

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

**Український державний університет науки і технологій**

Кафедра «Комп’ютерні інформаційні технології»

**Лабораторна робота №3**

**з дисципліни «Системи трансляцій»**

**на тему: «****Рефакторинг лексичного аналізатора.»**

Виконав:

Студент гр. ПЗ2011

Кулик С.В.

Прийняв:

Разносілін В.В.

Дніпро, 2023

**Тема:** Рефакторинг лексичного аналізатора.

**Мета:**

* Виконати рефакторинг лексичного аналізатора для покращення читабельності, ефективності та якості коду.
* Забезпечити модульність та легкість розширення лексичного аналізатора.

**Опис рефакторингу лексичного аналізатора**

Рефакторинг лексичного аналізатора спрямований на поліпшення структури, якості та читабельності коду, а також збільшення його підтримуваності та розширюваності. Інінціалізація двох порожніх строк ,змінна inQuotes зі значенням false, що вказує на те, що зараз ми не знаходимося в межах лапок

Функція проходиться по кожному символу c в рядку src за допомогою циклу foreach. В залежності від поточного стану inQuotes та типу символу c виконуються певні дії.

Якщо змінна inQuotes має значення true, це означає, що ми знаходимося всередині лапок. Тому поточний символ c додається до рядка token\_src. Якщо символ c дорівнює подвійній лапці \", це означає, що закінчився рядок у лапках. У такому випадку поточний токен token\_src додається до результату токенізації, рядок token\_src очищується, а змінна inQuotes змінюється на протилежне значення (тобто false).

**Оновлений код токенайзера**

using System;

using System.Collections.Generic;

using System.Globalization;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace ST1

{

internal class Tockenizer

{

public const int TOK\_UNKNOW = 0;

public const int TOK\_DIV = 100;

public const int TOK\_OP = 200;

public const int TOK\_CONST = 300;

public const int TOK\_ID = 1000;

public const int TOK\_EOF = Int32.MaxValue;

public const int TOK\_LPR = TOK\_DIV + 1;

public const int TOK\_RPR = TOK\_DIV + 2;

public const int OP\_CMP\_EQ = TOK\_OP + 1;

public const int OP\_CMP\_NE = TOK\_OP + 2;

public const int OP\_CMP\_LS = TOK\_OP + 3;

public const int OP\_CMP\_GR = TOK\_OP + 4;

public const int OP\_CMP\_LE = TOK\_OP + 5;

public const int OP\_CMP\_GE = TOK\_OP + 6;

public const int OP\_LOG\_AND = TOK\_OP + 7;

public const int OP\_LOG\_OR = TOK\_OP + 8;

public const int OP\_LOG\_NOT = TOK\_OP + 9;

public const int CONST\_INT = TOK\_CONST + 1;

public const int CONST\_FLT = TOK\_CONST + 2;

public const int CONST\_STR = TOK\_CONST + 3;

public const int CONST\_DT = TOK\_CONST + 4;

public const int CONST\_BOOL = TOK\_CONST + 5;

private List<Token> tokens;

public Dictionary<int, string> dict = new Dictionary<int, string>();

public struct Token

{

public int type;

public int subtype;

public string data;

}

public void fill\_dict()

{

dict[TOK\_UNKNOW] = "TOK\_UNKNOWN";

dict[TOK\_DIV] = "TOK\_DIV ";

dict[TOK\_OP] = "TOK\_OP\t";

dict[TOK\_CONST] = "TOK\_CONST";

dict[TOK\_ID] = "TOK\_ID\t";

dict[TOK\_EOF] = "TOK\_EOF ";

dict[TOK\_LPR] = "TOK\_LPR\t";

dict[TOK\_RPR] = "TOK\_RPR\t";

dict[OP\_CMP\_EQ] = "OP\_CMP\_EQ";

dict[OP\_CMP\_NE] = "OP\_CMP\_NE";

dict[OP\_CMP\_LS] = "OP\_CMP\_LS";

dict[OP\_CMP\_GR] = "OP\_CMP\_GR";

dict[OP\_CMP\_LE] = "OP\_CMP\_LE";

dict[OP\_CMP\_GE] = "OP\_CMP\_GE";

dict[OP\_LOG\_AND] = "OP\_LOG\_AND";

dict[OP\_LOG\_OR] = "OP\_LOG\_OR";

dict[OP\_LOG\_NOT] = "OP\_LOG\_NOT";

dict[CONST\_INT] = "CONST\_INT ";

dict[CONST\_FLT] = "CONST\_FLT ";

dict[CONST\_STR] = "CONST\_STR ";

dict[CONST\_DT] = "CONST\_DT ";

dict[CONST\_BOOL] = "CONST\_BOOL";

}

public Tockenizer(string src)

{

tokens = new List<Token>();

fill\_dict();

string cur = "";

string cur\_op = "";

bool inQuotes = false;

foreach (char c in src)

{

switch (c)

{

case ' ':

case '\n':

case '\t':

case ';':

if (!cur.Equals(""))

if (!inQuotes)

{

if (!cur\_op.Equals(""))

{

addToken(cur\_op);

cur\_op = "";

}

addToken(cur);

cur = "";

}

else cur += c;

break;

case '>':

case '<':

case '=':

if (!inQuotes)

{

if (!cur.Equals(""))

{

addToken(cur);

cur = "";

}

if (!cur\_op.Equals(""))

{

cur\_op += c;

addToken(cur\_op);

cur\_op = "";

} else

{

cur\_op += c;

}

}

else cur += c;

break;

case '(':

case ')':

if (!inQuotes)

{

if (!cur\_op.Equals(""))

{

addToken(cur\_op);

cur\_op = "";

}

if (!cur.Equals(""))

addToken(cur);

cur = "";

cur += c;

addToken(cur);

cur = "";

}

else cur += c;

break;

case '\"':

if (inQuotes)

{

addStringToken(cur);

cur = "";

}

inQuotes = !inQuotes;

break;

default:

cur += c;

break;

}

}

if (!cur.Equals("")) addToken(cur);

tokens.Add(GetToken(TOK\_EOF, TOK\_EOF, "EOF"));

}

private void addToken(string src)

{

Token token;

if (isDiv(src, out token) || isOp(src, out token) || isConst(src, out token) || isId(src, out token))

tokens.Add(token);

}

private void addStringToken(string src)

{

tokens.Add(GetToken(TOK\_CONST, CONST\_STR, src));

}

public List<Token> GetTokens()

{

return tokens;

}

public bool isDiv(string src, out Token token)

{

if (src[0] == '(') token = GetToken(TOK\_DIV, TOK\_LPR, src);

else if (src[0] == ')') token = GetToken(TOK\_DIV, TOK\_RPR, src);

else { token = GetEmptyToken(); return false; }

return true;

}

public bool isOp(string src, out Token token)

{

if (src.Equals("==")) token = GetToken(TOK\_OP, OP\_CMP\_EQ, src);

else

if (src.Equals("!=")) token = GetToken(TOK\_OP, OP\_CMP\_NE, src);

else

if (src.Equals("<")) token = GetToken(TOK\_OP, OP\_CMP\_LS, src);

else

if (src.Equals(">")) token = GetToken(TOK\_OP, OP\_CMP\_GR, src);

else

if (src.Equals("<=")) token = GetToken(TOK\_OP, OP\_CMP\_LE, src);

else

if (src.Equals(">=")) token = GetToken(TOK\_OP, OP\_CMP\_GE, src);

else

if (src.Equals("and")) token = GetToken(TOK\_OP, OP\_LOG\_AND, src);

else

if (src.Equals("or")) token = GetToken(TOK\_OP, OP\_LOG\_OR, src);

else

if (src.Equals("not")) token = GetToken(TOK\_OP, OP\_LOG\_NOT, src);

else { token = GetEmptyToken(); return false; }

return true;

}

public bool isId(string src, out Token token)

{

token = GetEmptyToken();

if (src == "" || !Char.IsLetter(src[0])) return false;

foreach (char ch in src)

if (!Char.IsLetter(ch) && !Char.IsDigit(ch) && ch != '.') return false;

token = GetToken(TOK\_ID, TOK\_ID, src);

return true;

}

public bool isConst(string src, out Token token)

{

token = GetEmptyToken();

return isInt(src, out token) || isFloat(src, out token) || isBool(src, out token);

}

public bool isInt(string src, out Token token)

{

int nint = 0;

if (Int32.TryParse(src, out nint)) token = GetToken(TOK\_CONST, CONST\_INT, src);

else { token = GetEmptyToken(); return false; }

return true;

}

public bool isBool(string src, out Token token)

{

if (bool.TryParse(src, out \_)) token = GetToken(TOK\_CONST, CONST\_BOOL, src);

else { token = GetEmptyToken(); return false; }

return true;

}

public bool isFloat(string src, out Token token)

{

NumberFormatInfo nfi = new NumberFormatInfo();

nfi.NumberDecimalSeparator = ",";

if (decimal.TryParse(src, NumberStyles.Any, nfi, out decimal dec)) token = GetToken(TOK\_CONST, CONST\_FLT, src);

else { token = GetEmptyToken(); return false; }

return true;

}

public Token GetToken(int type, int subtype, string src)

{

Token token = new Token();

token.type = type;

token.subtype = subtype;

token.data = src;

return token;

}

public Token GetEmptyToken()

{

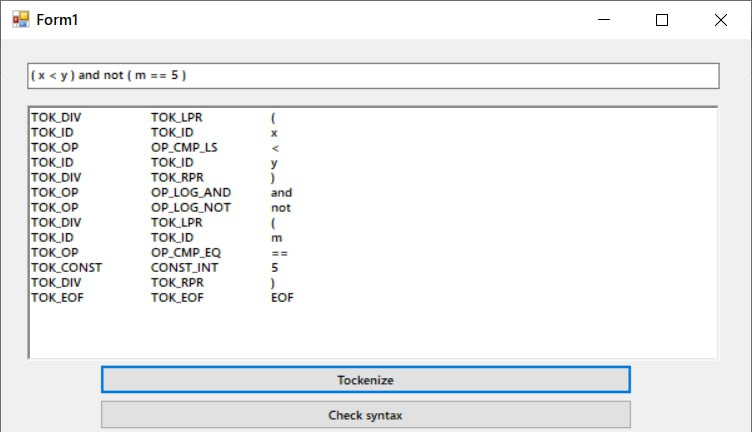
return GetToken(TOK\_UNKNOW, 0, "");

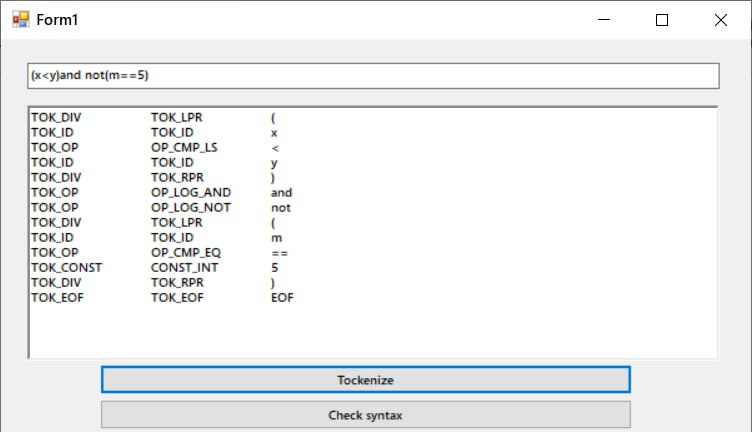
}

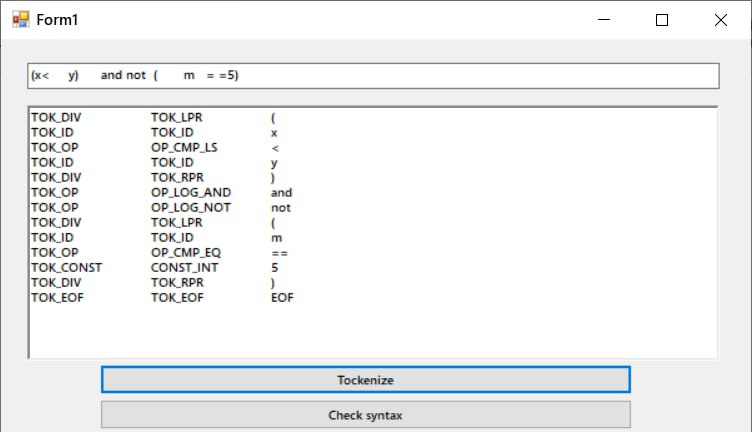
}

}

**Результати роботи програми**







**Висновки**

Рефакторинг лексичного аналізатора спрямований на поліпшення структури та якості коду, збільшення його читабельності та підтримуваності, а також можливості розширення.

Основні кроки рефакторингу лексичного аналізатора можуть включати наступне:

Розбиття на функції: Розгляньте лексичний аналізатор і виділіть окремі функції для різних етапів обробки (наприклад, отримання наступного символу, розпізнавання лексеми, визначення токенів тощо). Це допоможе зробити код більш структурованим та зрозумілим.

Використання констант та перелічень: Замініть магічні числа та рядки на іменовані константи або перелічення. Це зробить код більш зрозумілим та забезпечить його підтримуваність у разі змін.

Застосування адекватних імен змінних: Перегляньте імена змінних та функцій у вашому лексичному аналізаторі і переконайтеся, що вони є зрозумілими та відображають їхню призначеність. Використовуйте осмислені назви, щоб полегшити читання та розуміння коду.

Усунення дублювання коду: Виявіть повторюваний код у вашому лексичному аналізаторі та перенесіть його до окремих функцій або класів. Це зменшить розмір програми, полегшить підтримку та знизить ризик виникнення помилок.

Використання структур даних: Розгляньте використання підходящих структур даних для зберігання лексем, токенів та інших важливих даних.