Министерство образования Республики Беларусь

Учреждение образования
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей
Кафедра информатики
Дисциплина: Методы трансляции

ОТЧЕТ
к лабораторной работе №2
на тему

ЛЕКСИЧЕСКИЙ АНАЛИЗ

Студент М. А. Щур

Преподаватель Н. Ю. Гриценко

Минск 2025

СОДЕРЖАНИЕ

1 ЦЕЛЬ РАБОТЫ

Целью выполнения данной лабораторной работы является создание лексического анализатора для подмножества языка программирования, разработанного в предыдущей лабораторной работе. Анализатор должен корректно обрабатывать входные данные, выявлять и классифицировать лексические единицы, а также выявлять и сообщать о невалидных последовательностях символов. В ходе работы необходимо продемонстрировать обнаружение и обработку четырех различных лексических ошибок.

2 КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Лексический анализатор является первой фазой компилятора, и его основная задача заключается в считывании входных символов исходной программы, их группировке в лексемы и выводе последовательностей токенов для всех лексем. Лексема представляет собой структурную единицу языка, состоящую из элементарных символов и не содержащую в себе других структурных единиц языка [1].

После формирования потока токенов, он передается синтаксическому анализатору для дальнейшего разбора. В процессе работы лексического анализатора происходит взаимодействие с таблицей символов: когда выявляется лексема, относящаяся к идентификатору, она вносится в таблицу символов. Этот механизм позволяет лексическому анализатору получать необходимую информацию об идентификаторах, что способствует корректной передаче токенов синтаксическому анализатору [2].

Вызов лексического анализатора синтаксическим анализатором обычно осуществляется через команду *NextToken*, что заставляет лексический анализатор считывать символы из входного потока до тех пор, пока не будет идентифицирована следующая лексема.

Кроме идентификации лексем, лексический анализатор выполняет дополнительные функции, такие как отбрасывание комментариев и пробельных символов. Также важной задачей является синхронизация сообщений об ошибках с исходной программой. Например, лексический анализатор может отслеживать количество строк, чтобы каждое сообщение об ошибке содержало номер строки, в которой она была обнаружена. В некоторых компиляторах лексический анализатор создает копию исходного кода с вставленными сообщениями об ошибках, что позволяет легче локализовать их в тексте программы.

3 РЕЗУЛЬТАТЫ ВЫПОЛНЕНИЯ ЛАБОРАТОРНОЙ РАБОТЫ

В ходе выполнения лабораторной работы был реализован лексический анализатор языка *Fortran*. На рисунке 3.1 представлен результат запуска программы.

```
Token: MODULE     Lexeme: module     Line: 1, Column: 6, ID: 1
Token: IDENT      Lexeme: module1    Line: 1, Column: 14, ID: 2
Token: IMPLICIT   Lexeme: implicit   Line: 2, Column: 10, ID: 3
Token: NONE       Lexeme: none       Line: 2, Column: 15, ID: 4
Token: TYPE       Lexeme: type       Line: 4, Column: 6, ID: 5
Token: IDENT      Lexeme: person     Line: 4, Column: 13, ID: 6
Token: CHARACTER  Lexeme: character  Line: 5, Column: 13, ID: 7
Token: LPAREN     Lexeme: (          Line: 5, Column: 14, ID: 8
Token: IDENT      Lexeme: len        Line: 5, Column: 17, ID: 9
Token: ASSOP      Lexeme: =          Line: 5, Column: 18, ID: 10
Token: ICONST     Lexeme: 50         Line: 5, Column: 20, ID: 11
Token: RPAREN     Lexeme: )          Line: 5, Column: 21, ID: 12
Token: DCOLON     Lexeme: ::         Line: 5, Column: 24, ID: 13
Token: IDENT      Lexeme: name       Line: 5, Column: 29, ID: 14
Token: INTEGER    Lexeme: integer    Line: 6, Column: 11, ID: 15
Token: DCOLON     Lexeme: ::         Line: 6, Column: 14, ID: 16
Token: IDENT      Lexeme: age        Line: 6, Column: 18, ID: 17
Token: REAL       Lexeme: real       Line: 7, Column: 8, ID: 18
Token: DCOLON     Lexeme: ::         Line: 7, Column: 11, ID: 19
Token: IDENT      Lexeme: height     Line: 7, Column: 18, ID: 20
Token: END        Lexeme: end        Line: 8, Column: 5, ID: 21
Token: TYPE       Lexeme: type       Line: 8, Column: 10, ID: 22
Token: IDENT      Lexeme: person     Line: 8, Column: 17, ID: 6
Token: IDENT      Lexeme: contains   Line: 10, Column: 8, ID: 23
Token: FUNCTION   Lexeme: function   Line: 12, Column: 10, ID: 24
Token: IDENT      Lexeme: create_person Line: 12, Column: 24, ID: 25
```

Рисунок 3.1 – Результат разбиения тестового кода на токены

В исходном коде намеренно введен неожиданный символ. На рисунке 3.2 изображено, что лексический анализатор отработал корректно и вывел сообщение об ошибке.

```
Token: IDENT      Lexeme: LET        Line: 1, Column: 3, ID: 1
Token: IDENT      Lexeme: x          Line: 1, Column: 5, ID: 2
Token: ASSOP      Lexeme: =          Line: 1, Column: 7, ID: 3
Token: ICONST     Lexeme: 5          Line: 1, Column: 9, ID: 4
Error: Unexpected character: $ at line 1, column 10, ID: 5
```

Рисунок 3.2 – Неожиданный символ в коде программы

В исходном коде намеренно допущена ошибка: не закрыта скобка. На рисунке 3.3 изображено, что лексический анализатор отработал корректно и вывел сообщение об ошибке.

```
Token: IF         Lexeme: IF         Line: 1, Column: 2, ID: 1
Token: LPAREN     Lexeme: (          Line: 1, Column: 4, ID: 2
Token: IDENT      Lexeme: x          Line: 1, Column: 5, ID: 3
Token: GTHAN      Lexeme: >          Line: 1, Column: 7, ID: 4
Token: ICONST     Lexeme: 10         Line: 1, Column: 10, ID: 5
Token: THEN       Lexeme: THEN       Line: 1, Column: 15, ID: 6
Error: Unmatched delimiter: '(' at line 1, column 17, ID: 7
```

Рисунок 3.3 – Незакрытая скобка в коде программы

В исходном коде намеренно допущена ошибка: не закрыты кавычки. На рисунке 3.4 изображено, что лексический анализатор отработал корректно и вывел сообщение об ошибке.

```
Token: CHARACTER  Lexeme: CHARACTER  Line: 1, Column: 9, ID: 1
Token: LPAREN     Lexeme: (          Line: 1, Column: 10, ID: 2
Token: IDENT      Lexeme: len        Line: 1, Column: 13, ID: 3
Token: ASSOP      Lexeme: =          Line: 1, Column: 14, ID: 4
Token: ICONST     Lexeme: 20         Line: 1, Column: 16, ID: 5
Token: RPAREN     Lexeme: )          Line: 1, Column: 17, ID: 6
Token: DCOLON     Lexeme: ::         Line: 1, Column: 20, ID: 7
Token: IDENT      Lexeme: str        Line: 1, Column: 24, ID: 8
Token: ASSOP      Lexeme: =          Line: 1, Column: 26, ID: 9
Error: Unmatched delimiter: '"' at line 1, column 68, ID: 10
```

Рисунок 3.4 – Незакрытая кавычка в коде программы

В исходном коде намеренно допущена ошибка: введен неизвестный оператор. На рисунке 3.5 изображено, что лексический анализатор отработал корректно и вывел сообщение об ошибке

```
Token: DCOLON     Lexeme: ::         Line: 1, Column: 21, ID: 7
Token: IDENT      Lexeme: array      Line: 1, Column: 27, ID: 8
Error: Unknown operator: : at line 1, column 29, ID: 9
```

Рисунок 3.5 – Введен неизвестный оператор в коде

На рисунке 3.6 представлен корректный код программы.

```
module module1
  implicit none

  type person
    character(len=50) :: name
    integer :: age
    real :: height
  end type person

contains

  function create_person(name_in, age_in, height_in) result(person_out)
    implicit none
    character(len=*), intent(in) :: name_in
    integer, intent(in) :: age_in
    real, intent(in) :: height_in
    type(person) :: person_out
    person_out%name = name_in
    person_out%age = age_in
    person_out%height = height_in
  end functiocontains

  function create_person(name_in, age_in, height_in) result(person_out)
    implicit none
    character(len=*), intent(in) :: name_in
    integer, intent(in) :: age_in
    real, intent(in) :: height_in
    type(person) :: person_out
    person_out%name = name_in
    person_out%age = age_in
    person_out%height = height_in
  end function create_person

  subroutine print_person_info(person_in)
    implicit none
    type(person), intent(in) :: person_in
    print *, "Имя:", person_in%name
    print *, "Возраст:", person_in%age
    print *, "Рост:", person_in%height
  end subroutine print_person_info

end module module1
```

Рисунок 3.6 – Анализируемый корректный код

ВЫВОДЫ

В процессе выполнения данной лабораторной работы был создан лексический анализатор для подмножества языка программирования *Fortran*. Также были выявлены ошибки при определении неверных последовательностей символов. Результаты работы программы при обнаружении некорректных лексем были продемонстрированы.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Лексический анализатор [Электронный ресурс]. – Режим доступа: https://csc.sibsutis.ru/sites/csc.sibsutis.ru/files/courses/trans/LabWork2.pdf – Дата доступа: 26.01.2025

[2] Лексический анализатор [Электронный ресурс]. – Режим доступа: https://old.etu.ru/misc/LGA_2007_FINAL/Allpage/Section6/part_6.1_.html. – Дата доступа: 26.01.2025.

ПРИЛОЖЕНИЕ А
(обязательное)
Листинг кода

main.go

```
package main
import (
	"fmt"
	"os"
	"strings"
	"unicode"
)
type LexItem struct {
	token  Token
	lexeme string
	line   int
	column int
	id     int
}
type Lexer struct {
	input   string
	pos     int
	line    int
	column  int
	lexeme  string
	stack   []rune
	constID int
	idMap   map[string]int
}
func (lx *Lexer) NextToken() LexItem {
	state := "START"
	for lx.pos < len(lx.input) {
		ch := lx.input[lx.pos]
		lx.pos++
		lx.column++
		switch state {
		case "START":
			if unicode.IsSpace(rune(ch)) {
				if ch == '\n' {
					lx.line++
					lx.column = 0
```

```
                        }
                        continue
                    }
                    switch {
                    case unicode.IsLetter(rune(ch)) || ch == '_':
                        lx.lexeme = string(ch)
                        state = "INID"
                    case unicode.IsDigit(rune(ch)):
                        lx.lexeme = string(ch)
                        state = "ININT"
                    case ch == '"':
                        lx.lexeme = ""
                        lx.stack = append(lx.stack, '"')
                        state = "INSTRING"
                    case ch == '(':
                        lx.stack = append(lx.stack, '(')
                        return LexItem{token: LPAREN, lexeme: "(", line:
lx.line, column: lx.column, id: lx.nextID()}
                    case ch == ')':
                        return lx.handleClosing('(', ')')
                    case ch == '[':
                        lx.stack = append(lx.stack, '[')
                        return LexItem{token: LBRACKET, lexeme: "[", line:
lx.line, column: lx.column, id: lx.nextID()}
                    case ch == ']':
                        return lx.handleClosing('[', ']')
                    case ch == '!':
                        state = "COMMENT"
                    case strings.ContainsRune("+-*/=<>.,:()%$", rune(ch)):
                        lx.lexeme = string(ch)
                        state = "SIGN"
                    default:
                        return lx.errorToken(ch, "Unexpected character")
                    }

            case "INID":
                    if unicode.IsLetter(rune(ch)) || unicode.IsDigit(rune(ch)) ||
ch == '_' {
                        lx.lexeme += string(ch)
```

```
                } else {
                        lx.pos--
                        lx.column--
                        return lx.identOrKeyword()
                }
        case "ININT":
                if unicode.IsDigit(rune(ch)) {
                        lx.lexeme += string(ch)
                } else if ch == '.' {
                        lx.lexeme += string(ch)
                        state = "INREAL"
                } else {
                        lx.pos--
                        lx.column--
                        return lx.constantOrError(ICONST)
                }
        case "INREAL":
                if unicode.IsDigit(rune(ch)) || ch == 'E' || ch == '+' || ch
== '-' {
                        lx.lexeme += string(ch)
                } else {
                        lx.pos--
                        lx.column--
                        return lx.constantOrError(RCONST)
                }
        case "INSTRING":
                if ch == '"' {
                        if len(lx.stack) == 0 || lx.stack[len(lx.stack)-1] !=
'"' {
                                return lx.errorToken(ch, "Mismatched quotes")
                        }
                        lx.stack = lx.stack[:len(lx.stack)-1]
                        return LexItem{token: SCONST, lexeme: lx.lexeme, line:
lx.line, column: lx.column, id: lx.nextID()}
                }
                lx.lexeme += string(ch)
        case "COMMENT":
                if ch == '\n' {
                        lx.line++
```

```
                        lx.column = 0
                        state = "START"
                  }
            case "SIGN":
                  if lx.lexeme == ":" && ch == ':' {
                        lx.lexeme += string(ch)
                        return LexItem{token: DCOLON, lexeme: lx.lexeme, line:
lx.line, column: lx.column, id: lx.nextID()}
                  } else if lx.lexeme == "=" && ch == '=' {
                        lx.lexeme += string(ch)
                        return LexItem{token: EQ, lexeme: lx.lexeme, line:
lx.line, column: lx.column, id: lx.nextID()}
                  } else if (lx.lexeme == "<" && ch == '=') || (lx.lexeme ==
">" && ch == '=') || (lx.lexeme == "/" && ch == '=') {
                        lx.lexeme += string(ch)
                        return lx.signToken()
                  } else {
                        lx.pos--
                        lx.column--
                        return lx.operatorOrError()
                  }
            }
      }

      if len(lx.stack) > 0 {
            return lx.errorToken(lx.stack[len(lx.stack)-1], "Unmatched
delimiter")
      }
      return LexItem{token: DONE, lexeme: "", line: lx.line, column: lx.column,
id: lx.nextID()}
}
```