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 3](#_Toc160203922)

[2 Краткие теоретические сведения 4](#_Toc160203923)

[3 Результаты выполнения лабораторной работы 5](#_Toc160203924)

[Заключение **Ошибка! Закладка не определена.**](#_Toc160203925)

[Cписок использованных источников 9](#_Toc160203926)

[Приложение А (обязательное) Листинг кода 10](#_Toc160203927)

## 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.

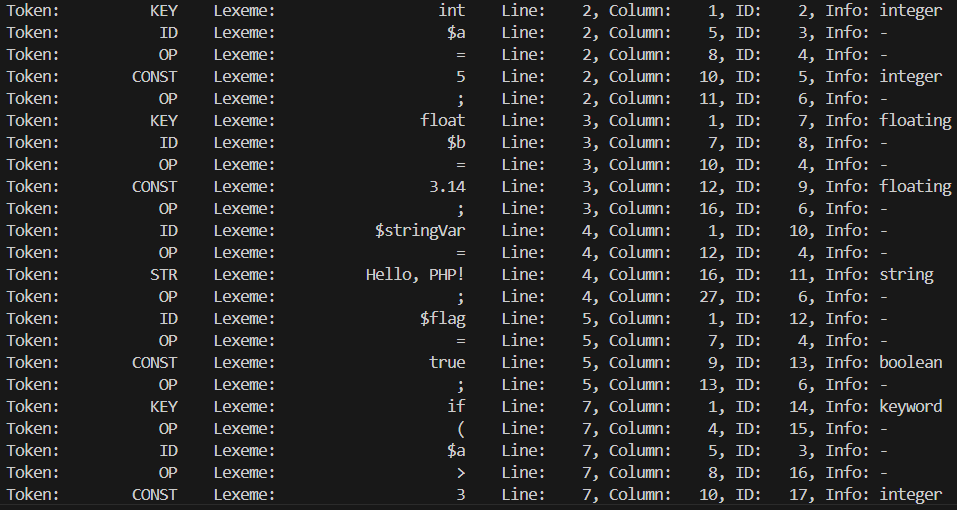


Рисунок 3.1 – Выполнение программы

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

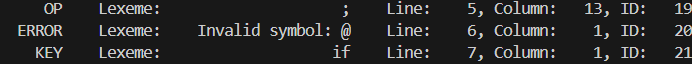


Рисунок 3.2 – Неожиданный символ в коде программы

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

Изображение выглядит как текст, снимок экрана, Шрифт, линия

Контент, сгенерированный ИИ, может содержать ошибки.

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.



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.

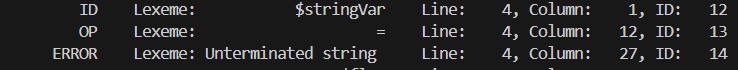


Рисунок 3.5 – Незакрытая кавычка в коде программы

На рисунке 3.6 представлен корректный код программы.

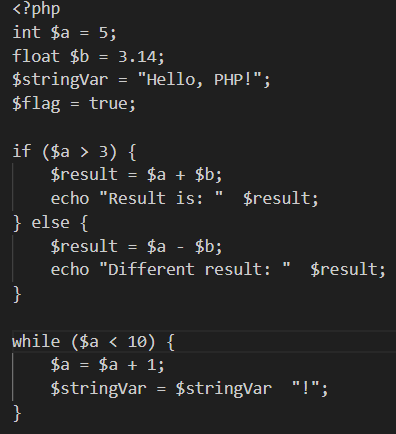


Рисунок 3.6 – Анализируемый корректный код

Изображение выглядит как текст, снимок экрана, Шрифт

Контент, сгенерированный ИИ, может содержать ошибки.

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