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		К.С. Некрасов
	подпись, дата	инициалы, фамилия

Санкт-Петербург 2023

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.

Задание

Для выполнения лабораторной работы требуется 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 задается 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.

Индивидуальное задание

Вариант: 14

Входной язык содержит логические выражения, разделенные символом ; (точка с запятой). Логические выражения состоят из идентификаторов, шестнадцатеричных чисел, знака присваивания (:=), знаков операций or, xor, and, not и круглых скобок.

Ход работы:

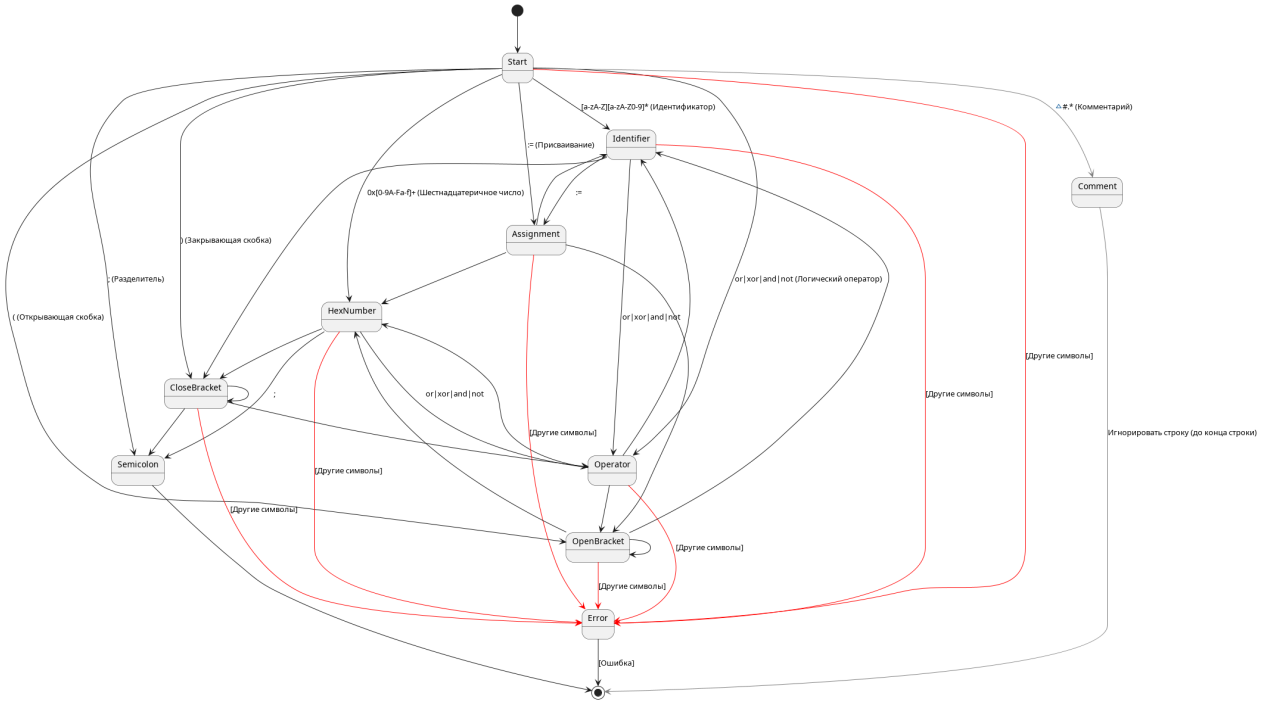
Описание регулярной грамматики

Разделитель: “;”
Идентификатор: “[a-zA-Z][a-zA-Z0-9]”
Оператор: “or|xor|and|not”

Присваивание: “:=”
Шестнадцатиричные числа: “0x[0-9A-Fa-f]+”
Комментарии: “#.*”
Скобки: “[()]”

Граф переходов для распознавания лексем

Граф определения лексем:



Разработка программы

Для разработки программы был выбран язык Golang.

Алгоритм разработанной программы таков: сперва мы читаем входной файл, далее разбиваем его на строки и для каждой строки в цикле обрезаем пробельные символы по краям, проверяем строку на непустоту и пытаемся найти лексему, в случае, если в строке найдена лексема, подходящая под один из заранее определённых шаблонов, она добавляется в таблицу лексем, если же не по одному из шаблонов не было найдено лексем с начала строки, в таблицу добавляется лексема ошибки. Далее обработанная лексема вырезается из обрабатываемой строки и мы возвращаемся к обрезанию пробельных символов. Процесс повторяется пока строка не окажется пустой. Этот алгоритм повторяется для каждой строки файла, после чего полученная таблица выводится в стандартный выход программы.

Текст программы

```
package main

import (
```

```
        "fmt"
        "goodhumored/lr1_analyzer/token"
        "os"
        "strings"
)

func main() {
        tt := &TokenTable{}                // создаём таблицу лексем
        source := getInput("./input.txt") // читаем файл

        // выводим содержимое
        fmt.Println("Содержимое входного файла:\n")
        fmt.Println(source)

        // запускаем распознание лексем
        recogniseTokens(source, tt)

        // выводим лексемы
        fmt.Println("Таблица лексем:")
        fmt.Print(tt)
}

// Читает файл с входными данными, вызывает панику в случае неудачи
func getInput(path string) string {
        data, err := os.ReadFile(path)
        if err ≠ nil {
                panic(err)
        }
        return string(data)
}

// Распознаёт токены в данной строке построчно и записывает в таблицу
func recogniseTokens(source string, tokenTable *TokenTable) {
        for _, line := range strings.Split(source, "\n") {
                recogniseTokensLine(line, tokenTable)
        }
}

// Распознаёт лексемы в данной строке и записывает в таблицу
func recogniseTokensLine(line string, tokenTable *TokenTable) {
```

```
        for {
                line = strings.Trim(line, " ") // обрезаем пробельные символы в стро
                if len(line) == 0 {            // если строка пустая - завершаем обр
                        return
                }
                nextToken := getNextToken(line)      // ищем очередную лексему
                tokenTable.Add(nextToken)            // добавляем лексему в таблицу
                line = line[nextToken.Position.End:] // вырезаем обработанную часть
        }
}

// Ищет очередную лексему в строке
func getNextToken(str string) token.Token {
        // проходим по всем шаблонам лексем
        for _, tokenPattern := range tokenPatterns {
                res := tokenPattern.Pattern.FindStringIndex(str)
                if res != nil {
                        return tokenPattern.Type(str[res[0]:res[1]], token.Position{
                }
        }
        return token.Error(str[0:1], token.Position{0, 1})
}
```

```
package token

// Типы лексем
const (
        DelimiterType   = "delimiter"   // Разделитель
        IdentifierType  = "identifier"  // Идентификатор
        HexType         = "hex_number"  // Шестнадцатиричное число
        AssignmentType  = "assignment"  // Присваивание
        OperatorType    = "operator"    // Логический оператор
        ParenthesesType = "parentheses" // Скобки
        ErrorType       = "error"       // Ошибка
        CommentType     = "comment"     // Комментарий
)

// Структура Position представляет положение лексемы в строке
type Position struct {
        Start int
        End   int
```

```
}

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

// Фабричная функция для токенов, возвращающая замкнутую лямбда функцию для создания
func tokenFactory(tokenType string) func(string, Position) Token {
        return func(value string, position Position) Token {
                return Token{
                        Value:    value,
                        Type:     tokenType,
                        Position: position,
                }
        }
}

// Функции создания лексем определённых типов
var Delimiter = tokenFactory(DelimiterType)
var Identifier = tokenFactory(IdentifierType)
var Assignment = tokenFactory(AssignmentType)
var Operator = tokenFactory(OperatorType)
var Parentheses = tokenFactory(ParenthesesType)
var Hex = tokenFactory(HexType)
var Error = tokenFactory(ErrorType)
var Comment = tokenFactory(CommentType)

package main

import "regexp"
import "goodhumored/lr1_analyzer/token"

// Вспомогатеьлная структура для установки соответствия шаблонов лексем
// с их фабричными функциями
type TokenPattern struct {
        Pattern *regexp.Regexp
        Type    func(string, token.Position) token.Token
}
```

```
// Массив соответствий шаблонов лексем
var tokenPatterns = []TokenPattern{
        {regex("(or|xor|and|not)"), token.Operator},
        {regex("(0x|[0-9$])[0-9a-fA-F]+"), token.Hex},
        {regex("[a-zA-Z][a-zA-Z0-9]+"), token.Identifier},
        {regex(":="), token.Assignment},
        {regex("#.*"), token.Comment},
        {regex("[()]"), token.Parentheses},
        {regex(";"), token.Delimiter},
}

// вспомогательная функция создающая объект регулярного выражения
// добавляющая в начале шаблона признак начала строки
func regex(pattern string) *regexp.Regexp {
        return regexp.MustCompile("^" + pattern)
}

package main

import (
        "fmt"
        "goodhumored/lr1_analyzer/token"

        "strings"
)

// Таблица лексем
type TokenTable struct {
        tokens []token.Token
}

// Метод добавления лексемы в таблицу
func (ts *TokenTable) Add(token token.Token) {
        ts.tokens = append(ts.tokens, token)
}

// Вспомогательная функция для генерации строки с таблицей лексем
func (ts *TokenTable) String() string {
        if len(ts.tokens) == 0 {
                return "Ни одного токена не найдено"
```

```
        }

        // Определяем максимальную ширину столбца
        maxTypeLen := len("Тип")
        maxValueLen := len("Значение")
        for _, token := range ts.tokens {
                if len(token.Type) > maxTypeLen {
                        maxTypeLen = len(token.Type)
                }
                if len(token.Value) > maxValueLen {
                        maxValueLen = len(token.Value)
                }
        }

        // создаем шапку и рамки
        header := fmt.Sprintf("| %-*s | %-*s |", maxTypeLen, "Тип", maxValueLen, "Зн
        border := fmt.Sprintf("+-%s-+-%s-+", strings.Repeat("-", maxTypeLen), string

        // Собираем таблицу
        res := border + "\n" + header + "\n" + border + "\n"
        for _, token := range ts.tokens {
                res += fmt.Sprintf("| %-*s | %-*s |\n", maxTypeLen, token.Type, maxV
        }
        res += border

        return res
}

// Вспомогательная функция для печати таблицы
func (ts *TokenTable) Print() {
        fmt.Println(ts.String())
}
```

Демонстрация работы программы:

Рисунок 1 – результат работы программы

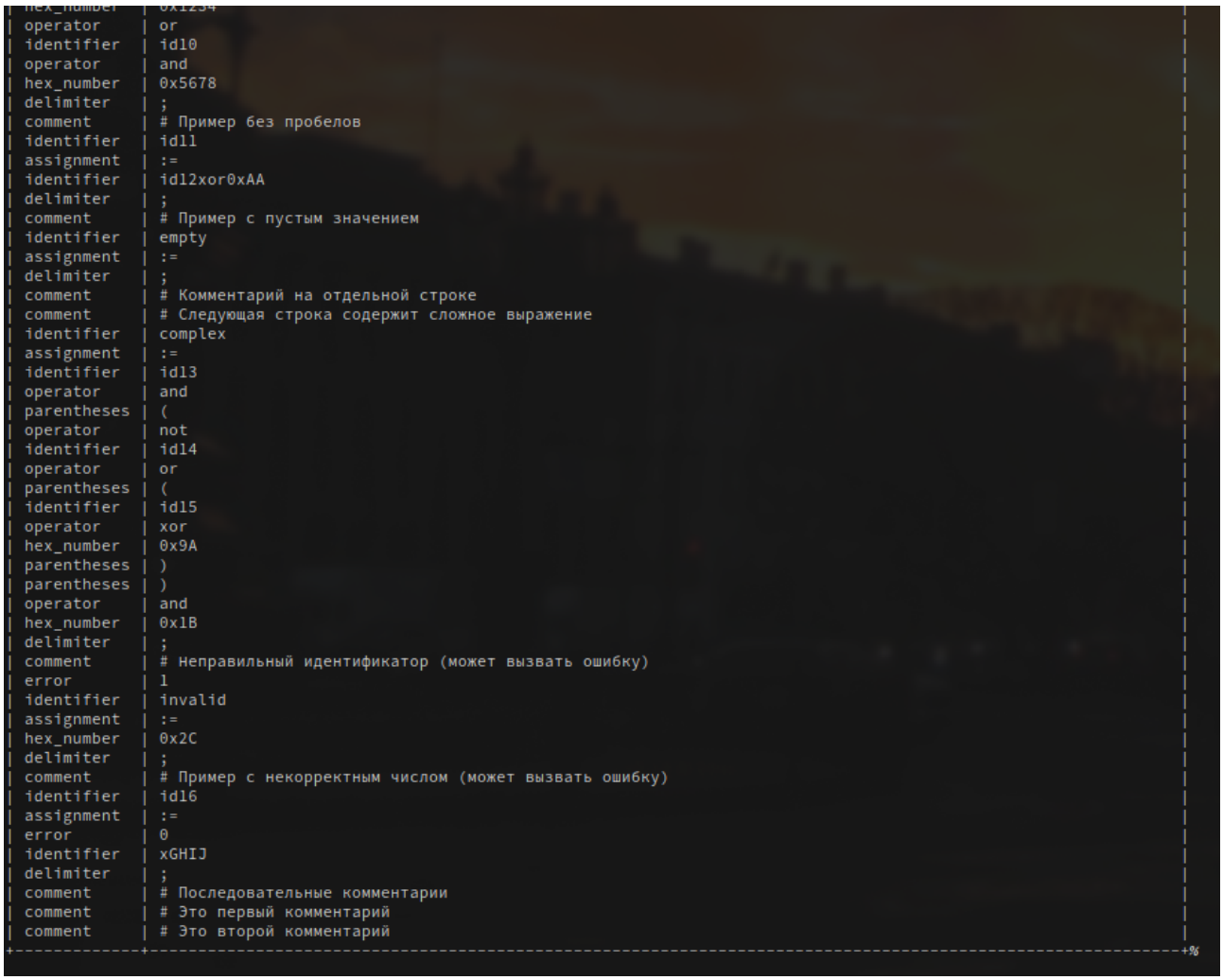


Рисунок 2 – результат работы программы

Вывод:

Изучены основные понятия теории регулярных языков и грамматик, ознакомился с назначением и принципами работы конечных автоматов (КА) и лексических анализаторов (сканеров). Получены практические навыки построения КА на основе заданной регулярной грамматики. Получены практические навыки построения сканера на примере заданного простейшего входного языка.