ГУАП

КАФЕДРА № 44

ОТЧЕТ
ЗАЩИЩЕН С ОЦЕНКОЙ

ПРЕПОДАВАТЕЛЬ

канд. техн. наук, доцент		Н.В. Кучин
должность, уч. степень, звание	подпись, дата	инициалы, фамилия

ОТЧЕТ О ЛАБОРАТОРНОЙ РАБОТЕ №2

ПОСТРОЕНИЕ СИНТАКСИЧЕСКОГО АНАЛИЗАТОРА И ПРОСТЕЙШЕГО ДЕРЕВА ВЫВОДА

по курсу: Системное программное обеспечение

РАБОТУ ВЫПОЛНИЛ

СТУДЕНТ гр. № 4142		Д.Р. Рябов
	подпись, дата	инициалы, фамилия

Санкт-Петербург 2024

Вариант №19

1. Цель работы: Изучение основных понятий теории грамматик простого и операторного предшествования, ознакомление с алгоритмами синтаксического анализа (разбора) для некоторых классов КС-грамматик, получение практических навыков создания простейшего синтаксического анализатора для заданной грамматики операторного предшествования. Получение практических навыков создания простейшего синтаксического анализатора для заданной грамматики операторного предшествования, обработка и представление результатов синтаксического анализа..

2. Задание:

Вариант 19.

Грамматика 1.

В список допустимых лексем входят: Идентификаторы, символьные константы (в одинарных кавычках)

Требуется написать программу, которая выполняет лексический анализ входного текста в соответствии с заданием, порождает таблицу лексем и выполняет синтаксический разбор текста по заданной грамматике. Текст на входном языке задается в виде символьного (текстового) файла. Допускается исходить из условия, что текст содержит не более одного предложения входного языка.

3. Запись заданной грамматики входного языка в форме Бэкуса-Наура

Язык G({S, F, T, E}, ***{:=,-,+,*,/,(,), 1, a, 'a'}, P****, S):*

S ->**a :=** F**;**

F-> F+T | T
T-> T*E | T/E | E
E-> **(**F**)** | **-(**F**)** | **a**

4. Заполненная матрица предшествования для грамматики

Символы	:=	-	+	*	/	(	)	a	‘a’	1	⊥n	⊥*k*
:=												<-
-	->		<-	<-	<-	<-	->	<-	<-	<-	>.	
+	->		<-	<-	<-	<-	->	<-	<-	<-		
*	->		->	<-	<-	<-	->	<-	<-	<-		
/	->		->	<-	<-	<-	->	<-	<-	<-	>.	
(	<-	<-	<-	<-	<-	<-	=.	<-	<-	<-	>.	
)	->		->	->	->		->				->	
a	->		->	->	->		->				->	
‘a’	->		->	->	->		->				->	
1	->		->	->	->		->				->	>.
⊥n	<-	<-	<-	<-	<-	<-		<-	<-	<-		

5. Исходная грамматика с одним не терминалом

S’ -> a := F

f -> F + F | F * F | F/F | (F) | -(F) | a

6. Пример разбора простейшего предложения

Входная цепочка: a := a + a

Входная строка	Стек	Действие
a := (a + a) * a⊥к	⊥*n*	÷п
:= (a + a) * a⊥к	⊥*n* a	÷п
(a + a) * a⊥к	⊥н a :=	÷п
a + a) * a⊥к	⊥н a := (	÷п
+ a) * a⊥к	⊥н a := (a	÷с
a) * a⊥к	⊥н a := (a +	÷п
) * a⊥к	⊥н a := (a + a	÷п
* a⊥к	⊥н a := (a + a)	÷с (свертка E -> (F))
* a⊥к	⊥н a := E	÷с (свертка T -> E)
* a⊥к	⊥н a := F	÷п
a⊥к	⊥н a := F *	÷п
⊥к	⊥н a := F * a	÷с (свертка T -> T * E)
⊥к	⊥н a := F	÷с (свертка F -> T)
⊥к	⊥н a := F	÷с (свертка S -> a := F)
⊥к	⊥н E	÷с

7. Пример построения дерева вывода

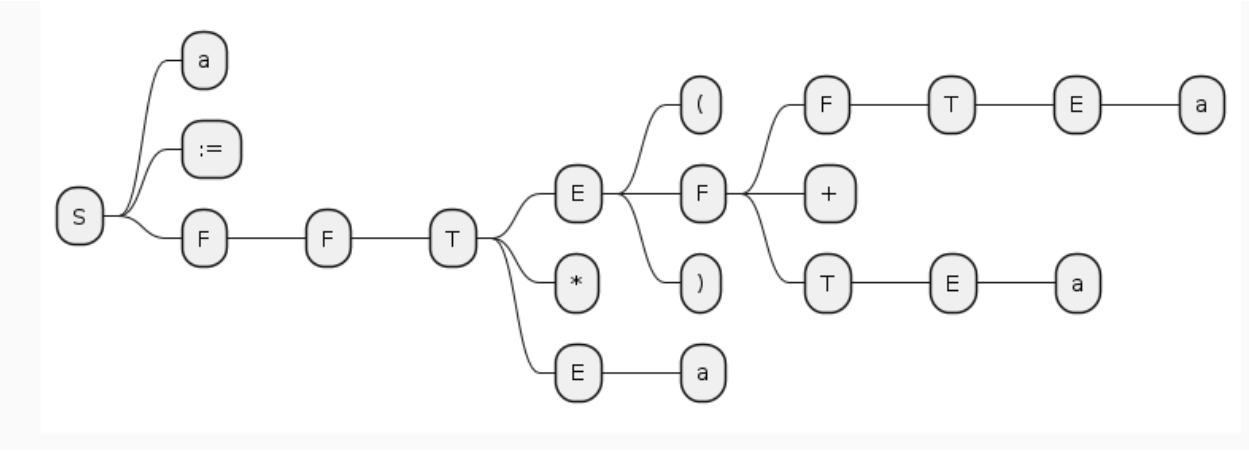

Рисунок 1 – Пример дерева вывода для входной цепочки

10. Текст программы

На листинге 1 представлен код программы на языке Golang. Программа считывает входной код из stdin. После разбивает его на лексемы используя код из предыдущей лабораторной представленный листингом 2. После чего разбивает его на дерево в соответствии с матрицей предшествования.

Листинг 1 – код построителя дерева вывода

```
package main

import (
    "encoding/json"
    "fmt"
    lab1 "lab2/modules" // Импорт пакета лексера из первой лабораторной
    "os"
)

// Rule представляет правило продукции в грамматике
type Rule struct {
    Svertka string   // Символ продукции
    Posl    []string // Последовательность продукции
}

// Node представляет узел в дереве разбора
type Node struct {
    Type     string // Нетерминал или терминал
```

```
    Lexem    string // Значение лексемы
    Children []*Node
}

func main() {
    // Генерация токенов с использованием лексера из первой лабораторной
    lexems := lab1.RunLexer()

    // Проверка на ошибки лексера
    if hasErrors(lexems) {
        fmt.Println("Лексер обнаружил ошибки.")
        return
    }

    // Определение правил грамматики
    rules := []Rule{
        {Svertka: "E", Posl: []string{"identifier"}},
        {Svertka: "E", Posl: []string{"arithmetic", "l_bracket", "F",
"r_bracket"}},
        {Svertka: "E", Posl: []string{"arithmetic", "l_bracket",
"identifier", "r_bracket"}},
        {Svertka: "E", Posl: []string{"l_bracket", "identifier",
"r_bracket"}},
        {Svertka: "E", Posl: []string{"l_bracket", "F", "r_bracket"}},
        {Svertka: "T", Posl: []string{"T", "arithmetic", "E"}},
        {Svertka: "T", Posl: []string{"E"}},
        {Svertka: "F", Posl: []string{"identifier", "arithmetic",
"identifier"}},
        {Svertka: "F", Posl: []string{"F", "arithmetic", "T"}},
        {Svertka: "F", Posl: []string{"T"}},
        {Svertka: "S", Posl: []string{"F", "operator", "identifier",
"delimiter"}},
        {Svertka: "S", Posl: []string{"F", "operator", "F",
"delimiter"}},
    }

    // Выполнение синтаксического анализа
    svertka, tree := parse(lexems, rules)

    if svertka == "S" {
        saveTree(tree)
    } else {
        fmt.Println("Выражение некорректно")
    }
}
```

```go
// hasErrors проверяет, были ли ошибки у лексера
func hasErrors(tokens []lab1.Token) bool {
    for _, token := range tokens {
        if token.Type == lab1.ErrorType {
            return true
        }
    }
    return false
}

// parse выполняет синтаксический анализ на основе заданных токенов и
правил грамматики
func parse(tokens []lab1.Token, rules []Rule) (string, *Node) {
    var stack []*Node

    // Помещение всех токенов в стек
    for _, token := range tokens {
        stack = append(stack, &Node{Type: token.Type, Lexem:
token.Value})
    }

    // Применение правил, пока стек не содержит только один элемент
    for len(stack) > 1 {
        applied := false

        for _, rule := range rules {
            if canApplyRule(stack, rule) {
                stack = applyRule(stack, rule)
                applied = true
                break
            } else {
                fmt.Printf("Невозможно применить правило: %s -> %v к
стеку\n", rule.Svertka, rule.Posl)
            }
        }

        if !applied {
            fmt.Println("Ошибка: невозможно применить ни одно правило.")
            return "ошибка", nil
        }
    }

    return stack[0].Type, stack[0]
}
```

```
// canApplyRule проверяет, может ли правило быть применено к текущему
стеку
func canApplyRule(stack []*Node, rule Rule) bool {
    if len(rule.Posl) > len(stack) {
        return false
    }

    for i, symbol := range rule.Posl {
        stackIndex := len(stack) - len(rule.Posl) + i
        if symbol != stack[stackIndex].Type {
            fmt.Printf("Несоответствие правила: символ правила %s, символ
стека %s в индексе %d стека\n", symbol, stack[stackIndex].Type,
stackIndex)
            return false
        }
    }

    return true
}

// applyRule применяет правило грамматики к текущему стеку
func applyRule(stack []*Node, rule Rule) []*Node {
    newNode := &Node{Type: rule.Svertka, Lexem: "", Children: []*Node{}}
    stackSize := len(stack)

    // Добавление дочерних элементов к новому узлу в правильном порядке
    for i := 0; i < len(rule.Posl); i++ {
        newNode.Children = append(newNode.Children,
stack[stackSize-len(rule.Posl)+i])
    }

    // Удаление соответствующих элементов из стека
    stack = stack[:stackSize-len(rule.Posl)]

    // Добавление нового узла в стек
    stack = append(stack, newNode)

    return stack
}

// saveTree сохраняет дерево разбора в JSON файл
func saveTree(tree *Node) {
    jsonData, err := json.MarshalIndent(tree, "", "  ")
    if err != nil {
```

```
        fmt.Println("Ошибка форматирования в JSON:", err)
        return
    }

    err = os.WriteFile("Tree.json", jsonData, 0644)
    if err != nil {
        fmt.Println("Ошибка записи в файл:", err)
        return
    }

    fmt.Println("Дерево сохранено в файл Tree.json")
}
```

Листинг 2 – код разбивателя на лексемы

```
package lab1

import (
    "bufio"
    "fmt"
    "os"
    "strings"
)

// Константы для разделителей и наборов символов
const (
    Delimiter     = ";"
// Символ разделителя
    Alphabet      =
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"           //
Алфавит
    Alphanumeric  =
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789" //
Алфавит и цифры
    Numbers       = "0123456789"
// Цифры
    OperatorChars = "+-*/"
// Символы операторов
    Parentheses   = "()"
// Символы скобок
)
```

```
// Типы лексем
const (
    DelimiterType   = "delimiter"   // Разделитель
    IdentifierType  = "identifier"  // Идентификатор
    ConstType       = "const"       // Константа
    AssignmentType  = "assignment"  // Присваивание
    OperatorType    = "operator"    // Оператор
    ParenthesesType = "parentheses" // Скобки
    NumberType      = "number"      // Число
    ErrorType       = "error"       // Ошибка
    CommentType     = "comment"     // Комментарий
)

// Структура TokenSaver для хранения и печати токенов
type TokenSaver struct {
    tokens []Token
}

// Структура Token представляет лексему с ее типом и значением
type Token struct {
    Type  string // Тип
    Value string // Значение
}

// Add добавляет токен в TokenSaver
func (ts *TokenSaver) Add(tokenType, value string) {
    ts.tokens = append(ts.tokens, Token{Type: tokenType, Value: value})
}

// Print выводит все токены из TokenSaver
func (ts *TokenSaver) Print() {
    fmt.Println("Токены:")
    for _, token := range ts.tokens {
        fmt.Printf("{ Тип: %s, Значение: %s }\n", token.Type,
token.Value)
    }
}

func main() {
    RunLexer()
}

// RunLexer обрабатывает ввод и генерирует токены
func RunLexer() []Token {
    state := DelimiterType          // Начальное состояние - разделитель
```

```
    token := ""                     // Текущий токен
    tokenSaver := &TokenSaver{}     // Сохранитель токенов
    characters := readCharacters() // Получаем ввод

    for i := 0; i < len(characters); i++ {
        char := string(characters[i]) // Текущий символ
        switch state {
        case DelimiterType:
            handleDelimiter(char, tokenSaver) // Обработка разделителя
            if char != " " && char != "\n" && char != "\r" {
                token = char                // Начинаем новый токен
                state = GetLexemeType(char) // Определяем тип лексемы
            }
        case IdentifierType, NumberType, OperatorType, ParenthesesType,
ErrorType:
            if IsValid(char, state) {
                token += char // Продолжаем собирать текущий токен
            } else {
                tokenSaver.Add(state, token) // Завершаем текущий токен и
сохраняем
                token = ""                   // Начинаем новый токен
                state = DelimiterType        // Возвращаемся к обработке
разделителей
                i--                          // Повторно обрабатываем
текущий символ
            }
        case AssignmentType:
            handleAssignment(char, tokenSaver, &state, &token, &i,
characters) // Обработка оператора присваивания
        case ConstType:
            handleConst(char, tokenSaver, &state, &token, &i, characters)
// Обработка константы
        case CommentType:
            if char == "\n" {
                state = DelimiterType // Обработка комментария до конца
строки
            }
        }
    }
    if token != "" && state != CommentType && token != "#" {
        tokenSaver.Add(state, token) // Добавляем последний токен, если
он есть
    }
    tokenSaver.Print() // Печать всех токенов
    return tokenSaver.tokens
```

```
}

// handleDelimiter обрабатывает разделитель
func handleDelimiter(char string, tokenSaver *TokenSaver) {
    if char == Delimiter {
        tokenSaver.Add(DelimiterType, Delimiter) // Добавляем разделитель
в токены
    }
}

// GetLexemeType возвращает тип лексемы для заданного символа
func GetLexemeType(char string) string {
    if strings.Contains(Alphabet, char) {
        return IdentifierType // Идентификатор
    } else if strings.Contains(Numbers, char) {
        return NumberType // Число
    } else if char == "'" {
        return ConstType // Константа
    } else if char == "#" {
        return CommentType // Комментарий
    } else if strings.Contains(OperatorChars, char) {
        return OperatorType // Оператор
    } else if strings.Contains(Parentheses, char) {
        return ParenthesesType // Скобки
    } else if char == ";" {
        return DelimiterType // Разделитель
    } else if char == ":" {
        return AssignmentType // Оператор присваивания
    }
    return ErrorType // Ошибка
}

// IsValid проверяет, является ли символ допустимым в текущем состоянии
func IsValid(char string, state string) bool {
    switch state {
    case IdentifierType:
        return strings.Contains(Alphanumeric, char) // Допустимы буквы и
цифры
    case NumberType:
        return strings.Contains(Numbers, char) // Допустимы только цифры
    case ConstType:
        return char != "'" // Допустимы любые символы, кроме '
    case OperatorType:
        return strings.Contains(OperatorChars, char) // Допустимы только
символы операторов
```

```
    case ParenthesesType:
        return strings.Contains(Parentheses, char) // Допустимы только
символы скобок
    }
    return false // В остальных случаях недопустимо
}

// handleAssignment обрабатывает оператор присваивания
func handleAssignment(char string, tokenSaver *TokenSaver, state
*string, token *string, i *int, characters []byte) {
    if char == "=" && (*i)+1 < len(characters) && characters[(*i)-1] ==
':' {
        *state = AssignmentType                 // Устанавливаем тип
лексемы - оператор присваивания
        *token = ":="                           // Задаем значение токена
как оператор присваивания
        (*i)++                                  // Пропускаем символ '=',
так как он уже обработан
        tokenSaver.Add(AssignmentType, *token) // Добавляем оператор
присваивания в токены
        *state = DelimiterType                  // Возвращаемся к
обработке разделителей
    } else {
        tokenSaver.Add(OperatorType, *token) // Добавляем текущий токен
как оператор
        *token = ""                             // Начинаем новый токен
        *state = ErrorType                      // Устанавливаем тип лексемы
- ошибка
    }
}

// handleConst обрабатывает константу
func handleConst(char string, tokenSaver *TokenSaver, state *string,
token *string, i *int, characters []byte) {
    if strings.Contains(Alphabet, char) && checkConstBrackets(*i,
characters) {
        *state = ConstType                  // Устанавливаем тип лексемы
- константа
        *token = fmt.Sprint("'", char, "'") // Задаем значение токена как
константу
        (*i)++                              // Пропускаем символ
константы
        (*i)++                              // Пропускаем символ "'", так
как он уже обработан
```

```
            tokenSaver.Add(ConstType, *token)    // Добавляем константу в
токены
            *state = DelimiterType               // Возвращаемся к обработке
разделителей
        } else {
            tokenSaver.Add(ErrorType, *token) // Добавляем текущий токен как
ошибку
            *token = ""                          // Начинаем новый токен
            *state = ErrorType                   // Устанавливаем тип лексемы -
ошибка
        }
}

// checkConstBrackets проверяет соответствие скобок в константе
func checkConstBrackets(i int, characters []byte) bool {
    return (string(characters[i-1]) == "'" && string(characters[i+1]) ==
"'") // Проверяем наличие скобок для константы
}

// readCharacters считывает символы из стандартного ввода
func readCharacters() []byte {
    reader := bufio.NewReader(os.Stdin) // Создаем новый Reader для
стандартного ввода
    var characters []byte
    for {
        input, err := reader.ReadString('\n') // Считываем строку из
ввода
        if err != nil {
            break // Завершаем цикл, если произошла ошибка или достигнут
конец ввода
        }
        characters = append(characters, []byte(input)...) // Добавляем
символы из строки в массив символов
    }
    return characters // Возвращаем массив символов
}
```

11. Примеры выполнения

Входное выражение:

a := (a + a) * a

Результат выполнения программы:

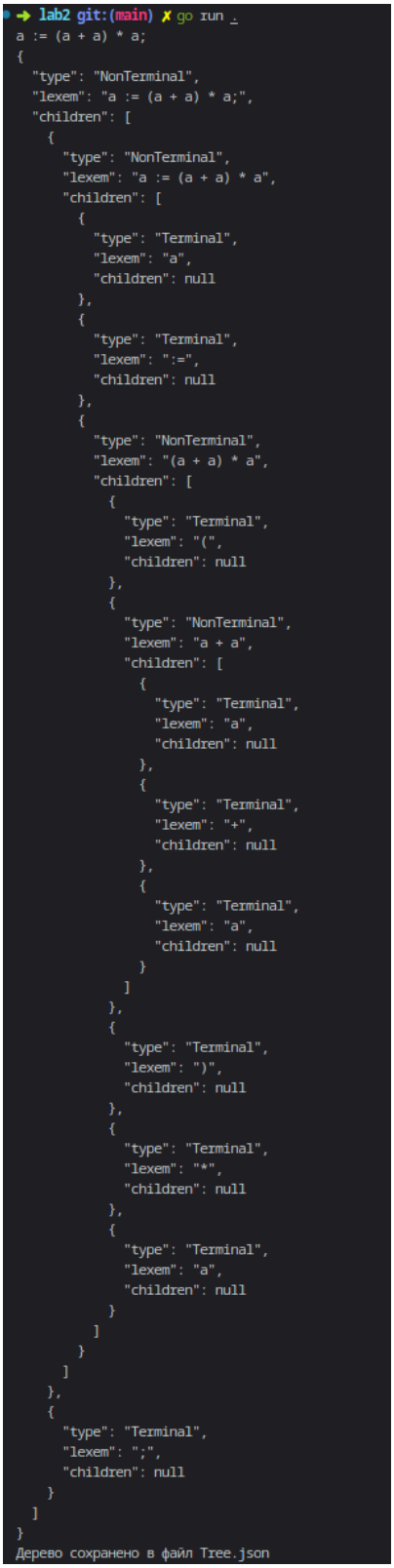

Рисунок 2 - Выполнение программы

Листинг 4 – Файл Tree.json для первого выражения


```
{
  "type": "NonTerminal",
  "lexem": "a := (a + a) * a;",
  "children": [
    {
      "type": "NonTerminal",
      "lexem": "a := (a + a) * a",
      "children": [
        {
          "type": "Terminal",
          "lexem": "a"
        },
        {
          "type": "Terminal",
          "lexem": ":="
        },
        {
          "type": "NonTerminal",
          "lexem": "(a + a) * a",
          "children": [
            {
              "type": "Terminal",
              "lexem": "("
            },
            {
              "type": "NonTerminal",
              "lexem": "a + a",
              "children": [
                {
                  "type": "Terminal",
                  "lexem": "a"
                },
                {
                  "type": "Terminal",
                  "lexem": "+"
                },
                {
                  "type": "Terminal",
                  "lexem": "a"
                }
              ]
            },
            {
              "type": "Terminal",
              "lexem": ")"
            },
            {
              "type": "Terminal",
              "lexem": "*"
            },
            {
              "type": "Terminal",
              "lexem": "a"
            }
          ]
        }
      ]
```

```
    },
    {
      "type": "Terminal",
      "lexem": ";"
    }
  ]
}
```


12. Заключение

Были изучены основные понятия теории грамматик простого и операторного предшествования, было проведено ознакомление с алгоритмами синтаксического анализа (разбора) для некоторых классов КС-грамматик, были получены практические навыки создания простейшего синтаксического анализатора для заданной грамматики операторного предшествования. Были получены практические навыки создания простейшего синтаксического анализатора для заданной грамматики операторного предшествования, были обработаны и представлены результаты синтаксического анализа.