

ГУАП

КАФЕДРА № 44

ОТЧЕТ
ЗАЩИЩЕН С ОЦЕНКОЙ
ПРЕПОДАВАТЕЛЬ

канд. техн. наук, доцент

должность, уч. степень, звание

подпись, дата

Н.В. Кучин

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

подпись, дата

К.С. Некрасов

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 наличии 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: “:=”

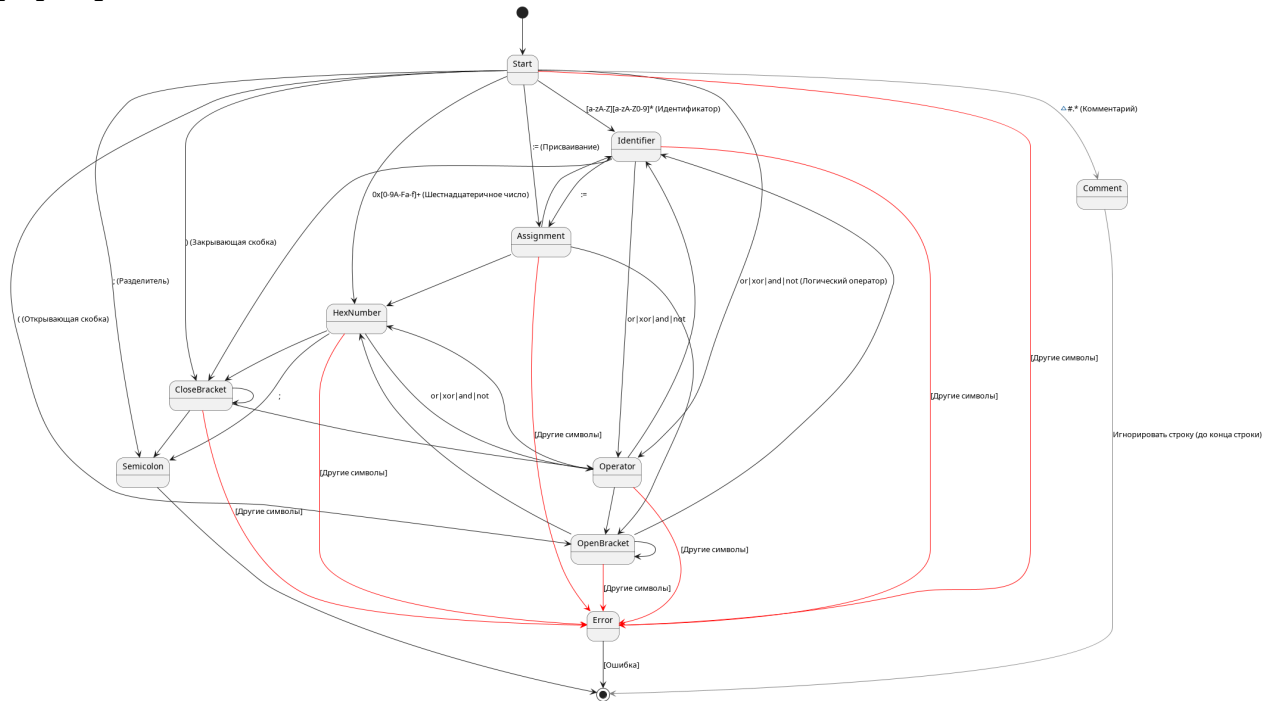
Шестнадцатирічні числа: “0x[0-9A-Fa-f]+”

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 распознавание 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 строке
        if len(line) == 0 {             // если строка пустая - завершаем обработку
            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{
                Start: res[0],
                End: res[1],
            })
        }
    }
    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"
```

```
// Вспомогательная 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, "Значение")
border := fmt.Sprintf("+-%s-+-%s-+", strings.Repeat("-", maxTypeLen), strings.Repeat("-", maxValueLen))

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

return res
}

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

```


Демонстрация работы программы:

```
~/Uni/6sem/system_software/lr1_analyzer/proj @ 3:30:31
$ go run _
Содержимое входного файла:

# Простой пример с одним идентификатором и числом
id1 := 0x1A;

# Использование всех операторов
id2 := id1 and 0xFF or id3 xor not 0x0F;

# Вложенные скобки
result := ((id4 or 0x10) and (id5 xor not id6));

# Смешанный пример с комментариями
id7 := not (0xAB and id8); # Комментарий после выражения

# Пример с идентификаторами и шестнадцатеричными числами
id9 := 0x1234 or id10 and 0x5678;

# Пример без пробелов
id11:=id12xor0xAA;

# Пример с пустым значением
empty := ;

# Комментарий на отдельной строке
# Следующая строка содержит сложное выражение
complex := id13 and (not id14 or (id15 xor 0x9A)) and 0x1B;

# Неправильный идентификатор (может вызвать ошибку)
idinvalid := 0x2C;

# Пример с некорректным числом (может вызвать ошибку)
id16 := 0xGHIJ;

# Последовательные комментарии
# Это первый комментарий
# Это второй комментарий
```

Таблица лексем:

Тип	Значение
comment	# Простой пример с одним идентификатором и числом
identifier	id1
assignment	:=
hex_number	0x1A
delimiter	;
comment	# Использование всех операторов
identifier	id2
assignment	:=
identifier	id1
operator	and
hex_number	0xFF
operator	or
identifier	id3
operator	xor
operator	not
hex_number	0x0F
delimiter	;
comment	# Вложенные скобки
identifier	result
assignment	:=
parentheses	(
parentheses	(
identifier	id4
operator	or
hex_number	0x10
parentheses)
operator	and
parentheses	(
identifier	id5
operator	xor
operator	not
identifier	id6
parentheses)
parentheses)
delimiter	;
comment	# Смешанный пример с комментариями
identifier	id7
assignment	:=
operator	not
parentheses	(
hex_number	0xAB
operator	and
identifier	id8
parentheses)
delimiter	;
comment	# Комментарий после выражения
comment	# Пример с идентификаторами и шестнадцатеричными числами
identifier	id9
assignment	:=
hex_number	0x1234
operator	or
identifier	id10
operator	and
hex_number	0x5678
delimiter	;
identifier	id11
assignment	:=
identifier	id12
operator	xor
hex_number	0xAA
delimiter	;
identifier	empty
assignment	:=
delimiter	;
comment	# Комментарий на отдельной строке
comment	# Следующая строка содержит сложное выражение
identifier	id13
operator	and
operator	not
operator	or
operator	xor
hex_number	0x9A
operator)
operator	and
hex_number	0x1B
delimiter	;
comment	# Неправильный идентификатор (может вызвать ошибку)
identifier	idinvalid
assignment	:=
hex_number	0x2C
delimiter	;
comment	# Пример с некорректным числом (может вызвать ошибку)
identifier	id16
assignment	:=
hex_number	0xGHIJ
delimiter	;
comment	# Последовательные комментарии
comment	# Это первый комментарий
comment	# Это второй комментарий

```

hex_number 0x1234
operator or
identifier id10
operator and
hex_number 0x5678
delimiter ;
comment # Пример без пробелов
identifier id11
assignment :=
identifier id12xor0xAA
delimiter ;
comment # Пример с пустым значением
identifier empty
assignment :=
delimiter ;
comment # Комментарий на отдельной строке
comment # Следующая строка содержит сложное выражение
identifier complex
assignment :=
identifier id13
operator and
parentheses (
operator not
identifier id14
operator or
parentheses (
identifier id15
operator xor
hex_number 0x9A
parentheses )
parentheses )
operator and
hex_number 0x1B
delimiter ;
comment # Неправильный идентификатор (может вызвать ошибку)
error 1
identifier invalid
assignment :=
hex_number 0x2C
delimiter ;
comment # Пример с некорректным числом (может вызвать ошибку)
identifier id16
assignment :=
error 0
identifier x6H1J
delimiter ;
comment # Последовательные комментарии
comment # Это первый комментарий
comment # Это второй комментарий

```

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.