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 П. Н. Носкович

Преподаватель Н. Ю. Гриценко

Минск 2025

СОДЕРЖАНИЕ

1 ЦЕЛЬ РАБОТЫ

Цель данной лабораторной работы – разработать лексический анализатор для подмножества языка программирования, созданного в предыдущей лабораторной работе. Анализатор должен правильно обрабатывать входные данные, идентифицировать и классифицировать лексические единицы, а также определять и сообщать о некорректных последовательностях символов. В ходе работы необходимо продемонстрировать обнаружение и обработку четырех различных лексических ошибок.

Лексический анализатор будет принимать на вход текстовый файл (например, *INPUT.TXT*), содержащий исходный код программы, которую необходимо проанализировать. Чтение текста будет происходить по символам, и каждый символ будет классифицироваться согласно заранее определённым лексическим правилам.

Когда лексический анализатор будет читать символы из исходного кода, он будет собирать их в лексемы, которые будут представлять собой токены. Токен – это абстрактная единица, которая несёт информацию о типе лексемы (например, целое число или идентификатор) и самой её строковой форме.

Для каждой ошибки лексический анализатор должен генерировать соответствующее сообщение, указывающее на тип ошибки и позицию, на которой она возникла, для дальнейшей корректировки исходного кода.

В конце работы необходимо показать, как лексический анализатор успешно обрабатывает входной файл с программой, распознавая и классифицируя лексемы, а также корректно сообщает об ошибках. В качестве примера можно использовать небольшой фрагмент программы, содержащий как правильные, так и ошибочные лексемы.

2 КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Лексический анализатор представляет собой первую фазу компилятора, его основная задача состоит в чтении входных символов исходной программы, их группировании в лексемы и вывод последовательностей токенов для всех лексем исходной программы.

Лексема – это структурная единица языка, которая состоит из элементарных символов языка и не содержит в своём составе 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 *parse*, 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 *PHP*. На рисунке 3.1 представлен результат запуска программы.

```
Token:      KEY   Lexeme:              int   Line:    2, Column:    1, ID:    2, Info: integer
Token:       ID   Lexeme:               $a   Line:    2, Column:    5, ID:    3, Info: -
Token:       OP   Lexeme:                =   Line:    2, Column:    8, ID:    4, Info: -
Token:    CONST   Lexeme:                5   Line:    2, Column:   10, ID:    5, Info: integer
Token:       OP   Lexeme:                ;   Line:    2, Column:   11, ID:    6, Info: -
Token:      KEY   Lexeme:            float   Line:    3, Column:    1, ID:    7, Info: floating
Token:       ID   Lexeme:               $b   Line:    3, Column:    7, ID:    8, Info: -
Token:       OP   Lexeme:                =   Line:    3, Column:   10, ID:    4, Info: -
Token:    CONST   Lexeme:             3.14   Line:    3, Column:   12, ID:    9, Info: floating
Token:       OP   Lexeme:                ;   Line:    3, Column:   16, ID:    6, Info: -
Token:       ID   Lexeme:         $stringVar   Line:    4, Column:    1, ID:   10, Info: -
Token:       OP   Lexeme:                =   Line:    4, Column:   12, ID:    4, Info: -
Token:      STR   Lexeme:       Hello, PHP!   Line:    4, Column:   16, ID:   11, Info: string
Token:       OP   Lexeme:                ;   Line:    4, Column:   27, ID:    6, Info: -
Token:       ID   Lexeme:            $flag   Line:    5, Column:    1, ID:   12, Info: -
Token:       OP   Lexeme:                =   Line:    5, Column:    7, ID:    4, Info: -
Token:    CONST   Lexeme:             true   Line:    5, Column:    9, ID:   13, Info: boolean
Token:       OP   Lexeme:                ;   Line:    5, Column:   13, ID:    6, Info: -
Token:      KEY   Lexeme:               if   Line:    7, Column:    1, ID:   14, Info: keyword
Token:       OP   Lexeme:                (   Line:    7, Column:    4, ID:   15, Info: -
Token:       ID   Lexeme:               $a   Line:    7, Column:    5, ID:    3, Info: -
Token:       OP   Lexeme:                >   Line:    7, Column:    8, ID:   16, Info: -
Token:    CONST   Lexeme:                3   Line:    7, Column:   10, ID:   17, Info: integer
```

Рисунок 3.1 – Выполнение программы

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

```
    OP   Lexeme:                    ;   Line:    5, Column:   13, ID:   19
 ERROR   Lexeme:   Invalid symbol: @   Line:    6, Column:    1, ID:   20
   KEY   Lexeme:                   if   Line:    7, Column:    1, ID:   21
```

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.

```
        ID   Lexeme:          $stringVar   Line:    4, Column:    1, ID:   10,
        OP   Lexeme:                   =   Line:    4, Column:   12, ID:    4,
     ERROR   Lexeme: Unterminated string   Line:    4, Column:   27, ID:   11,
```

Рисунок 3.3 – Незакрытая скобка в коде программы

В исходном коде намеренно допущена ошибка: введен неизвестный оператор. На рисунке 3.4 изображено, что лексический анализатор отработал корректно и вывел сообщение об ошибке.

```
Token:      CONST    Lexeme:                  3    Line:    6, Column:    1, ID:   20
Token:      ERROR    Lexeme:    Invalid symbol: ^    Line:    6, Column:    2, ID:   21
Token:      CONST    Lexeme:                  2    Line:    6, Column:    3, ID:   22
```

Рисунок 3.4 – Введен неизвестный оператор в коде

В исходном коде намеренно допущена ошибка: не закрыты кавычки. На рисунке 3.5 изображено, что лексический анализатор отработал корректно и вывел сообщение об ошибке.

```
        ID    Lexeme:           $stringVar    Line:    4, Column:    1, ID:   12
        OP    Lexeme:                    =    Line:    4, Column:   12, ID:   13
     ERROR    Lexeme: Unterminated string    Line:    4, Column:   27, ID:   14
```

Рисунок 3.5 – Незакрытая кавычка в коде программы

На рисунке 3.6 представлен корректный код программы.

```php
<?php
int $a = 5;
float $b = 3.14;
$stringVar = "Hello, PHP!";
$flag = true;

if ($a > 3) {
    $result = $a + $b;
    echo "Result is: "  $result;
} else {
    $result = $a - $b;
    echo "Different result: "  $result;
}

while ($a < 10) {
    $a = $a + 1;
    $stringVar = $stringVar  "!";
}
```

Рисунок 3.6 – Анализируемый корректный код

```php
for ($i = 0; $i < 3; $i++) {
    $flag = $flag;
    echo "Flag is: "  $flag;
}

function testFunction($param1, $param2) {
    return $param1 * $param2;
}

$multResult = testFunction(5);
echo "Multiplication result: "  $multResult;

class MyClass {
    public $myVar;

    function construct($value) {
        $this->myVar = $value;
    }

    function getVar() {
        return $this->myVar;
    }
}

$myObject = new MyClass(42);
echo "Object value: "  $myObject->getVar();
?>
```

Рисунок 3.6 – Анализируемый корректный код. Лист 2

Проведенные на различных входных данных тесты подтвердили корректность работы анализатора и его способность выявлять как валидные, так и невалидные конструкции.

ЗАКЛЮЧЕНИЕ

В ходе выполнения данной лабораторной работы был создан лексический анализатор для обработки подмножества языка программирования *PHP*. Основной задачей было анализировать исходный код, разбивая его на отдельные лексемы (ключевые слова, операторы, идентификаторы, константы и так далее) и проверять их корректность. В результате работы программы каждая строка исходного кода подвергалась тщательному разбору, и в случае обнаружения некорректных лексем выводился отчет с указанием типа ошибки и позиции в исходном коде.

Кроме того, были протестированы различные элементы языка *PHP*, включая переменные разных типов (целые числа, числа с плавающей запятой, строки, логические переменные), операторы присваивания, арифметические операторы и условные конструкции. В результате работы лексического анализатора программа корректно разбивала исходный код на отдельные лексемы и адекватно реагировала на ошибки синтаксиса.

Лексический анализатор успешно выполняет свою задачу по разбору исходного кода на лексемы, выявлению синтаксических ошибок и демонстрации результатов обработки каждой строки. В ходе работы была продемонстрирована способность программы распознавать ошибки в коде и выводить их с точной информацией о месте возникновения.

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: 15.02.2025

[2] Введение в теорию компиляторов [Электронный ресурс]. – Режим доступа: https://habr.com/ru/articles/515420/

ПРИЛОЖЕНИЕ А
(обязательное)
Листинг кода

lexical_analyzer.f90

```
program lexical_analyzer
implicit none

! Ключевые слова PHP
character(len=10), dimension(12) :: keywords = (/ 'if      ', 'else    ',
'while   ', 'return  ', &
                                                  'int     ', 'float   ',
'string  ', 'bool    ', &
                                                  'function', 'class   ',
'echo    ', 'for     ' /)

! Операторы PHP (унифицированная длина 2 символа)
character(len=2), dimension(15) :: operators = (/ '= ', '==', '+ ', '- ', '*
', '/ ', '( ', ') ', '{ ', '} ', '; ', '< ', '> ', '? ', ', ' /)

! Теги PHP
character(len=5), dimension(2) :: tags = (/ '<?php', '?>   ' /)

! Исходная строка программы
character(len=200) :: line
integer :: i, len_line, line_number, io_status
character(len=1) :: ch

! Таблица имен
type :: name_table_entry
    character(len=20) :: name
    integer :: id
end type name_table_entry

type(name_table_entry), dimension(100) :: name_table
integer :: name_table_size = 0

! Счетчик токенов
integer :: token_id = 0

! Стек для отслеживания открывающих скобок
integer, dimension(100) :: bracket_stack
integer :: bracket_stack_size = 0

! Открытие файла
open(unit=10, file='INPUT.TXT', status='old', action='read')
line_number = 0

! Чтение файла построчно
do
    read(10, '(A)', iostat=io_status) line
    if (io_status /= 0) exit

    line_number = line_number + 1
    len_line = len_trim(line)
    i = 1

    ! Анализ строки
    do while (i <= len_line)
        ch = line(i:i)

        select case (ch)
```

```
        case ('A':'Z', 'a':'z', '$')
            call process_identifier(line, i, len_line, name_table,
name_table_size, line_number, keywords)
        case ('0':'9')
            call process_constant(line, i, len_line, name_table,
name_table_size, line_number)
        case ('=', '+', '-', '*', '/', '(', ')', '{', '}', ';', '<', '>',
'?', ',')
            ! Обработка тегов
            if (i < len_line .and. line(i:i+4) == '<?php') then
                call print_token('TAG', '<?php', line_number, i)
                i = i + 5
            else if (i < len_line .and. line(i:i+1) == '?>') then
                call print_token('TAG', '?>', line_number, i)
                i = i + 2
            else
                call process_operator(line, i, len_line, line_number,
operators)
            end if
        case ('"', '''')
            call process_string(line, i, len_line, name_table,
name_table_size, line_number)
        case (' ')
            i = i + 1
        case default
            call print_token('ERROR', 'Invalid symbol: ' // ch, line_number,
i)
            i = i + 1
        end select
    end do

    ! Проверка на незакрытые скобки в конце строки
    if (bracket_stack_size > 0) then
        call print_token('ERROR', 'Unclosed bracket at line ', line_number,
bracket_stack(bracket_stack_size))
        bracket_stack_size = 0
    end if
end do

close(10)

contains

subroutine print_token(token_type, lexeme, line_num, column)
    character(len=*), intent(in) :: token_type, lexeme
    integer, intent(in) :: line_num, column
    integer :: token_id_local

    token_id = token_id + 1
    token_id_local = token_id

    ! Форматированный вывод с фиксированной шириной столбцов
    write(*, '(A, A12, A, A20, A, I4, A, I4, A, I4)') &
        'Token: ', trim(token_type), &
        '    Lexeme: ', trim(lexeme), &
        '    Line: ', line_num, &
        ', Column: ', column, &
        ', ID: ', token_id_local
end subroutine print_token

subroutine process_identifier(line, i, len_line, name_table, name_table_size,
line_number, keywords)
    character(len=200), intent(in) :: line
    integer, intent(inout) :: i
```

```
    integer, intent(in) :: len_line
    type(name_table_entry), dimension(100), intent(inout) :: name_table
    integer, intent(inout) :: name_table_size
    integer, intent(in) :: line_number
    character(len=10), dimension(12), intent(in) :: keywords

    character(len=20) :: identifier
    character(len=10) :: buffer
    integer :: j, id

    identifier = ''
    j = 0

    ! Собираем идентификатор
    do while (i <= len_line .and. (line(i:i) >= 'A' .and. line(i:i) <= 'Z'
.or. &
                                   line(i:i) >= 'a' .and. line(i:i) <= 'z'
.or. &
                                   line(i:i) >= '0' .and. line(i:i) <= '9'
.or. &
                                   line(i:i) == '$'))
        j = j + 1
        identifier(j:j) = line(i:i)
        i = i + 1
    end do

    identifier = trim(identifier)

    ! Проверка, является ли идентификатор ключевым словом
    do j = 1, size(keywords)
        if (identifier == trim(keywords(j))) then
            call print_token('KEY', identifier, line_number, i -
len_trim(identifier))
            return
        end if
    end do

    ! Проверка, есть ли идентификатор в таблице
    id = -1
    do j = 1, name_table_size
        if (trim(name_table(j)%name) == identifier) then
            id = name_table(j)%id
            exit
        end if
    end do

    if (id == -1) then
        ! Если идентификатор новый, добавляем его в таблицу
        name_table_size = name_table_size + 1
        name_table(name_table_size)%name = identifier
        name_table(name_table_size)%id = name_table_size
        id = name_table_size
    end if

    ! Выводим токен с соответствующим ID
    write(buffer, '(I0)') id
    call print_token('ID', identifier, line_number, i - len_trim(identifier))
end subroutine process_identifier

subroutine process_constant(line, i, len_line, name_table, name_table_size,
line_number)
    character(len=200), intent(in) :: line
    integer, intent(inout) :: i
    integer, intent(in) :: len_line
```

```
    type(name_table_entry), dimension(100), intent(inout) :: name_table
    integer, intent(inout) :: name_table_size
    integer, intent(in) :: line_number

    character(len=200) :: constant
    character(len=10) :: buffer
    integer :: j
    logical :: has_dot

    constant = ''
    j = 0
    has_dot = .false.

    do while (i <= len_line .and. (line(i:i) >= '0' .and. line(i:i) <= '9'
.or. line(i:i) == '.'))
        if (line(i:i) == '.' .and. has_dot) exit
        if (line(i:i) == '.') has_dot = .true.
        j = j + 1
        constant(j:j) = line(i:i)
        i = i + 1
    end do

    constant = trim(constant)

    ! Заполняем таблицу символов
    name_table_size = name_table_size + 1
    name_table(name_table_size)%name = constant
    name_table(name_table_size)%id = name_table_size

    write(buffer, '(I0)') name_table_size
    call print_token('CONST', constant, line_number, i - len_trim(constant))
end subroutine process_constant

subroutine process_string(line, i, len_line, name_table, name_table_size,
line_number)
    character(len=200), intent(in) :: line
    integer, intent(inout) :: i
    integer, intent(in) :: len_line
    type(name_table_entry), dimension(100), intent(inout) :: name_table
    integer, intent(inout) :: name_table_size
    integer, intent(in) :: line_number

    character(len=200) :: str
    character(len=10) :: buffer
    integer :: j
    character(len=1) :: quote_char

    str = ''
    j = 0
    quote_char = line(i:i)
    i = i + 1

    do while (i <= len_line .and. line(i:i) /= quote_char)
        j = j + 1
        str(j:j) = line(i:i)
        i = i + 1
    end do

    if (i <= len_line) then
        i = i + 1
    else
        call print_token('ERROR', 'Unterminated string at line ' //
trim(str), line_number, i)
        return
```

```
    end if

    str = trim(str)

    ! Заполняем таблицу символов
    name_table_size = name_table_size + 1
    name_table(name_table_size)%name = str
    name_table(name_table_size)%id = name_table_size

    write(buffer, '(I0)') name_table_size
    call print_token('STR', str, line_number, i - len_trim(str))
end subroutine process_string

subroutine process_operator(line, i, len_line, line_number, operators)
    character(len=200), intent(in) :: line
    integer, intent(inout) :: i
    integer, intent(in) :: len_line
    integer, intent(in) :: line_number
    character(len=2), dimension(15), intent(in) :: operators

    character(len=2) :: op
    integer :: j

    if (i < len_line .and. line(i:i+1) == '==') then
        op = '=='
        i = i + 1
    else
        op = line(i:i)
    end if
    i = i + 1

    ! Проверка на допустимый оператор
    do j = 1, size(operators)
        if (op == trim(operators(j))) then
            if (op == '(') then
                ! Добавляем открывающую скобку в стек
                bracket_stack_size = bracket_stack_size + 1
                bracket_stack(bracket_stack_size) = i - 1
            else if (op == ')') then
                ! Проверяем, есть ли соответствующая открывающая скобка
                if (bracket_stack_size > 0) then
                    bracket_stack_size = bracket_stack_size - 1
                else
                    call print_token('ERROR', 'Unmatched closing bracket',
line_number, i - 1)
                end if
            end if
            call print_token('OP', op, line_number, i - len_trim(op))
            return
        end if
    end do

    call print_token('ERROR', 'Invalid operator: ' // op, line_number, i -
len_trim(op))
end subroutine process_operator

end program lexical_analyzer
```