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»

Выполнил             Я. Ю. Прескурел

Проверил                          Н. Ю. Гриценко

Минск 2024

СОДЕРЖАНИЕ

[1 Постановка задачи 3](#_Toc158040802)

[2 Краткие теоретические сведения 4](#_Toc158040803)

[3 Результаты выполнения лабораторной работы 5](#_Toc158040804)

[Выводы 7](#_Toc158040805)

[Список использованных источников 8](#_Toc158040806)

[Приложение А (обязательное) Листинг исходного кода 9](#_Toc158040807)

## 1 ПОСТАНОВКА ЗАДАЧИ

Целью выполнения данной лабораторной работы является разработка лексического анализатора подмножества языка программирования в Python. Также необходимо определить лексические правила и выполнить перевод потока символов в поток токенов, при определении неверной последовательности символов необходимо обнаружить ошибку и выдать сообщение о ней.

## 2 КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Первая фаза компиляции называется лексическим анализом   
или сканированием. Лексический анализатор читает поток символов, составляющих исходную программу, и группирует эти символы в значащие последовательности, называющиеся лексемами.[1]

Лексема – это структурная единица языка, которая состоит   
из элементарных символов и не содержит в своем составе других структурных единиц языка. Лексемами языка программирования являются идентификаторы, константы, ключевые слова языка, знаки операции. На вход лексического анализатора поступает текст исходной программы, а выходная информация передается для дальнейшей обработки синтаксическому анализатору. Для каждой лексемы анализатор строит выходной токен, где имя токена связано с его значением в коде.

Использование лексического анализатора упрощает работу с текстом исходной программы на этапе синтаксического разбора и сокращается объем обрабатываемой информации. Для выделения в тексте и разбора лексем можно применять простую и эффективную технику анализа, в то время как на этапе синтаксического анализа конструкций исходного языка используются достаточно сложные алгоритмы. Лексический анализатор отделяет сложный по конструкции синтаксический анализатор от работы непосредственно   
с текстом исходной программы. В большинстве компиляторов лексический   
и синтаксический анализаторы – это взаимосвязанные части.

При написании данной лабораторной работы были применены следующие теоретические сведения и концепции:

1 Словари языка программирования C#: при помощи использования словарей были представлены наборы ключевых слов, операторов и типов данных в языке программирования Python. Эти словари используются   
для классификации токенов в программе.

2 Приведение типов: при помощи приведения типов были созданы проверки строк, чтобы выяснить, является ли она целым числом, символом, числом с плавающей точкой, логическим значением или строкой соответственно.

3 Циклы и ветвления языка программирования C#: при помощи разнообразных циклов и ветвлений были организованы разбитие исходного кода на токены, а также классификация каждого токена на основе его типа.

4 Чтение из файла: для работы с кодом были использованы функции чтения из файла для удобства работы.

Все вышеперечисленные концепции были использованы для написания лексического анализатора подмножества языка программирования в Python, а также для определения лексических правил и перевода потока символов   
в поток токенов.

# 3 РЕЗУЛЬТАТЫ ВЫПОЛНЕНИЯ ЛАБОРАТОРНОЙ РАБОТЫ

В ходе лабораторной работы был разработан лексический анализатор для языка программирования Python. Для работы с кодом используются файлы. При запуске программы код анализируется и разбивается на токены. Результат работы лексического анализатора представлен на рисунке 3.1.

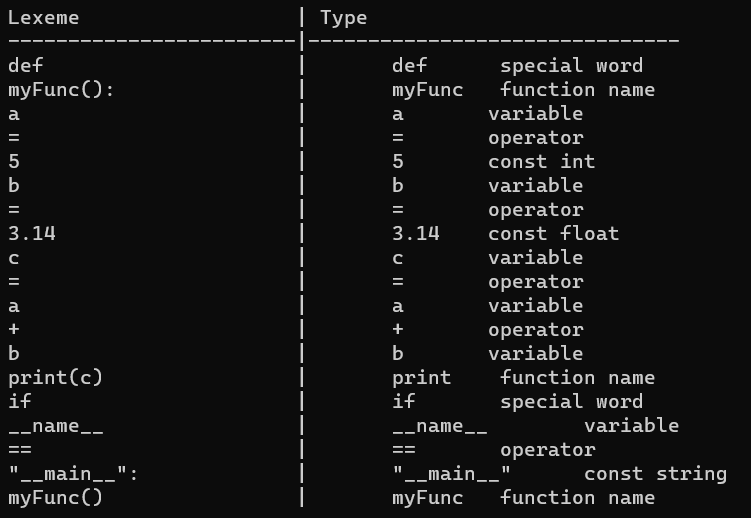


Рисунок 3.1 – Результат работы лексического анализатора

Кроме вывода лексем и их значений программа обрабатывает некоторые ошибки в коде. Если совершить попытку дать имя переменной, начиная его с цифры, то это будет обозначено в таблице лексем как ошибка. Результат нахождения данной ошибки представлен на рисунке 3.2.



Рисунок 3.2 – Результат нахождения ошибки при неправильном названии переменной

Если неправильно написать ключевое слово, то в таблице этот момент будет представлен как ошибка. Результат нахождения данной ошибки представлен на рисунке 3.3.



Рисунок 3.3 – Результат нахождения ошибки при неправильном написании ключевого слова

Если при использовании ключевого слова будет пропущен символ, то в таблице этот момент будет представлен как ошибка. Результат нахождения данной ошибки представлен на рисунке 3.4.



Рисунок 3.4 – Результат нахождения ошибки при пропущенном символе

Если допустить ошибку при написании def, то данный момент в таблице будет представлен как ошибка. Результат нахождения данной ошибки представлен на рисунке 3.5.



Рисунок 3.5 – Результат нахождения ошибки при написании функции

Таким образом, по итогу лабораторной работы был разработан лексический анализатор кода, написанного на языке программирования С#, а также реализовано нахождение разного рода лексических ошибок.

## ВЫВОДЫ

В ходе выполнения данной лабораторной работы был разработан лексический анализатор подмножества языка программирования Python. Также были определены лексические правила и выполнен перевод потока символов в поток токенов. При определении неверной последовательности символов была реализована возможность обнаружение ошибок и демонстрация сообщений о данных ошибках.

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Лексический анализатор [Электронный ресурс]. – Режим доступа: https://csc.sibsutis.ru/sites/csc.sibsutis.ru/files/courses/trans/. – Дата доступа: 18.02.2024.
2. Введение в Python [Электронный ресурс]. – Режим доступа: https://metanit.com/python/tutorial/1.1.php. – Дата доступа: 18.02.2024.
3. Типы данных [Электронный ресурс]. – Режим доступа: https://metanit.com/python/tutorial/2.2.php. – Дата доступа: 18.02.2024.
4. Операторы в Python [Электронный ресурс]. – Режим доступа: <https://learn.microsoft.com/ru-ru/cpp/c-operators>. – Дата доступа: 18.02.2024
5. Функции Python [Электронный ресурс]. – Режим доступа: https://metanit.com/python/tutorial/2.8.php. – Дата доступа: 18.02.2024.
6. Классы Python [Электронный ресурс]. – Режим доступа: https://metanit.com/python/tutorial/7.1.php. – Дата доступа: 18.02.2024.

## ПРИЛОЖЕНИЕ А

## (обязательное)

## Листинг исходного кода

Листинг 1 – Программная код функции классификации

using System;

using System.Globalization;

using System.Text.RegularExpressions;

public static class Program

{

static Dictionary<int, string> operators = new Dictionary<int, string>()

{

{1, "+"},

{2, "-"},

{3, "/"},

{4, "\*"},

{5, "\*\*"},

{6, "%"},

{7, ">"},

{8, "<"},

{9, "="},

{10, "=="},

{11, ">="},

{12, "<="},

{13, "!="},

{14, "-="},

{15, "\*="},

{16, "/="},

{17, "%="},

{18, "+="},

{19, "("},

{20, ")"}

};

static Dictionary<int, string> specialWords = new Dictionary<int, string>()

{

{1, "while"},

{2, "if"},

{3, "elif"},

{4, "else"},

{5, "break"},

{6, "continue"},

{7, "for"},

{8, "range"},

{9, "def"},

{10, "lambda"},

{11, "return"}

};

static List<string> variables = new List<string>();

static List<string> functions = new List<string>();

public static string path = @"C:\Users\37529\Desktop\Лекции\МТран\Labs\Lab2\Lab2\script.txt";

static void PrintTable(string lexeme, string type)

{

Console.WriteLine($"{lexeme,-20}\t|\t{type}");

}

static string GetDictionaryKey(string value, Dictionary<int, string> dict)

{

foreach (var item in dict)

{

if (Math.Abs(item.Value.Length - value.Length) > 2)

{

continue;

}

int count = item.Value.Length > value.Length ? item.Value.Length : value.Length;

int len = count;

for (int i = 0; i < len; i++)

{

if (i < item.Value.Length && i < value.Length && item.Value[i] == value[i])

{

count--;

}

}

if (count == 0)

{

return item.Key.ToString();

}

else if (count < len - 1)

{

return value + "? may be " + item.Value;

}

}

return "-1";

}

static string IsBool(string value)

{

if (value == "True" || value == "False")

{

return value + "\tconst bool";

}

return "not bool";

}

static string IsString(string value)

{

if (value.IndexOf('"') == value.LastIndexOf('"'))

{

return "not string";

}

return value + "\tconst string";

}

static string GetNumberType(string num)

{

num = num.Replace(',', '.');

if (int.TryParse(num, out \_))

{

return "const int";

}

if (float.TryParse(num, NumberStyles.Float, CultureInfo.InvariantCulture, out \_))

{

return "const float";

}

return "not num";

}

static void GetFunctionName(string line, string lexeme)

{

int startIndex = line.IndexOf(lexeme);

if (startIndex != -1)

{

int openParenIndex = line.IndexOf('(', startIndex);

if (openParenIndex != -1)

{

// Extract function name from the substring between lexeme and (

string functionName = line.Substring(startIndex, openParenIndex - startIndex).Trim();

if (!string.IsNullOrEmpty(functionName))

{

functions.Add(functionName);

}

}

else

{

// If no open parenthesis is found, it might still be a function without parameters

functions.Add(lexeme);

}

}

}

static string IsFunc(string name)

{

foreach (var item in functions)

{

if (name.StartsWith(item + "("))

{

return item + "\t function name";

}

}

return "not function";

}

static string GetLexemeType(string lexeme)

{

if (lexeme.IndexOf(',') != -1)

{

lexeme = lexeme.Remove(lexeme.IndexOf(','));

}

if (lexeme.IndexOf(':') != -1)

{

lexeme = lexeme.Remove(lexeme.IndexOf(':'));

}

if (lexeme == "")

{

return "";

}

string key = GetDictionaryKey(lexeme, operators);

if (key != "-1" && int.TryParse(key, out var number))

{

string result = lexeme + " " + " operator";

return result;

}

else if (key != "-1")

{

return key;

}

key = GetDictionaryKey(lexeme, specialWords);

if (key != "-1" && int.TryParse(key, out number))

{

string result = lexeme + "\t" + " special word";

return result;

}

else if (key != "-1")

{

return key;

}

string numberType = GetNumberType(lexeme);

if (numberType != "not num")

{

string result = lexeme + "\t" + numberType;

return result;

}

string isbool = IsBool(lexeme);

if (isbool != "not bool")

{

return isbool;

}

string isString = IsString(lexeme);

if (isString != "not string")

{

return isString;

}

string isFunc = IsFunc(lexeme);

if (isFunc != "not function")

{

return isFunc;

}

variables.Add(lexeme);

return lexeme + "\tvariable";

}

public static async Task Main(string[] args)

{

Console.WriteLine("Lexeme\t\t | Type");

Console.WriteLine("------------------------|-------------------------------");

try

{

using (StreamReader reader = new StreamReader(path))

{

string? line;

while ((line = await reader.ReadLineAsync()) != null)

{

string[] lexemes = line.Split(' ', StringSplitOptions.RemoveEmptyEntries);

foreach (string lexeme in lexemes)

{

if (string.IsNullOrWhiteSpace(lexeme))

continue; // Пропускаем пустые лексемы

GetFunctionName(line, lexeme);

if (lexeme == "def")

{

GetFunctionName(line, lexeme);

}

//Console.WriteLine(GetLexemeType(lexeme));

string lexemeType = GetLexemeType(lexeme);

PrintTable(lexeme, lexemeType);

}

}

}

}

catch (FileNotFoundException ex)

{

Console.WriteLine($"File not found: {ex.Message}");

}

catch (Exception ex)

{

Console.WriteLine($"Lexical error: {ex.Message}");

Console.WriteLine($"StackTrace: {ex.StackTrace}");

}

}

}