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 № 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 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' в нормальной форме Грейбах.

2 Практическая часть

2.1 Листинг

Листинг 2.1 – Исходный код модели для грамматики

```
const (
    Empty = ""
)

type Rule struct {
    NonTerminal  string
    Combinations [][]string
}

type Grammar struct {
    NonTerminals []string
    Terminals    []string
    Start        string

    Rules []Rule
}
```

Листинг 2.2 – Исходный код алгоритма устранения левой рекурсии

```
func RemoveLeftRecursionV0(g *grammar.Grammar) *grammar.Grammar {
    n := len(g.Rules)
    for i := 0; i < n; i++ {
        for j := 0; j < i; j++ {
            // Замена Ai -> Aj
            newCombinations := make([][]string, 0,
               len(g.Rules[i].Combinations))
            for _, iCombination := range g.Rules[i].Combinations
               {
                if iCombination[0] != g.Rules[j].NonTerminal {
                    newCombinations = append(newCombinations,
                       iCombination)
                } else {
                    for _, jCombination := range
                       g.Rules[j].Combinations {
                        newCombinations =
                           append(newCombinations,
                           append(slices.Clone(jCombination),
```

```
                        iCombination[1:]...))
                }
            }
        }

        g.Rules[i].Combinations = newCombinations
    }

    // Устранение непосредственно левой рекурсии
    recursiveCombinations := make([][]string, 0,
       len(g.Rules[i].Combinations))
    nonRecursiveCombinations := make([][]string, 0,
       len(g.Rules[i].Combinations))
    for _, iCombination := range g.Rules[i].Combinations {
        if iCombination[0] == g.Rules[i].NonTerminal {
            recursiveCombinations =
               append(recursiveCombinations, iCombination)
        } else {
            nonRecursiveCombinations =
               append(nonRecursiveCombinations, iCombination)
        }
    }

    if len(recursiveCombinations) == 0 {
        continue
    }

    newNonTerminal := g.Rules[i].NonTerminal + "'"

    g.Rules[i].Combinations = make([][]string, 0,
       len(nonRecursiveCombinations))
    for _, nonRecursiveCombination := range
       nonRecursiveCombinations {
        g.Rules[i].Combinations =
           append(g.Rules[i].Combinations,
           append(nonRecursiveCombination, newNonTerminal))
    }

    newRule := grammar.Rule{
        NonTerminal:  newNonTerminal,
        Combinations: make([][]string, 0,
```

```
                len(recursiveCombinations)),
        }
        for _, recursiveCombination := range
            recursiveCombinations {
            newRule.Combinations = append(newRule.Combinations,
                append(recursiveCombination[1:], newNonTerminal))
        }
        newRule.Combinations = append(newRule.Combinations,
            []string{grammar.Empty})

        g.Rules = append(g.Rules, newRule)
        g.NonTerminals = append(g.NonTerminals, newNonTerminal)
    }

    return g
}

func RemoveLeftRecursionV1(g *grammar.Grammar) *grammar.Grammar {
    n := len(g.Rules)
    for i := 0; i < n; i++ {
        for j := 0; j < i; j++ {
            // Замена Ai -> Aj
            newCombinations := make([][]string, 0,
                len(g.Rules[i].Combinations))
            for _, iCombination := range g.Rules[i].Combinations
                {
                if iCombination[0] != g.Rules[j].NonTerminal {
                    newCombinations = append(newCombinations,
                        iCombination)
                } else {
                    for _, jCombination := range
                        g.Rules[j].Combinations {
                        newCombinations =
                            append(newCombinations,
                            append(slices.Clone(jCombination),
                            iCombination[1:]...))
                    }
                }
            }

            g.Rules[i].Combinations = newCombinations
```

```
75          }
76
77          // Устранение непосредственно левой рекурсии
78          recursiveCombinations := make([][]string, 0,
               len(g.Rules[i].Combinations))
79          nonRecursiveCombinations := make([][]string, 0,
               len(g.Rules[i].Combinations))
80          for _, iCombination := range g.Rules[i].Combinations {
81              if iCombination[0] == g.Rules[i].NonTerminal {
82                  recursiveCombinations =
                       append(recursiveCombinations, iCombination)
83              } else {
84                  nonRecursiveCombinations =
                       append(nonRecursiveCombinations, iCombination)
85              }
86          }
87
88          if len(recursiveCombinations) == 0 {
89              continue
90          }
91
92          newNonTerminal := g.Rules[i].NonTerminal + "'"
93
94          g.Rules[i].Combinations = make([][]string, 0,
               len(nonRecursiveCombinations))
95          for _, nonRecursiveCombination := range
               nonRecursiveCombinations {
96              g.Rules[i].Combinations =
                       append(g.Rules[i].Combinations,
                       slices.Clone(nonRecursiveCombination))
97              g.Rules[i].Combinations =
                       append(g.Rules[i].Combinations,
                       append(nonRecursiveCombination, newNonTerminal))
98          }
99
100         newRule := grammar.Rule{
101             NonTerminal:  newNonTerminal,
102             Combinations: make([][]string, 0,
                       len(recursiveCombinations)),
103         }
104         for _, recursiveCombination := range
```

```
            recursiveCombinations {
                newRule.Combinations = append(newRule.Combinations,
                    slices.Clone(recursiveCombination[1:]))
                newRule.Combinations = append(newRule.Combinations,
                    append(recursiveCombination[1:], newNonTerminal))
            }

            g.Rules = append(g.Rules, newRule)
            g.NonTerminals = append(g.NonTerminals, newNonTerminal)
        }

        return g
}
```

Листинг 2.3 – Исходный код алгоритма приведения к форме Грейбах

```
func sortRules(g *grammar.Grammar) {
    slices.SortFunc(g.Rules, func(rule1 grammar.Rule, rule2
        grammar.Rule) int {
        for _, c := range rule2.Combinations {
            if rule1.NonTerminal == c[0] {
                return 1
            }
        }

        for _, c := range rule1.Combinations {
            if rule2.NonTerminal == c[0] {
                return -1
            }
        }

        return 0
    })
}

func Greibach(g *grammar.Grammar) *grammar.Grammar {
    sortRules(g)

    for i := len(g.Rules) - 1; i >= 0; i-- {
        for j := i + 1; j < len(g.Rules); j++ {
            newCombinations := make([][]string, 0,
                len(g.Rules[i].Combinations))
            for _, iCombination := range g.Rules[i].Combinations
```

```
                {
                    if iCombination[0] != g.Rules[j].NonTerminal {
                        newCombinations = append(newCombinations,
                            iCombination)
                    } else {
                        for _, jCombination := range
                            g.Rules[j].Combinations {
                            newCombinations =
                                append(newCombinations,
                                append(slices.Clone(jCombination),
                                iCombination[1:]...))
                        }
                    }
                }

                g.Rules[i].Combinations = newCombinations
            }
        }

        rules := make([]grammar.Rule, 0, len(g.Rules))
        for i, r := range g.Rules {
            for j, c := range r.Combinations {
                for k := len(c) - 1; k > 0; k-- {
                    if !slices.Contains(g.Terminals, c[k]) {
                        continue
                    }

                    oldTerminal := c[k]
                    newNonTerminal := oldTerminal + "'"
                    g.Rules[i].Combinations[j][k] = newNonTerminal

                    if !slices.Contains(g.NonTerminals,
                        newNonTerminal) {
                        g.NonTerminals = append(g.NonTerminals,
                            newNonTerminal)
                        rules = append(rules, grammar.Rule{
                            NonTerminal:  newNonTerminal,
                            Combinations: [][]string{{oldTerminal}},
                        })
                    }
                }
```

```
59            }
60        }
61
62        g.Rules = append(g.Rules, rules...)
63
64        return g
65    }
```

2.2 Результаты выполнения программы

Таблица 2.1 – Результаты выполнения программы для устранения левой рекурсии

Входная грамматика	Результат
Устранение левой рекурсии	
2 S A 4 a b c d 5 S ->Aa S ->b A ->Ac A ->Sd A ->ϵ S	3 S A A' 4 a b c d 10 S ->Aa S ->b A ->bd A ->bdA' A ->ϵ A ->ϵA' A' ->c A' ->cA' A' ->ad A' ->adA' S
3 E T F 5 + * ( ) a 6 E ->E+T E ->T T ->T*F T ->F F ->(E) F ->a E	5 E T F E' T' 5 + * ( ) a 10 E ->T E ->TE' T ->F T ->FT' F ->(E) F ->a E' ->+T E' ->+TE' T' ->*F T' ->*FT' E

Таблица 2.2 – Результаты выполнения программы для приведения к форме Грейбах

Приведение к форме Грейбах	
5 E T F E' T' 5 + * ( ) a 10 E ->T E ->TE' T ->F T ->FT' F ->(E) F ->a E' ->+T E' ->+TE' T' ->*F T' ->*FT' E	6 E T F E' T')' 5 + * ( ) a 19 E ->(E)' E ->a E ->(E)'T' E ->aT' E ->(E)'E' E ->aE' E ->(E)'T'E' E ->aT'E' T ->(E)' T ->a T ->(E)'T' T ->aT' F ->(E)' F ->a E' ->+T E' ->+TE' T' ->*F T' ->*FT')' ->) E

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.