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 №1

ПОСТРОЕНИЕ РАСПОЗНАВАТЕЛЯ ДЛЯ РЕГУЛЯРНОЙ ГРАММАТИКИ И ЛЕКСИЧЕСКОГО АНАЛИЗАТОРА

по курсу: Системное программное обеспечение

РАБОТУ ВЫПОЛНИЛ

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

Санкт-Петербург 2023

Вариант №19

1. Цель работы: Изучение основных понятий теории регулярных языков и грамматик, ознакомление с назначением и принципами работы конечных автоматов (КА) и лексических анализаторов (сканеров). Получение практических навыков построения КА на основе заданной регулярной грамматики. Получение практических навыков построения сканера на примере заданного простейшего входного языка.

2. Задание:

Построить регулярную грамматику в соответствии с вариантом задания.

Вариант 19. Входной язык содержит арифметические выражения, разделенные символом **;** (точка с запятой). Арифметические выражения состоят из идентификаторов, символьных констант (один символ в одинарных кавычках), знака присваивания (**:=**), знаков операций +, -, *, / и круглых скобок.

Написать программу, которая выполняет лексический анализ входного текста в соответствии с заданием и порождает таблицу лексем с указанием их типов и значений. Текст на входном языке задаётся в виде символьного (текстового) файла. Программа должна выдавать сообщения о наличие во входном тексте ошибок, которые могут быть обнаружены на этапе лексического анализа. Наличие синтаксических ошибок проверять не требуется.

Длину идентификаторов и строковых констант можно считать ограниченной 32 символами. Программа должна допускать наличие комментариев неограниченной длины во входном файле. Форму организации комментариев предлагается выбрать самостоятельно.

Любые лексемы, не предусмотренные вариантом задания, встречающиеся в исходном тексте, должны трактоваться как ошибочные.

3. Описание регулярной грамматики (с помощью регулярных выражений).

Далее представлены описания лексем в виде регулярных выражений.

Разделитель: ;
Идентификатор: [a-zA-Z][a-zA-Z0-9]*
Символьная константа: '[^']'
Присваивание: :=
Операции: [+\-*/]
Скобки: [()]
Комментарий: #

4. Граф переходов КА для распознавания лексем.

На рисунке 1 приведён граф переходов для распознавания лексем.

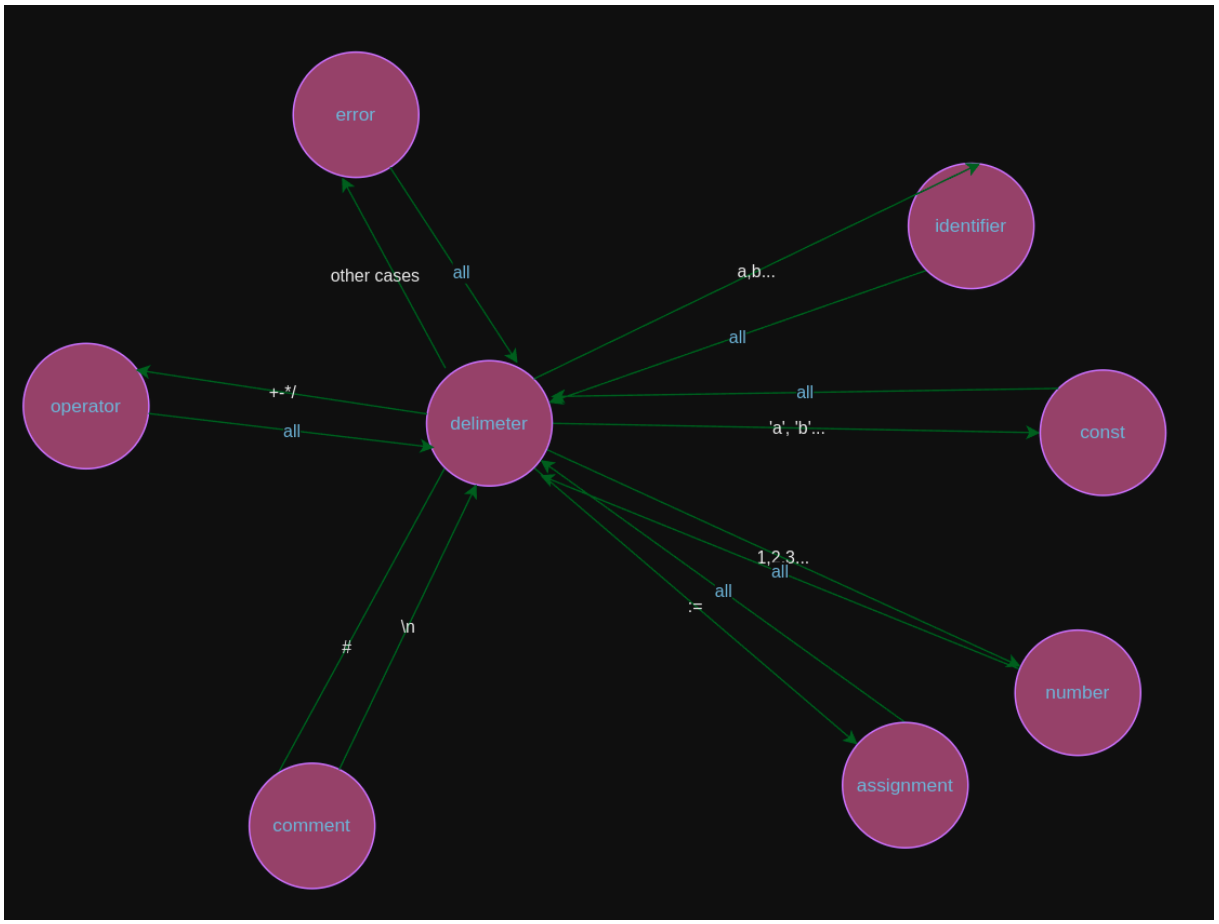

Рисунок 1 — Граф переходов для распознавания лексем.

5. Пример анализируемого входного текста и результат работы лексического анализатора

На рисунке 2 представлен результат работы программы:

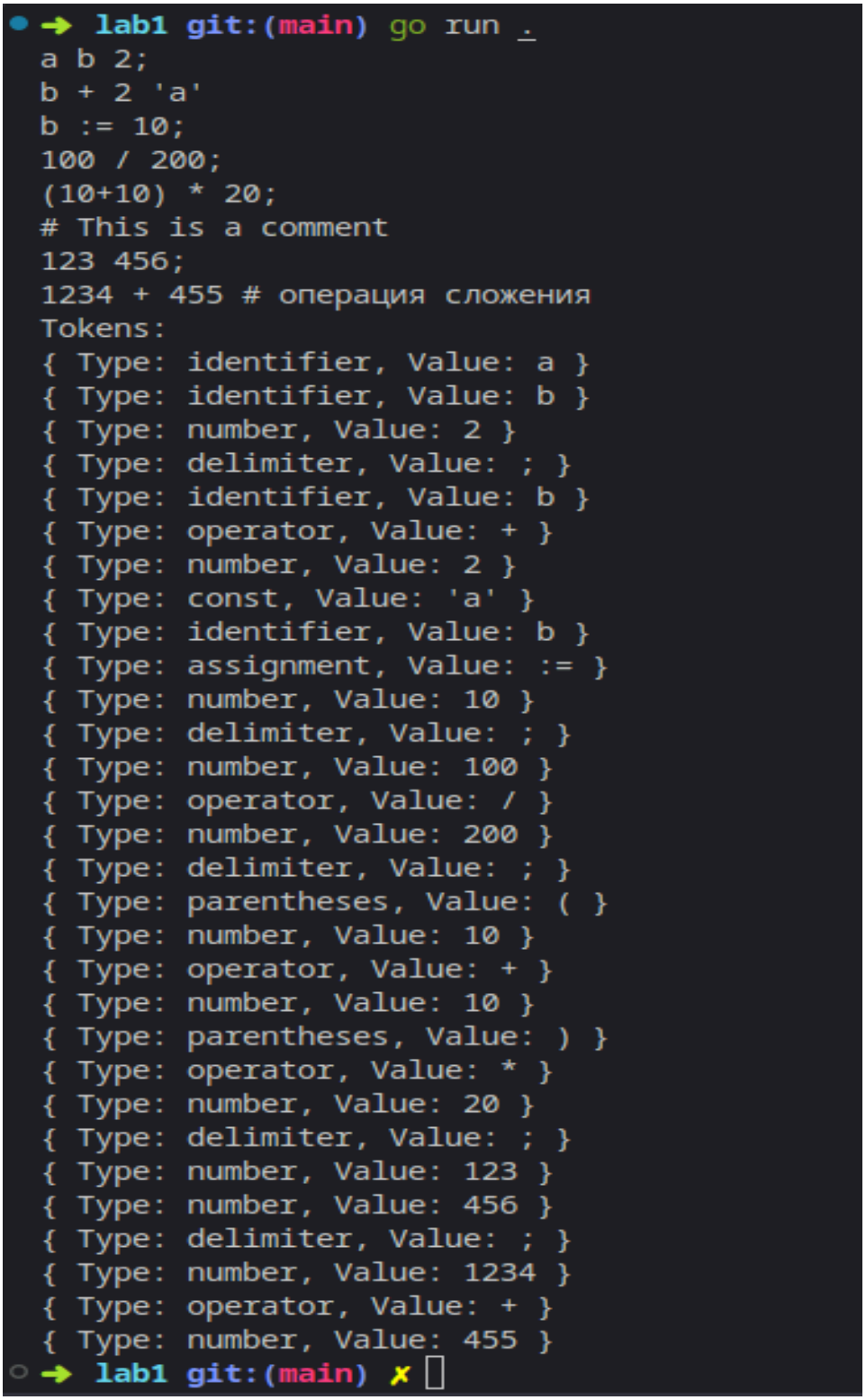

Рисунок 2 – Результат работы программы

На рисунке 3 представлен результат работы программы с ошибками

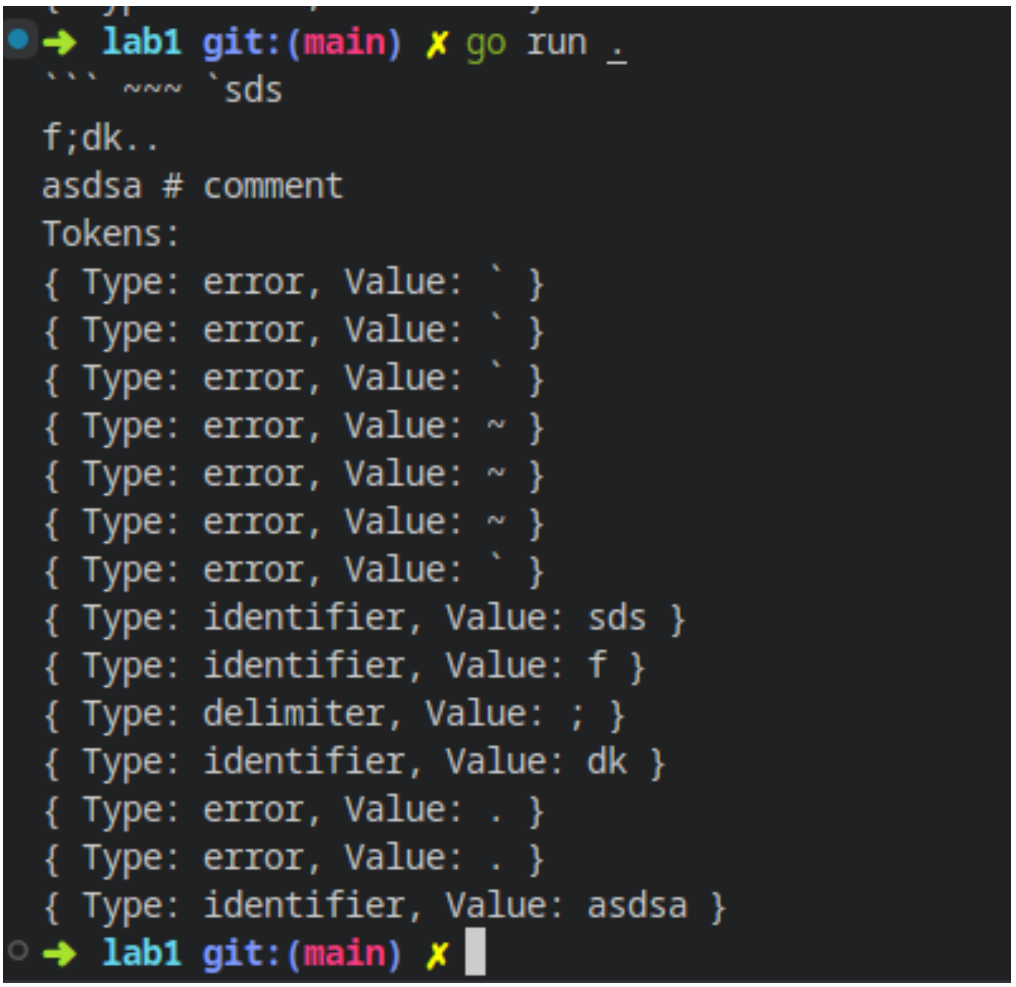

Рисунок 3 — Результат обработки файла с ошибками

6. Программа

Ниже приведен текст программы на языке golang (листинг 1). Программа считывает текст из stdin, разбивает на знаки, после этого она эмулирует работу конечного автомата для последующего распознавания лексем.

```
package main

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

```go
// 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() {
    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
}

// 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
}
```

Листинг 1 — Текст программы

7. Вывод

Были изучены основные понятия теории регулярных языков и грамматик, было проведено ознакомление с назначением и принципами работы конечных автоматов (КА) и лексических анализаторов (сканеров). Были получены практические навыки построения КА на основе заданной регулярной грамматики. Были получены практические навыки построения сканера на примере заданного простейшего входного языка.