Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИНФОРМАТИКИ

* РАДИОЭЛЕКТРОНИКИ КАФЕДРА ИНФОРМАТИКИ

Отчёт по лабораторной работе №2

По теме “Лексический анализатор”

Выполнил:

студент гр. 053503

Буткевич Г. О.

Проверил:

Гриценко Н. Ю.

Минск 2023

СОДЕРЖАНИЕ

1 Цель работы3

2 Созданный лексический анализ4

3 Демонстрация5

4 Выводы9

Приложение А (обязательное) Код программы лексического анализатора10

**1 ЦЕЛЬ РАБОТЫ**

Разработка лексического анализатора подмножества языка программирования, определённого в лабораторной работе 1. Программа анализа определяет лексические правила и выполняет перевод потока символов программ лабораторной работы 1 в поток лексем (токенов).

На вход программы подается текстовый файл, содержащий строки текста программы.

Например, строка присваивания переменной значения арифметического выражения в виде

ПЕРЕМЕННАЯ = ВЫРАЖЕНИЕ.

Выражение может включать:

* Знаки сложения и умножения («+» и «\*»);
* Круглые скобки («(» и «)»);
* Константы (например, 5; 3.8; 1e+18, 8.41E–10);
* Имена переменных.

Имя переменной – это последовательность букв и цифр, начинающаяся с буквы.

Разбор выражения COST = (PRICE+TAX) \*0.98.

Проанализируем выражение:

COST, PRICE и TAX – лексемы-идентификаторы;

0.98 – лексема-константа;

− =, +, \* – просто лексемы.

Пусть все константы и идентификаторы можно отображать в лексемы типа <идентификатор> (<ИД>). Тогда выходом лексического анализатора будет последовательность лексем <ИД1> = (<ИД2> + <ИД3>) \* <ИД4>.

Вторая часть компоненты лексемы (указатель, т.е. номер лексемы в таблице имен) – показана в виде индексов. Символы «=», «+» и «\*» трактуются как лексемы, тип которых представляется ими самими. Они не имеют связанных с ними данных и, следовательно, не имеют указателей.

После того, как в результате лексического анализа лексемы распознаны, информация о некоторых из них собирается и записывается в таблицу имён.

Нахождение и локализация лексических ошибок программы: не менее 4-х ошибок. Локализация - № строки, № позиции в строке.

**2 СОЗДАННЫЙ ЛЕКСИЧЕСКИЙ АНАЛИЗАТОР**

Код исследуемой программы в файле подается на вход программы-анализатора и считывается построчно. С помощью строки регулярных выра-жений в каждой строке выделяются такие типы токенов, как строковые, символьные и целочисленные константы, символы-операторы, разделители и ключевые слова.

Таким образом ни один символ входного файла не будет потерян из-за не-соответствия какой-то части заданных правил регулярных выражений.

**3 ДЕМОНСТРАЦИЯ**

Код программы для анализа (c намеренно добавленными ошибками):

fun quicksort(items:List<Int>):List<Int>{

iff (items.count()) < 2 { // iff вместо if

return items

}

val 5ivot = items[items.count()/2] //5ivot вместо pivot

val equal = items.filter { it =!= pivot } // неправильный оператор !=!

val less = items.filter { it <+ pivot } //неправильный оператор <+

val greater = items.filter { it > pivot }

return quicksort(less) + equal + quicksort(greater)

}

fun main(args: Array<String>) {

println(“Original list:”)

val numbers = listOf<Int>(2, 4, 7, 3, 6, 9, 5, 1, 0)

println(numbers)

println(“Ordered list:”)

val ordered = quicksort(numbers)

println(ordered)

}

На рисунке 1 показаны ошибки, которые были намеренно добавлены в код. В первой строке показана ошибка при написании оператора if, а именно iff вместо if. Во второй строке показана ошибка при попытке создания переменной, название которой начинается на цифру, а именно 5ivot. Третья строка с ошибкой появилась из-за того, что переменная с неправильным названием не была добавлена в список переменных анализатора. В четвертой строке показана ошибка с неправильным оператором =!=. С пятой по седьмую строки показаны ошибки в связи с неправильным объявлением переменной pivot, а именно 5ivot вместо pivot.

При исправлении ошибок будет показан вывод с несколькими таблицами. На рисунке 2 показана таблица с ключевыми словами, найденными в коде. На рисунке 3 показана таблица с найденными операторами. На рисунке 4 – таблица с найденными разделителями. На рисунке 5 – таблица с целочисленными константами. На рисунке 6 – строковые константы. На рисунке 7 – символьные константы. На рисунке 8 – найденные в коде переменные.

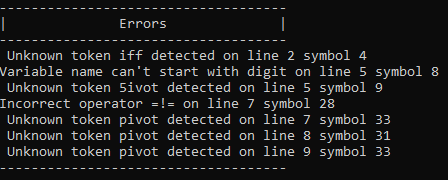


Рисунок 1 – Таблица со всеми ошибками

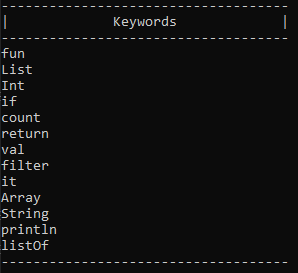


Рисунок 2 – Таблица с найденными ключевыми словами

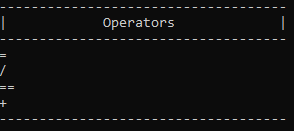


Рисунок 3 – Таблица с найденными операторами

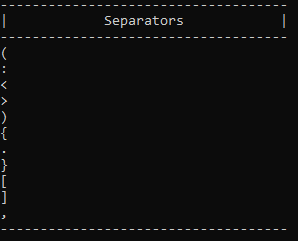


Таблица 4 – Таблица с найденными разделителями

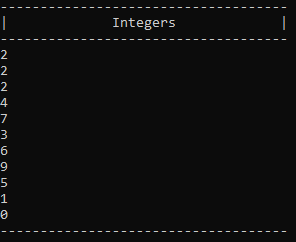


Рисунок 5 – Таблица с найденными целочисленными константами

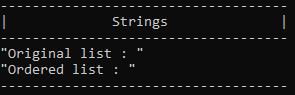


Рисунок 6 – Таблица с найденными строковыми константами

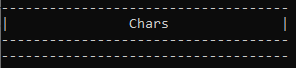


Рисунок 7 – Таблица с найденными символьными константами (в данном коде они отсутствуют)

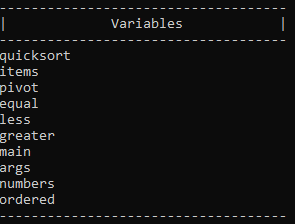


Рисунок 8 – Таблица с переменными (в том числе названиями функций)

**4 ВЫВОДЫ**

Был разработан лексический анализатор подмножества языка Kotlin. По-лучены знания о принципах работы, обязанностях и особенностях построения лексических анализаторов языков программирования, описанных в отчёте лабораторной работы.

**ПРИЛОЖЕНИЕ А**

**(обязательное)**

**Текст программы лексического анализатора**

using System.Text;

namespace lab2

{

class Program

{

private static String[] separators = { ";", "{", "}", ">", "<", "|", "&", "~", ":", ".", "#", "##", ",", "(", ")", "[", "]", "()", "[]" };

private static String[] operators = { "&&", "||", "++", "--", "==", "<=", ">=", "!=", "\*", "/", "%", "=", "+=", "\*=", "/=", "-=", "+", "-" };

private static String[] keywords = { "print", "var", "val", "while", "if", "return", "count", "filter", "println", "listOf", "fun", "Int", "Array", "String", "List", "it" };

private static String[] words;

private static List<String> variables = new();

private static List<String> storedSeparators = new();

private static List<String> storedOperators = new();

private static List<String> storedKeywords = new();

private static List<String> storedIntegers = new();

private static List<String> storedStrings = new();

private static List<String> storedChars = new();

private static List<String> storedErrors = new();

private static int lineNum;

private static int charNum;

static void Main(string[] args)

{

String data = File.ReadAllText("file.txt") + "\n$~";

for (int j = 0; j < operators.Length; j++)

data = data.Replace(operators[j], " " + operators[j] + " ");

for (int i = 0; i < separators.Length; i++)

data = data.Replace(separators[i], " " + separators[i] + " ");

var newData = new StringBuilder(data);

for (int i = 0; i < newData.Length; i++)

{

if (newData[i] == '\n')

newData[i] = '#';

}

data = newData.ToString();

data = data.Replace("\n", String.Empty);

data = data.Replace("\r", String.Empty);

data = data.Replace("\t", String.Empty);

data = data.Replace("$", String.Empty);

data = data.Replace("#", "\n");

data = data.Replace("~", String.Empty);

words = data.Split(' ');

for (int i = 0; i < words.Length; i++)

{

if (words[i] != "" && words[i][0] == '\"')

{

for (int j = 1; ; j++)

{

string temp = words[i + j];

if (temp == "")

{

continue;

}

words[i] = words[i] + " " + words[i + j];

words[i + j] = "";

if (temp == "\"" || temp[temp.Length - 1] == '\"')

{

break;

}

}

}

if ((words[i] == "=" && words[i + 2] == "=") || (words[i] == "+" && words[i + 2] == "+") || (words[i] == "-" && words[i + 2] == "-") || (words[i] == "\*" && words[i + 2] == "=") || (words[i] == "/" && words[i + 2] == "=") || (words[i] == "+" && words[i + 2] == "=") || (words[i] == "-" && words[i + 2] == "=") || (words[i] == "&" && words[i + 2] == "&") || (words[i] == "|" && words[i + 2] == "|") || (words[i] == "(" && words[i + 2] == ")") || (words[i] == "[" && words[i + 2] == "]") || (words[i] == "-" && words[i + 2] == "-") || (words[i] == "<" && words[i + 2] == "=") || (words[i] == ">" && words[i + 2] == "="))

{

words[i] += words[i + 2];

words[i + 2] = String.Empty;

}

else if (words[i] == "!" && words[i + 1] == "=")

{

words[i] += words[i + 1];

words[i + 1] = String.Empty;

}

}

for (int i = 0; i < words.Length; i++)

{

charNum += words[i].Length;

if (words[i].Length > 0 && words[i][0] == '\n')

{

int j = 0;

while (words[i].Length > j && words[i][j] == '\n')

{

lineNum++;

charNum = 0;

j++;

}

string replaceString = "\n";

for (int k = 1; k > j; k++)

{

replaceString += "\n";

}

words[i] = words[i].Replace(replaceString, "");

}

if (words[i] == "var" || words[i] == "val")

{

if (!variables.Contains(words[i + 1]))

{

if (Char.IsDigit(words[i + 1][0]))

{

storedErrors.Add("Variable name can't start with digit on line " + (lineNum + 1) + " symbol " + (charNum + 5));

}

else

{

variables.Add(words[i + 1]);

}

}

else

{

storedErrors.Add("Trying to declare existing variable " + words[i + 1] + " on line " + (lineNum + 1) + " symbol " + (charNum + 5));

}

}

if (words[i] == "fun")

{

if (!variables.Contains(words[i + 1]))

{

variables.Add(words[i + 1]);

if (words[i + 2] == "(" && words[i + 3] != ")" && words[i + 4] == ":")

{

if (!variables.Contains(words[i + 3]))

{

variables.Add(words[i + 3]);

}

}

}

else

{

storedErrors.Add("Trying to declare existing function " + words[i + 1] + " on line " + (lineNum + 1) + " symbol " + (charNum + 5));

}

}

int u = 1;

while (i - u > 0 && words[i - u] == "")

{

u++;

}

if (operators.Contains(words[i]) && operators.Contains(words[i - u]))

{

storedErrors.Add("Incorrect operator " + words[i - u] + words[i] + " on line " + (lineNum + 1) + " symbol " + (charNum + 1));

}

if (words[i] == "if" && words[i + 2] != "(")

{

storedErrors.Add("Incorrect if statement on line " + (lineNum + 1) + " symbol " + (charNum + 5));

}

if (words[i] == "return" && !(variables.Contains(words[i + 1]) || Int32.TryParse(words[i + 1], out int temp)))

{

storedErrors.Add("Incorrect return statement on line " + (lineNum + 1) + " symbol " + (charNum + 1));

}

CheckLexicalAnalyzer(words[i]);

}

if (storedErrors.Count > 0)

{

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

Console.WriteLine("| Errors |");

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

for (int i = 0; i < storedErrors.Count; i++)

{

Console.WriteLine(storedErrors[i]);

}

Console.WriteLine("------------------------------------\n");

}

else

{

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

Console.WriteLine("| Keywords |");

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

for (int i = 0; i < storedKeywords.Count; i++)

{

Console.WriteLine(storedKeywords[i]);

}

Console.WriteLine("------------------------------------\n");

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

Console.WriteLine("| Operators |");

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

for (int i = 0; i < storedOperators.Count; i++)

{

Console.WriteLine(storedOperators[i]);

}

Console.WriteLine("------------------------------------\n");

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

Console.WriteLine("| Separators |");

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

for (int i = 0; i < storedSeparators.Count; i++)

{

Console.WriteLine(storedSeparators[i]);

}

Console.WriteLine("------------------------------------\n");

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

Console.WriteLine("| Integers |");

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

for (int i = 0; i < storedIntegers.Count; i++)

{

Console.WriteLine(storedIntegers[i]);

}

Console.WriteLine("------------------------------------\n");

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

Console.WriteLine("| Strings |");

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

for (int i = 0; i < storedStrings.Count; i++)

{

Console.WriteLine(storedStrings[i]);

}

Console.WriteLine("------------------------------------\n");

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

Console.WriteLine("| Chars |");

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

for (int i = 0; i < storedChars.Count; i++)

{

Console.WriteLine(storedChars[i]);

}

Console.WriteLine("------------------------------------\n");

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

Console.WriteLine("| Variables |");

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

for (int i = 0; i < variables.Count; i++)

{

Console.WriteLine(variables[i]);

}

Console.WriteLine("------------------------------------\n");

}

}

private static String Parse(String item)

{

StringBuilder str = new StringBuilder();

if (CheckOperators(item) == true && !storedOperators.Contains(item))

storedOperators.Add(item);

else if (CheckSeparator(item) == true && item != "#" && !storedSeparators.Contains(item))

storedSeparators.Add(item);

else if (CheckKeywords(item) == true && !storedKeywords.Contains(item))

storedKeywords.Add(item);

else if (item.Equals("\r") || item.Equals("\n") || item.Equals("\r\n"))

str.Append(item);

return str.ToString();

}

private static bool CheckSeparator(String str) => separators.Contains(str);

private static bool CheckOperators(String str) => operators.Contains(str);

private static bool CheckKeywords(String str) => keywords.Contains(str);

private static void CheckLexicalAnalyzer(String str)

{

StringBuilder token = new StringBuilder();

bool isCheck = false;

if (CheckOperators(str.ToString()))

{

Parse(str.ToString());

return;

}

for (int i = 0; i < str.Length; i++)

{

try

{

int intValue;

if (Int32.TryParse(str, out intValue) && !isCheck)

{

storedIntegers.Add(str);

//Console.WriteLine(" (integerValue, <" + str + ">) ");

isCheck = true;

}

else if (str.Equals("\r") || str.Equals("\n") || str.Equals("\r\n")) { }

else if (CheckOperators(str[i].ToString()))

{

if (CheckOperators(str.ToString()))

{

Parse(str.ToString());

}

else

{

Parse(str[i].ToString());

}

}

else if (CheckSeparator(str[i].ToString()))

//Console.WriteLine(Parse(str[i].ToString()));

Parse(str[i].ToString());

else if (str.Contains("\""))

{

if (str[i + 1].ToString() != "\"")

//Console.WriteLine();

do { i++; } while (str == "\"");

if (i == 1)

//Console.WriteLine(" (String, <" + str + ">) ");

storedStrings.Add(str);

}

else if (str.Contains("\'"))

{

if (str[i + 1].ToString() != "\'")

//Console.WriteLine();

do { i++; } while (str == "\'");

if (i == 1)

//Console.WriteLine(" (Char, <" + str + ">) ");

storedChars.Add(str);

}

else

{

token.Append(str);

try

{

if (keywords.Contains(token.ToString()))

//Console.WriteLine(Parse(token.ToString()));

Parse(token.ToString());

if (variables.Contains(token.ToString()))

//Console.WriteLine(Parse(token.ToString()));

Parse(token.ToString());

else

{

int intValu;

if (!separators.Contains(str[i].ToString()))

if (!operators.Contains(str[i].ToString()) || !operators.Contains(str.ToString()))

if (!keywords.Contains(str.ToString()))

if (!variables.Contains(str.ToString()))

if (!str.Contains("\"") || !str.Contains("\'"))

if (!Int32.TryParse(str[i].ToString(), out intValu) && !isCheck)

if (!str.Equals("\r") || !str.Equals("\n") || !str.Equals('\r') || !str.Equals('\n') || !str.Equals("\r\n") || !str.Equals("#"))

{

storedErrors.Add(" Unknown token " + str + " detected on line " + (lineNum + 1) + " symbol " + (charNum + 1));

isCheck = true;

}

}

}

catch (Exception) { }

token.Remove(i, i);

}

}

catch (Exception) { }

}

}

}

}