

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

**Український державний університет науки і технологій**

Кафедра «Комп’ютерні інформаційні технології»

**Лабораторна робота №1**

**з дисципліни «Системи трансляцій»**

**на тему: «Розробка специфікацій компілюючої системи.Проектування лексичного аналізатору.»**

Виконав:

Студент гр. ПЗ2011

Кулик С.В.

Прийняв:

Разносілін В.В.

Дніпро, 2023

**Тема:** Розробка специфікацій компілюючої системи.Проектування лексичного аналізатору.

**Мета роботи**

* Набути навичок проектування синтаксису и семантики алгоритмічних мов.
* Вивчення закономірностей та особливостей побудови лексичного аналізатору.
* Набуття досвіду розробки алгоритмічно складних програм.

1. **Робота програми**

На початку роботи програми, при запуску користувач має ввести рядок, що розбивається на складові – токени у вигляд структури. В цій структурі є поля які допомагають визначити вид токену, його підтип, місце у вхідному реченні та власне значення. Введений рядок аналізується на правильну послідовність також перевіряється чи правильно записано всі операції

Якщо перевірка проходить успішно, в програмі формується абстрактне синтаксичне дерево, на основі якого, далі при роботі програми будуються операції в постфіксній формі, послідовно далі і створюється DLL код

1. **Синтаксичний аналіз**

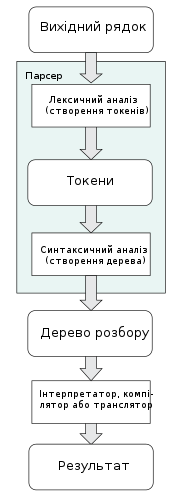
Синтаксичний аналізатор — це програмний компонент, який приймає вхідні дані (часто текст) і створює структуру даних — часто дерево розбору, абстрактне деерво синтаксису або іншу ієрархічну структуру — забезпечує структурне представлення вводу, перевіряє правильність синтаксису в процесі. Аналізу можуть передувати або слідувати інші кроки, або їх можна об'єднати в один крок. Аналізатору часто передує окремий лексичний аналізатор, який створює токени з послідовності введених символів; Крім того, їх можна об'єднати у парсінг без скасування. Аналізатори можуть бути запрограмовані вручну або автоматично, або напів автоматично генератором парсерів. Розбір допомагає шаблону, який виробляє відформатований вихід. Вони можуть використовуватись у різних ділянках, але часто з'являються разом, наприклад, пара scanf/ptintf, або як вхідний (аналіз вхідних даних) та вихідний етапи (створення кінцевого коду) компілятора.

Вхідними даними для синтаксичного аналізатора часто є текст деякою комп’ютерною мовою, але також може бути текстом природною мовою або менш структурованими текстовими даними, в цьому випадку, як правило, витягуються лише окремі частини тексту, а не дерево розбору. Параметри відрізняються від дуже простих функцій, таких як scanf, до складних програм, таких як інтерфейс компілятора C++ або HTML -аналізатор веббраузера. Важливий клас простий синтаксичний аналіз виконується за допомогою регулярних виразів, в яких група регулярних виразів визначає регулярну мову та двигун регулярного виразу, автоматично генеруючи аналізатор для цієї мови, що дозволяє узгодити шаблон та вилучення тексту. В інших контекстах регулярні вирази замість цього використовуються перед розбором, як етап лексизації, вихід якого потім використовується аналізатором.

Використання аналізаторів залежить від вхідних даних. У випадку з мовами даних часто використовується синтаксичний аналізатор як функція читання файлів у програмі, наприклад, читання в HTML або XML -тексті; ці приклади є мовами розмітки даних. У випадку мов пррограмування є компонентом компілятора або інтерпретатора, який аналізує початковий код мови комп’ютерного програмування для створення певної форми внутрішнього представлення; аналізатор є ключовим кроком в інтерфейсі компілятора. Мови програмування, як правило, вказуються в термінах детермінистичної контексто-вільної граматики, оскільки для них можуть бути написані швидкі та ефективні аналізатори. Для компіляторів сам аналіз може бути виконаний за один прохід або кілька проходів — див. Одно-прохідний компілятор і багатопрохідний матеріал.

**Огляд процесу**

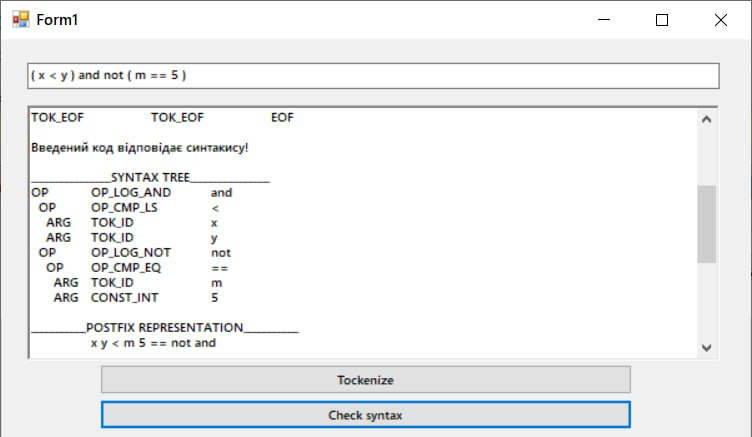
Схема роботи синтаксичного аналізатора



Завдання аналізатора по суті полягає в тому, щоб визначити, як вхід можна отримати з початкового символу граматики. Це можна зробити по суті двома способами:

* Низхідний синтаксичний аналіз — аналіз згори донизу можна розглядати як спробу знайти найбільший початок z слова x, що може бути початком слова, що виводиться, шляхом пошуку в синтаксичному дереві за допомогою розгорнення згори донизу заданих формальних правил граматики. Токени читаються зліва направо. Інклюзивний вибір використовується для задоволення багатозначності шляхом розширення всіх альтернативних правих правил граматики.
* Синтаксичний аналіз знизу — Синтаксичний аналізатор може початися з введення та спробувати переписати символ на його початку. Інтуїтивно, синтаксичний аналізатор намагається знайти найдовші основні елементи, потім елементи, що містять їх, і так далі. Аналізатор LR є прикладами аналізаторів знизу вгору. Іншим терміном, що використовують для цього типу аналізатора, є Shift-Reduce, синтаксичний аналіз.

**Синтаксичне дерево завдання**



1. **Лексичний аналізатор**

Лексичний розбір — це процес перетворення послідовності символів в послідовність токенів (груп символів що відповідають певним шаблонам), та визначення їх типів. Програма, чи функція що виконує лексичний аналіз, називається лексичним аналізатором, токенізатором чи сканером. Часто сканер є звичайною функцією що використовується парсером (синтаксичним аналізатором), для отримання наступного токена з потоку вхідних символів в процесі компіляції. Також часто лексичний аналіз використовують для підсвітки синтаксису певних мов.

Лексичний аналізатор формує першу фазу аналізу коду, що відбувається під час початкової стадії компіляції. Аналіз зазвичай відбувається за один прохід.

Лексичні аналізатори та парсери найчастіше використовуються для компіляторів, але можуть використовуватися для інших мовних засобів, таких як лінтерів. Лексичний аналіз можна розділити на два етапи: сканування, яке сегментує вхідний рядок у синтаксичні одиниці, які називаються лексемами, та класифікує їх у класи токенів; та обчислення, яке перетворює лексеми в оброблені значення.

Лексичні аналізатори, як правило, досить прості, причому більша частина складності відкладена на фази парсера або семантичного аналізу, та можуть бути згенеровані генераторами лексичних аналізаторів, особливо за допомогою lex, або похідних. Однак, лексичні аналізатори можуть іноді мати деяку складність, таку як обробка структури фрази, щоб полегшити вхідні дані, та спростити парсер, та можуть бути написані частково, або повністю, вручну, або для підтримки більшої кількості функцій, або для більшої продуктивності.

1. **Рекурсивний спуск**

Рекурсивний спуск — алгоритм синтаксичного аналізу, будується на основі взаємно рекурсивних процедур (або не рекурсивних еквівалентів), кожна із яких реалізує одну із продукцій граматики.

У синтаксичному аналізі, аналогічно, відомий метод рекурсивного спуску, що ґрунтується на «зашиванні» правил граматики безпосередньо в керуючі конструкції розпізнавача. Ідеї ​​низхідного аналізу, прийняті в LL (1)-граматиках, в ньому повністю зберігаються:

* відбувається послідовний перегляд вхідного рядка зліва-направо;
* черговий символ вхідного рядка є основою вибору однієї з правих частин правил групи при заміні поточного нетерміналу;
* термінальні символи вхідного рядка та правої частини правила «взаємно знищуються»;
* Виявлення нетерміналу у правій частині рекурсивно повторює цей же процес.

У методі рекурсивного спуску вони зазнають таких змін:

* кожному нетерміналу відповідає окрема процедура (функція), що розпізнає (вибирає та «закриває») одну з правих частин правила, що має в лівій частині цей нетермінал (тобто для кожної групи правил пишеться свій розпізнавач);
* у вхідному рядку є покажчик (індекс) на поточний символ, що закривається. Цей символ є підставою для вибору необхідної правої частини правила. Сам вибір «зашитий» у розпізнавачі у вигляді конструкцій if або switch . Правила вибору базуються на побудові безлічі символів, що вибирають, як це прийнято в LL (1)-граматиці;
* перегляд вибраної частини реалізований у тексті процедури-розпізнавача шляхом порівняння очікуваного символу правої частини та поточного символу вхідного рядка;
* якщо у правій частині очікується термінальний символ і він збігається з черговим символом вхідного рядка, символ у вхідному рядку пропускається, а розпізнавач переходить до наступного символу правої частини;
* розбіжність термінального символу правої частини та чергового символу вхідного рядка свідчить про синтаксичну помилку;
* якщо у правій частині зустрічається нетермінальний символ, то для нього необхідно викликати аналогічну процедуру, що розпізнає (функцію).

1. **Код програми**
   1. **Код блоку переводу виразу до постфіксного вигляду**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Security.Cryptography.X509Certificates;

using System.Text;

using System.Threading.Tasks;

using static ST1.Tockenizer;

namespace ST1

{

//<COMP\_OP> ::= '==' | '!=' | '<' | '>' | '<=' | '>='

//<COMP\_TERM> ::= TOK\_ID | TOK\_CONST

//<COMP> ::= <COMP\_TERM> <COMP\_OP> <COMP\_TERM>

//<COMP> ::= TOK\_ID

//<COMP> ::= '(' <LOGIC> ')'

//<LOGIC\_TERM> ::= <COMP>

//<LOGIC\_TERM> ::= 'not' <COMP>

//<LOGIC> ::= <LOGIC2> <LOGIC\_TAIL>

//<LOGIC\_TAIL> ::= 'or' <LOGIC2> <LOGIC\_TAIL>

//<LOGIC\_TAIL> ::= e

//<LOGIC2> ::= <LOGIC\_TERM> <LOGIC2\_TAIL>

//<LOGIC2\_TAIL> ::= 'and' <LOGIC\_TERM> <LOGIC2\_TAIL>

//<LOGIC2\_TAIL> ::= e

class SyntaxNode {

public const int OP = 1;

public const int ARG = 2;

private int type;

private int index;

private SyntaxNode left;

private SyntaxNode right;

public SyntaxNode(int type, SyntaxNode left, int index, SyntaxNode right) {

this.type = type;

this.left = left;

this.right = right;

this.index = index;

}

public Boolean isOp() {

return type == OP;

}

public Boolean isArg()

{

return type == ARG;

}

public static SyntaxNode newOp(SyntaxNode left, int index, SyntaxNode right) {

return new SyntaxNode(OP, left, index, right);

}

public static SyntaxNode newArg(int index)

{

return new SyntaxNode(ARG, null, index, null);

}

static Tockenizer tockenizer = new Tockenizer("");

public static void toList(SyntaxNode node, List<Token> tokens, List<String> list, String tab) {

String res = tab;

Token t = tokens[node.index];

if (node.type == ARG)

{

res += "ARG\t";

}

else

{

res += "OP\t";

}

res += tockenizer.dict[t.subtype] + "\t";

res += t.data;

list.Add(res);

if (node.left != null) toList(node.left, tokens, list, tab + " ");

if (node.right != null) toList(node.right, tokens, list, tab + " ");

}

public static String toPostfixStr(SyntaxNode node, List<Token> tokens, String tab)

{

Token t = tokens[node.index];

if (node.type == ARG)

{

return t.data + tab;

}

else

{

if (t.subtype == Tockenizer.OP\_LOG\_NOT)

{

return toPostfixStr(node.left, tokens, tab) + t.data + tab;

}

else {

return toPostfixStr(node.left, tokens, tab) + toPostfixStr(node.right, tokens, tab) + t.data + tab;

}

}

}

public static void toPostfix(SyntaxNode node, List<Token> list, List<Token> tokens)

{

Token t = tokens[node.index];

if (node.type == ARG)

{

list.Add(t);

}

else

{

if (t.subtype == Tockenizer.OP\_LOG\_NOT)

{

toPostfix(node.left, list, tokens);

list.Add(t);

}

else

{

toPostfix(node.left, list, tokens);

toPostfix(node.right, list, tokens);

list.Add(t);

}

}

}

//( x < y ) and not ( m == 5 )

// x y < m 5 == not and

}

internal class Syntax

{

private List<Token> t;

private int i;

private SyntaxNode root;

public SyntaxNode getRoot()

{

return root;

}

public Syntax(List<Token> tokens)

{

t = tokens;

i = 0;

root = null;

if (!checkSyntax()) throw new Exception("Error check syntax");

}

private bool checkSyntax()

{

return Logic(out root) && t[i].type == Tockenizer.TOK\_EOF;

}

private bool updateIndex(bool exp, out int index) {

if (exp) {

index = i;

i++;

} else {

index = -1;

}

return exp;

}

//<LOGIC> ::= <LOGIC2> <LOGIC\_TAIL>

private bool Logic(out SyntaxNode node)

{

SyntaxNode left;

node = null;

return Logic2(out left) && LogicTail(left, out node);

}

//<LOGIC2> ::= <LOGIC\_TERM> <LOGIC2\_TAIL>

private bool Logic2(out SyntaxNode node)

{

SyntaxNode left;

node = null;

return LogicTerm(out left) && Logic2Tail(left, out node);

}

private bool isNot(out int index) {

return updateIndex(t[i].subtype == Tockenizer.OP\_LOG\_NOT, out index);

}

//<LOGIC\_TERM> ::= <COMP>

//<LOGIC\_TERM> ::= 'not' <COMP>

private bool LogicTerm(out SyntaxNode node)

{

int index;

if (isNot(out index) && Comp(out node))

{

node = SyntaxNode.newOp(node, index, null);

return true;

}

return Comp(out node);

}

private bool isBr(int brType) {

int index;

return updateIndex(t[i].subtype == brType, out index);

}

//<COMP> ::= '(' <LOGIC> ')'

//<COMP> ::= <COMP\_TERM> <COMP\_OP> <COMP\_TERM>

//<COMP> ::= TOK\_ID

private bool Comp(out SyntaxNode node) {

int left, right, op;

int old\_i = i;

node = null;

if (isBr( Tockenizer.TOK\_LPR))

{

return Logic(out node) && isBr(Tockenizer.TOK\_RPR);

}

if (CompTerm(out left) && CompOp(out op) && CompTerm(out right))

{

node = SyntaxNode.newOp(SyntaxNode.newArg(left), op, SyntaxNode.newArg(right));

return true;

}

i = old\_i;

if (CompTerm(out left))

{

node = SyntaxNode.newArg(left);

return true;

}

return false;

}

private bool isTerm() {

return t[i].type == Tockenizer.TOK\_ID || t[i].type == Tockenizer.TOK\_CONST;

}

//<COMP\_TERM> ::= TOK\_ID | TOK\_CONST

private bool CompTerm(out int index) {

return updateIndex(isTerm(), out index);

}

private bool isOp() {

return t[i].type == Tockenizer.TOK\_OP && t[i].subtype <= TOK\_OP + 6;

}

//<COMP\_OP> ::= '==' | '!=' | '<' | '>' | '<=' | '>='

private bool CompOp(out int index) {

return updateIndex(isOp(), out index);

}

private bool isAnd(out int index)

{

return updateIndex(t[i].subtype == Tockenizer.OP\_LOG\_AND, out index);

}

//<LOGIC2\_TAIL> ::= 'and' <LOGIC\_TERM> <LOGIC2\_TAIL>

//<LOGIC2\_TAIL> ::= e

private bool Logic2Tail(SyntaxNode left, out SyntaxNode node)

{

SyntaxNode right;

int index;

node = left;

if (isAnd(out index))

{

return LogicTerm(out right) && Logic2Tail(SyntaxNode.newOp(left, index, right), out node);

}

return true;

}

private bool isOr(out int index)

{

return updateIndex(t[i].subtype == Tockenizer.OP\_LOG\_OR, out index);

}

//<LOGIC\_TAIL> ::= 'or' <LOGIC2> <LOGIC\_TAIL>

//<LOGIC\_TAIL> ::= e

private bool LogicTail(SyntaxNode left, out SyntaxNode node)

{

SyntaxNode right;

int index;

node = left;

if (isOr(out index))

return Logic2(out right) && LogicTail(SyntaxNode.newOp(left, index, right), out node);

return true;

}

}

}

**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 ";

}

public Tockenizer(string src)

{

tokens = new List<Token>();

fill\_dict();

string cur = "";

bool inQuotes = false;

foreach (char c in src)

{

switch (c)

{

case ' ':

if (!cur.Equals(""))

if (!inQuotes)

{

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(CONST\_STR, 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)) 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);

}

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 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, "");

}

}

}

* 1. **Код для створення dll коду**

using System;

using System.Collections;

using System.Collections.Generic;

using System.Drawing;

using System.Linq;

using System.Reflection.Emit;

using System.Security.Principal;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

using System.Windows.Forms.VisualStyles;

using static ST1.Compile;

using static ST1.Tockenizer;

namespace ST1

{

internal class Compile

{

public enum IdType

{

IdUnknown,

IdBool,

IdInt,

IdFloat,

IdStr

}

public class IdInfo

{

private String IdName;

private IdType IdType;

private String LLName;

private int LLField;

public String getIdName() { return IdName; }

public IdType getIdType() { return IdType; }

public String getLLName() { return LLName; }

public int getLLField() { return LLField; }

public bool GetIdInfo(String str)

{

if (string.IsNullOrEmpty(str) || str[0] == ';') return false;

//string[] sa = str.Split(new char[] {' '}, StringSplitOptions.RemoveEmptyEntries);

//if (sa.Length < 2) return false;

IdName = str;

IdType = IdType.IdInt;

// type

//sa[1] = sa[1].ToUpper();

//switch (sa[1])

//{

// case "BOOL":

// IdType = IdType.IdBool;

// break;

// case "INT":

// IdType = IdType.IdInt;

// break;

// case "FLT":

// IdType = IdType.IdFloat;

// break;

// case "STR":

// IdType = IdType.IdStr;

// break;

// default:

// return false;

//}

LLName = "%" + IdName;

LLField = -1;

//if (sa.Length >= 4)

//{

// LLName = "%" + sa[2];

// LLField = int.Parse(sa[3]);

//}

return true;

}

}

private class StackItem {

private String key;

private IdType t;

public StackItem(String key, IdType t) {

this.key = key;

this.t = t;

}

public String getKey() { return key; }

public IdType getIdType() { return t; }

}

private Stack<StackItem> stack;

private Dictionary<String, IdInfo> map;

private List<Token> tokens;

public Compile(List<Token> tokens)

{

map = new Dictionary<String, IdInfo>();

this.tokens = tokens;

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

{

if (tokens[i].type == TOK\_ID)

{

AddIdent(tokens[i].data);

}

}

//TIdMap CreateIdMap(TStrings S)

//{

// TIdMap Result = null;

// if (S == null || S.Count == 0)

// {

// return Result;

// }

// Result = new TIdMap();

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

//// {

//// if (!AddIdent(Result, S[i]))

//// {

//// return null;

//// }

//// }

// return Result;

//}

}

private StackItem getStackItem(String n, IdType t) {

return new StackItem(n, t);

}

private bool AddIdent(String S)

{

IdInfo id = new IdInfo();

if (S == "" || S[0] == ';')

{

return false;

}

if (!id.GetIdInfo(S))

{

return false;

}

map[id.getIdName()] = id;

return true;

}

private String newReg(ref int idx) {

String res = "%var." + idx;

idx++;

return res;

}

private String command\_not(ref int idx)

{

StackItem left = stack.Pop();

String reg = newReg(ref idx);

String result = String.Format("{0} = xor i1 {1}, true\n", reg, left.getKey());

stack.Push(getStackItem(reg, IdType.IdBool));

return result;

}

private String command\_binary(ref int idx, String opCode)

{

StackItem right = stack.Pop();

StackItem left = stack.Pop();

String reg = newReg(ref idx);

String result = string.Format("{0} = {1} i1 {2}, {3}\n", reg, opCode, left.getKey(), right.getKey());

stack.Push(getStackItem(reg, IdType.IdBool));

return result;

}

private String command\_cmp\_int(ref int idx, int cmpCode, String left, String right, String intType="i32")

{

String reg = newReg(ref idx);

String cmpName = getСmpNameInt(cmpCode);

String result = String.Format("{0} = icmp {1} {2} {3}, {4}\n", reg, cmpName, intType, left, right);

stack.Push(getStackItem(reg, IdType.IdBool));

return result;

}

private String command\_cmp\_float(ref int idx, int cmpCode, String left, String right, String floatType="float")

{

String reg = newReg(ref idx);

String cmpName = getСmpNameFloat(cmpCode);

String result = String.Format("{0} = fcmp {1} {2} {3}, {4}\n", reg, cmpName, floatType, left, right);

stack.Push(getStackItem(reg, IdType.IdBool));

return result;

}

private String command\_cmp\_str(ref int idx, int cmpCode, String left, String right)

{

String tmp = newReg(ref idx);

String result = string.Format("{0} = call i32 @strcmp(ptr {1}, ptr {2})\n", tmp, left, right);

String reg = newReg(ref idx);

String cmpName = getСmpNameInt(cmpCode);

result += string.Format("{0} = icmp {1} i32 {2}, 0\n", reg, cmpName, tmp);

stack.Push(getStackItem(reg, IdType.IdBool));

return result;

}

private String command\_cmp(ref int idx, int cmpCode)

{

StackItem right = stack.Pop();

StackItem left = stack.Pop();

String result = "";

switch (left.getIdType())

{

case IdType.IdBool:

result = command\_cmp\_int(ref idx, cmpCode, left.getKey(), right.getKey());

break;

case IdType.IdInt:

result = command\_cmp\_int(ref idx, cmpCode, left.getKey(), right.getKey());

break;

case IdType.IdFloat:

result = command\_cmp\_float(ref idx, cmpCode, left.getKey(), right.getKey());

break;

case IdType.IdStr:

result = command\_cmp\_str(ref idx, cmpCode, left.getKey(), right.getKey());

break;

}

return result;

}

private String floatTextToHex(String text, bool IsSingle)

{

double D;

float S;

if (!float.TryParse(text, out S) || !double.TryParse(text, out D))

return "";

if (IsSingle)

D = S;

long val = BitConverter.DoubleToInt64Bits(D);

return $"0x{val:X16}";

}

private String stringTextToConst(String t, String str)

{

String result = $"@\"{t.ToUpper()}\"";

str += String.Format("{0} = linkonce\_odr dso\_local unnamed\_addr constant [{1} x i8] c\"{2}\0\", align 1\n", result, t.Length + 1, t);

return $"\"{t}\"";

}

private void pushId(ref int idx, IdInfo id, ref String code)

{

if (id.getLLField() == -1)

{

stack.Push(getStackItem( id.getLLName(), id.getIdType()));

return;

}

String ptr = newReg(ref idx);

code += string.Format("{0} = getelementptr inbounds %struct.Customer, {0} {1}, i64 0, i32 {2}\n",

ptr, id.getLLName(), id.getLLField());

if (id.getIdType() == IdType.IdStr)

{

stack.Push(getStackItem(ptr, id.getIdType()));

return;

}

String val = newReg(ref idx);

String res = "";

switch (id.getIdType())

{

case IdType.IdBool:

res = string.Format("{0} = load i8, ptr {1}\n", val, ptr);

break;

case IdType.IdInt:

res = string.Format("{0} = load i32, ptr {1}\n", val, ptr);

break;

case IdType.IdFloat:

res = string.Format("{0} = load float, ptr {1}\n", val, ptr);

break;

}

code += res;

stack.Push(getStackItem(val, id.getIdType()));

}

public String EmitLLVM(int startIdx = 0)

{

String res\_const = "";

String res\_code = "";

if (tokens == null) return "";

stack = new Stack<StackItem>();

int idx = startIdx;

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

{

Token token = tokens[i];

switch (token.type)

{

case TOK\_ID:

pushId(ref idx, map[token.data], ref res\_code);

break;

case TOK\_CONST:

switch (token.subtype)

{

case CONST\_BOOL:

stack.Push(getStackItem(token.data, IdType.IdBool));

break;

case CONST\_INT:

stack.Push(getStackItem(token.data, IdType.IdInt));

break;

case CONST\_STR:

stack.Push(getStackItem(stringTextToConst(token.data, res\_const), IdType.IdStr));

break;

case CONST\_FLT:

stack.Push(getStackItem(floatTextToHex(token.data, true), IdType.IdFloat));

break;

}

break;

case TOK\_OP:

if (token.subtype <= OP\_CMP\_GE)

res\_code += command\_cmp(ref idx, token.subtype);

if (token.subtype == OP\_LOG\_AND)

res\_code += command\_binary(ref idx, "and");

if (token.subtype == OP\_LOG\_OR)

res\_code += command\_binary(ref idx, "or");

if (token.subtype == OP\_LOG\_NOT)

res\_code += command\_not(ref idx);

break;

}

}

StackItem res = stack.Pop();

res\_code += String.Format("ret i1 {0}", res.getKey());

return res\_code;

}

private String getСmpNameInt(int cmpCode) {

String cmpName = "";

switch (cmpCode)

{

case OP\_CMP\_EQ:

cmpName = "eq";

break;

case OP\_CMP\_NE:

cmpName = "ne";

break;

case OP\_CMP\_LS:

cmpName = "slt";

break;

case OP\_CMP\_GR:

cmpName = "sgt";

break;

case OP\_CMP\_LE:

cmpName = "sle";

break;

case OP\_CMP\_GE:

cmpName = "sge";

break;

}

return cmpName;

}

private String getСmpNameFloat(int cmpCode)

{

String cmpName = "";

switch (cmpCode)

{

case OP\_CMP\_EQ:

cmpName = "oeq";

break;

case OP\_CMP\_NE:

cmpName = "one";

break;

case OP\_CMP\_LS:

cmpName = "olt";

break;

case OP\_CMP\_GR:

cmpName = "ogt";

break;

case OP\_CMP\_LE:

cmpName = "ole";

break;

case OP\_CMP\_GE:

cmpName = "oge";

break;

}

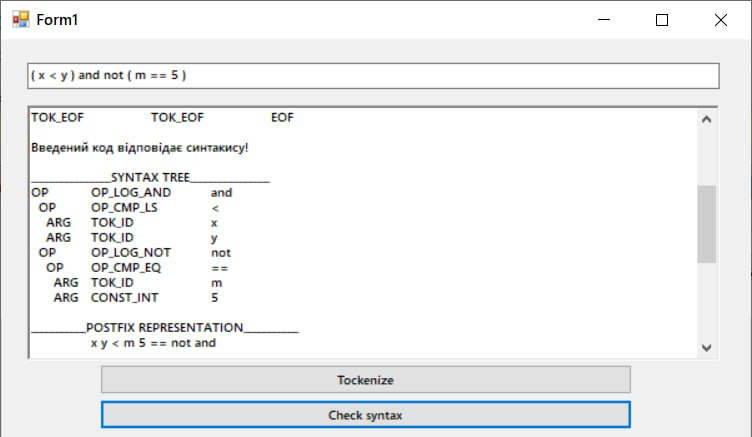
return cmpName;

}

}

}

1. **Результати роботи програми**



**Висновки**

В процесі виконання лабораторної роботи було здобуто навички розробки програми, що генерує DLL код, як працює синтаксичний та лексичний аналізатор, та як він застосовується в програмі, також про роботу токенайзера. Було розроблено програму перевірки аріфметичних виразів, та окремо, навички роботи з DLL кодом