченные автоматы, а для языков общего вида — машина Тьюринга.

3.5 Оцените временную и емкостную сложность предложенного вам алгоритма.

Временная сложность — $O(|P|^2)$, где P — конечное множество продукций (правил) грамматики.

Ёмкостная сложность — $O(|P|)$, где P — конечное множество продукций (правил) грамматики.