

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 № 3
на тему

СИНТАКСИЧЕСКИЙ АНАЛИЗАТОР

Студент

П. Н. Носкович

Преподаватель

Н. Ю. Гриценко

Минск 2025

СОДЕРЖАНИЕ

1 Постановка задачи.....	3
2 Краткие теоретические сведения.....	4
3 Результаты выполнения лабораторной работы.....	5
Выводы	7
Список использованных источников	8
Приложение А(обязательное) Листинг программного кода	9

1 ПОСТАНОВКА ЗАДАЧИ

Цель данной лабораторной работы заключается в разработке синтаксического анализатора для потока токенов, который был получен с помощью лексического анализатора, выполненного в рамках второй лабораторной работы. Также акцентируется внимание на выявлении ошибок, возникающих при обработке неправильной последовательности токенов.

2 КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Синтаксический анализ – это ключевой этап в многоступенчатом процессе создания компилятора для языков программирования. На этом этапе происходит глубокая проверка входной последовательности символов (строки кода) с целью выяснить, насколько она соответствует заранее определенным правилам и структурам, указанным в формальной грамматике языка.

Синтаксический анализ начинается после завершения первого этапа, известного как лексический анализ. По итогам успешного синтаксического анализа формируется так называемое дерево разбора или синтаксическое дерево, которое строится на основе грамматики, установленной для данного языка. Синтаксический анализатор проверяет код программы на соответствие правилам, определенным в контекстно-свободной грамматике. Если входные данные соответствуют требованиям, анализатор создает соответствующее дерево разбора для исходного кода.

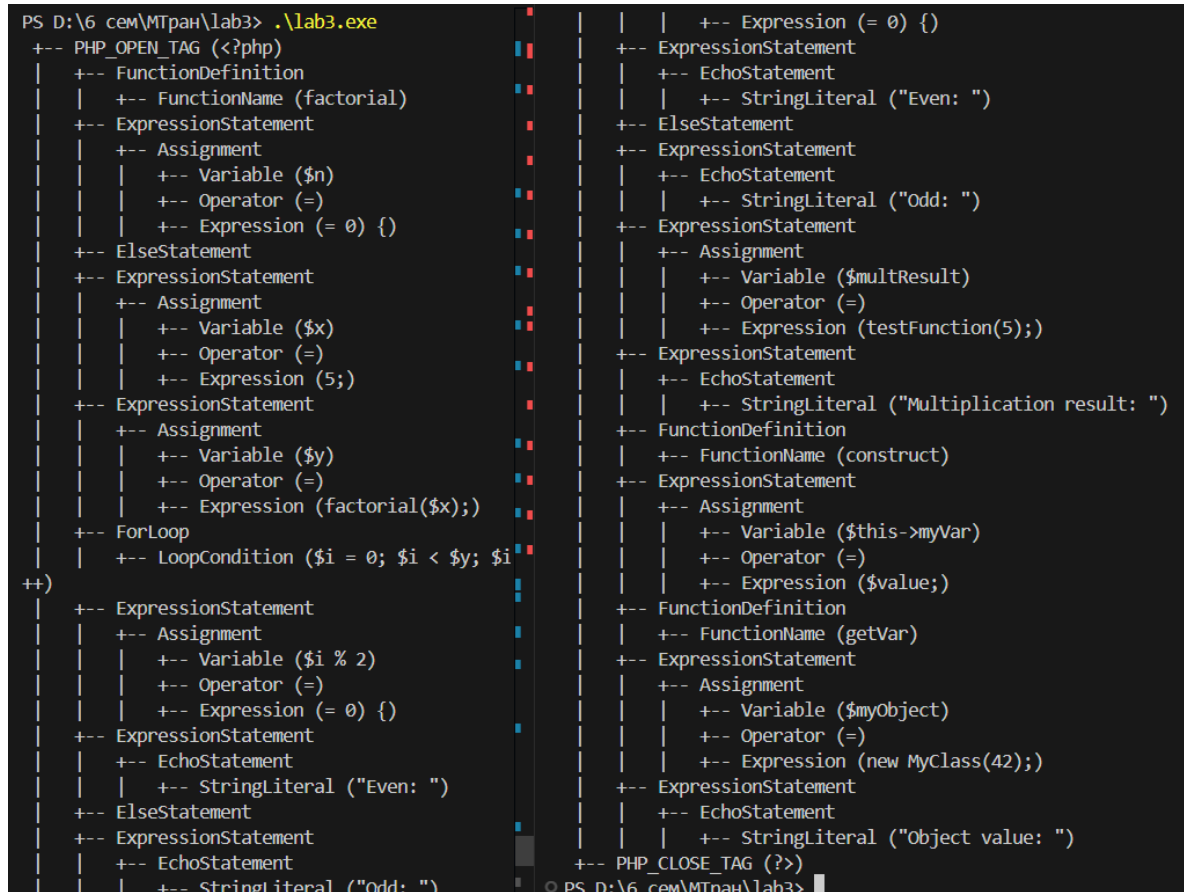
Существуют различные подходы к синтаксическому анализу, которые можно разделить на две основные категории: нисходящие и восходящие методы. Нисходящие методы начинают построение дерева разбора с его верхней части (корня) и движутся к листьям. В этой категории выделяют два основных типа: прогнозирующий анализ и рекурсивный анализ спуска.

Прогнозирующий анализ способен предугадать, какое правило грамматики (продукция) следует применить для обработки конкретной входной строки в процессе синтаксического разбора. Этот метод использует механизм, известный как «точка просмотра вперед», позволяя анализатору учитывать последующие символы во входной строке для принятия решений без возврата. Прогнозирующий анализатор также называют парсером *LL*, где «*LL*» указывает на сканирование строки слева направо и построение вывода слева направо с одним символом вперед для предсказания [1]. Рекурсивный анализ спуска, в свою очередь, обрабатывает входную строку рекурсивно, создавая дерево фраз, и включает ряд небольших функций, каждая из которых отвечает за разбор определенного нетерминального символа грамматики.

Методы восходящего анализа, напротив, начинают сбор дерева разбора с листьев и постепенно поднимаются к корню. Эти методы часто применяются в современных компиляторах и реализуются с помощью различных инструментов, таких как генераторы синтаксических анализаторов, которые автоматизируют и упрощают процесс создания анализаторов для разработчиков компиляторов [2].

3 РЕЗУЛЬТАТЫ ВЫПОЛНЕНИЯ ЛАБОРАТОРНОЙ РАБОТЫ

Во время выполнения лабораторной работы был реализован синтаксический анализатор языка C. Скриншот результатов представлен на рисунке 3.1.



```
PS D:\6 сем\МТран\lab3> .\lab3.exe
+-- PHP_OPEN_TAG (<?php)
|   +-- FunctionDefinition
|   |   +-- FunctionName (factorial)
|   +-- ExpressionStatement
|   |   +-- Assignment
|   |   |   +-- Variable ($n)
|   |   |   +-- Operator (=)
|   |   |   +-- Expression (= 0) {}
|   +-- ElseStatement
|   +-- ExpressionStatement
|   |   +-- Assignment
|   |   |   +-- Variable ($x)
|   |   |   +-- Operator (=)
|   |   |   +-- Expression (5;)
|   +-- ExpressionStatement
|   |   +-- Assignment
|   |   |   +-- Variable ($y)
|   |   |   +-- Operator (=)
|   |   |   +-- Expression (factorial($x);)
|   +-- ForLoop
|   |   +-- LoopCondition ($i = 0; $i < $y; $i++)
|   |   +-- ExpressionStatement
|   |   |   +-- Assignment
|   |   |   |   +-- Variable ($i % 2)
|   |   |   |   +-- Operator (=)
|   |   |   |   +-- Expression (= 0) {}
|   |   +-- ExpressionStatement
|   |   |   +-- EchoStatement
|   |   |   |   +-- StringLiteral ("Even: ")
|   +-- ElseStatement
|   +-- ExpressionStatement
|   |   +-- EchoStatement
|   |   |   +-- StringLiteral ("Odd: ")
|   +-- PHP_CLOSE_TAG (?>)
|   +-- ExpressionStatement
|   |   +-- EchoStatement
|   |   |   +-- StringLiteral ("Multiplication result: ")
|   +-- FunctionDefinition
|   |   +-- FunctionName (construct)
|   +-- ExpressionStatement
|   |   +-- Assignment
|   |   |   +-- Variable ($this->myVar)
|   |   |   +-- Operator (=)
|   |   |   +-- Expression ($value;)
|   +-- FunctionDefinition
|   |   +-- FunctionName (getVar)
|   +-- ExpressionStatement
|   |   +-- Assignment
|   |   |   +-- Variable ($myObject)
|   |   |   +-- Operator (=)
|   |   |   +-- Expression (new MyClass(42);)
|   +-- ExpressionStatement
|   |   +-- EchoStatement
|   |   |   +-- StringLiteral ("Object value: ")
|   +-- PHP_CLOSE_TAG (?>)
```

Рисунок 3.1 – Итог работы синтаксического анализатора

Изначальное содержание файла с анализируемым программным кодом представлено на рисунке 3.2.

```
<?php
function factorial($n) {
    if ($n == 0) {
        return 1;
    } else {
        return $n * factorial($n - 1);
    }
}

$x = 5;
$y = factorial($x);

for ($i = 0; $i < $y; $i++) {
    if ($i % 2 == 0) {
        echo "Even: " . $i;
    } else {
        echo "Odd: " . $i;
    }
}

$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 3.2 – Изначальное содержание текстового файла

ВЫВОДЫ

В ходе выполнения лабораторной работы по написанию синтаксического анализатора были изучены основные принципы синтаксического анализа, алгоритмы разбора и структуры данных, необходимые для построения дерева разбора. Был реализован синтаксический анализатор, способный проверять правильность порядка лексем в соответствии с грамматикой языка.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Синтаксический анализ [Электронный ресурс]. – Режим доступа: <https://edu.vsu.ru/mod/resource/view.php?id=25354> – Дата доступа: 05.03.2024

[2] Введение в теорию компиляторов [Электронный ресурс]. – Режим доступа: <https://habr.com/ru/articles/515420/> – Дата доступа: 05.03.2024

ПРИЛОЖЕНИЕ А
(обязательное)
Листинг программного кода

```
program php_parser
  implicit none
  character(len=1000) :: line
  integer :: iostat

  ! Открываем файл INPUT.TXT
  open(unit=10, file="INPUT.TXT", status="old", action="read",
iostat=iostat)
  if (iostat /= 0) then
    print *, "Ошибка: не удалось открыть файл INPUT.TXT"
    stop
  end if

  print *, "+-- PHP_OPEN_TAG (<?php)"

  ! Читаем файл построчно
  do
    read(10, '(A)', iostat=iostat) line
    if (iostat /= 0) exit
    call process_line(trim(line))
  end do

  print *, "+-- PHP_CLOSE_TAG (?>)"

  close(10)

contains

  subroutine process_line(line)
    character(len=*) :: line
    character(len=100) :: token

    if (index(line, "echo") /= 0) then
      print *, "|  +-- ExpressionStatement"
      print *, "|  |  +-- EchoStatement"
      call extract_string(line, token)
      print *, "|  |  +-- StringLiteral (", trim(token), ")"
    else if (index(line, "=") /= 0 .and. index(line, "echo") == 0 .and.
index(line, "for") == 0) then
      print *, "|  +-- ExpressionStatement"
      print *, "|  |  +-- Assignment"
      call extract_variable(line, token)
      print *, "|  |  +-- Variable (", trim(token), ")"
      print *, "|  |  +-- Operator (=)"
      call extract_expression(line, token)
    else if (index(line, "if") /= 0) then
      print *, "|  +-- IfStatement"
      call extract_condition(line, token)
      print *, "|  |  +-- Condition (", trim(token), ")"
    else if (index(line, "else") /= 0) then
      print *, "|  +-- ElseStatement"
    else if (index(line, "for") /= 0) then
      print *, "|  +-- ForLoop"
      call extract_for_loop(line)
    else if (index(line, "while") /= 0) then
      print *, "|  +-- WhileLoop"
    else if (index(line, "function") /= 0) then
      print *, "|  +-- FunctionDefinition"
```

```

        call extract_function(line, token)
        print *, "|  |  |  +-- FunctionName (", trim(token), ")"
endif
end subroutine process_line

subroutine extract_string(line, token)
    character(len=*) :: line
    character(len=100) :: token
    integer :: start, stop

    start = index(line, '"')
    stop = index(line(start+1:), '"') + start
    if (start > 0 .and. stop > start) then
        token = line(start:stop)
    else
        token = "Error"
    end if
end subroutine extract_string

subroutine extract_variable(line, token)
    character(len=*) :: line
    character(len=100) :: token
    integer :: pos

    pos = index(line, "$")
    if (pos > 0) then
        token = adjustl(line(pos:index(line, "=")-1))
    else
        token = "Error"
    end if
end subroutine extract_variable

subroutine extract_expression(line, token)
    character(len=*) :: line
    character(len=100) :: token
    integer :: pos

    pos = index(line, "=") + 1
    if (pos > 1) then
        token = adjustl(line(pos:))
        print *, "|  |  |  +-- Expression (", trim(token), ")"
    else
        token = "Error"
    end if
end subroutine extract_expression

subroutine extract_condition(line, token)
    character(len=*) :: line
    character(len=100) :: token
    integer :: start, stop

    start = index(line, "(")
    stop = index(line, ")")
    if (start > 0 .and. stop > start) then
        token = line(start+1:stop-1)
    else
        token = "Error"
    end if
end subroutine extract_condition

subroutine extract_for_loop(line)
    character(len=*) :: line
    character(len=100) :: token
    integer :: start, stop

```

```

    start = index(line, "(")
    stop = index(line, ")")
    if (start > 0 .and. stop > start) then
        token = line(start+1:stop-1)
        print *, "|    |    +-- LoopCondition (", trim(token), ")"
    else
        print *, "|    |    +-- Error: неверный синтаксис цикла for"
    end if
end subroutine extract_for_loop

subroutine extract_function(line, token)
    character(len=*) :: line
    character(len=100) :: token
    integer :: start, stop

    start = index(line, "function") + 9
    stop = index(line, "(") - 1
    if (start > 0 .and. stop > start) then
        token = line(start:stop)
    else
        token = "Error"
    end if
end subroutine extract_function

end program php_parser

```