Міністерство освіти і науки України

Національний Технічний Університет України

“Київський Політехнічний Інститут”

Факультет Прикладної Математики

Кафедра Системного програмування та

спеціалізованих комп’ютерних систем

Лабораторна робота №1

з дисципліни

“Інженерія програмного забезпечення.

Основи проектування трансляторів”

Виконав: Перевірив(ла):

Студент групи КВ-22 ”\_\_\_\_” \_\_\_\_\_\_\_\_\_\_\_\_ 2015 р.

Ткаченко Роман Юрійович \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_ балів

Київ 2015

**Завдання**

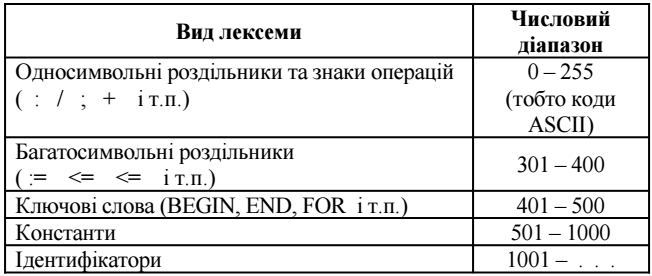
1. Розробити програму лексичного аналізатора (ЛА) для підмножини мови програмування SIGNAL.

2. Лексичний аналізатор має забезпечувати наступні дії:

• видалення (пропускання) пробільних символів: пробіл (код ASCII 32), повернення каретки (код ASCII 13); перехід на новий рядок (код ASCII 10), горизонтальна та вертикальна табуляція (коди ASCII 9 та 11), перехід на нову сторінку (код ASCII 12);

• згортання ключових слів;

• згортання багато-символьних роздільників (якщо передбачаються граматикою варіанту);

• згортання констант із занесенням до таблиці значення та типу константи (якщо передбачаються граматикою варіанту);

• згортання ідентифікаторів;

• видалення коментарів, заданих у вигляді (\*<текст коментаря>\*).

3. Для кодування лексем при їх згортанні необхідно використовувати числові діапазони, вказані в Таблиці.

4. Входом ЛА має бути наступне:

• вхідна програма, написана підмножиною мови SIGNAL відповідно до варіанту;

• таблиця кодів ASCII з атрибутами для визначення токенів;

• таблиця ключових слів;

• таблиця констант, в яку попередньо можуть бути занесені стандартні константи, якщо потрібно;

• таблиця ідентифікаторів, в яку попередньо занесені

наперед визначені ідентифікатори, якщо потрібно.

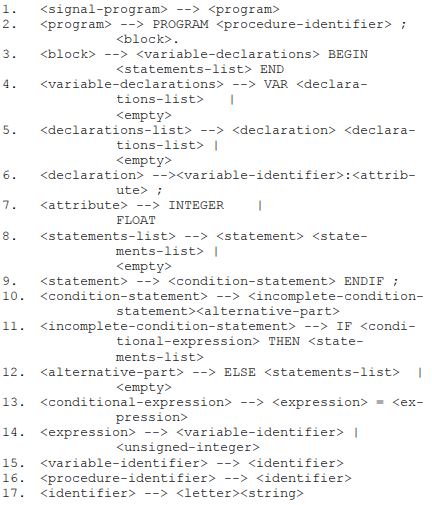
5. Виходом ЛА має бути наступне:

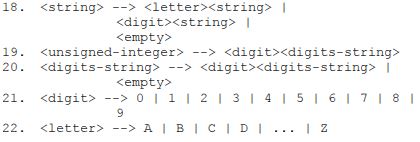
• закодований рядок лексем;

• таблиця констант, що сформована для конкретної програми і яка містить значення та тип констант;

• таблиця ідентифікаторів, що сформована для конкретної програми.

Граматика (варіант 19):





**Програма**

namespace lexer

{

enum attrType { whitespace, constant, identifier, oneSymbDelimiter, manySymbDelimiter, begCom, invalid };

class LexicalAnalizer

{

public LexicalAnalizer(string[] lines, string filepath = "")

{

attributes = SerializeTables.DeserializeAttributes();

identifiers = SerializeTables.DeserializeIdentifiers();

keyWords = SerializeTables.DeserializeKeyWords();

constants = new List<Constant>();

errors = new List<Error>();

this.filepath = filepath;

this.lines = lines;

}

private string[] lines;

private string filepath;

private List<Attributes> attributes;

public List<Identifier> identifiers;

public List<KeyWord> keyWords;

public List<Constant> constants;

public static string commentSymbol = "\*";

public static string endCom = ")";

public static Dictionary<attrType, int> attributesTypes = new Dictionary<attrType, int>

{

{ attrType.whitespace, 0 },

{ attrType.constant, 1},

{ attrType.identifier, 2},

{ attrType.oneSymbDelimiter, 3},

{ attrType.manySymbDelimiter, 4},

{ attrType.begCom, 5},

{ attrType.invalid, 6}

};

private List<Error> errors;

public delegate void WorkDoneHandler(List<LexicalAnalizerOutput> output, List<Error> errors, List<Constant> constants, List<Identifier> identifiers);

public event WorkDoneHandler WorkDone;

public void Analize()

{

List<LexicalAnalizerOutput> result = new List<LexicalAnalizerOutput>();

string[] lines = this.lines;

if (filepath.Length > 1)

{

if (File.Exists(filepath))

{

lines = File.ReadAllLines(filepath);

}

else

{

throw new FileNotFoundException();

}

}

int i = 0; // row number

int j = 0; // symbol number

bool supressedOutput = false; // true if whitespaced

string buffer = "";

int lexCode = 0;

string currLexem = ""; // used only for logging

if (attributes.Count != 0)

{

for (i = 0; i < lines.Count(); i++)

{

string currentLine = lines[i];

j = 0;

while (j < currentLine.Length)

{

supressedOutput = false;

char currentSymbol = currentLine[j];

int symbolAttr = GetSymbolAttr(currentSymbol);

if (symbolAttr == attributesTypes[attrType.whitespace]) // whitespace

{

while (++j < currentLine.Length)

{

currentSymbol = currentLine[j];

symbolAttr = GetSymbolAttr(currentSymbol);

if (symbolAttr != attributesTypes[attrType.whitespace])

break;

}

supressedOutput = true;

}

else if (symbolAttr == attributesTypes[attrType.constant]) // constant

{

buffer = makeBuffer(currentLine, attrType.constant, ref j);

currLexem = buffer;

lexCode = CheckConst(buffer);

}

else if (symbolAttr == attributesTypes[attrType.identifier]) // identifier

{

buffer = makeBuffer(currentLine, attrType.identifier, ref j);

currLexem = buffer;

lexCode = CheckIdentifier(buffer);

}

else if (symbolAttr == attributesTypes[attrType.oneSymbDelimiter]) // divider

{

lexCode = (int)currentSymbol;

currLexem = currentSymbol.ToString();

j++;

}

else if (symbolAttr == attributesTypes[attrType.begCom]) // Comment

{

int indexBeforeSkipping = i;

SkipComment(lines, ref i, ref j);

supressedOutput = true;

if (indexBeforeSkipping != i)

break;

}

else

{

j++;

errors.Add(new Error { message = "\*\*Error\*\* Invalid symbol", row = i, pos = j});

supressedOutput = true;

}

if (!supressedOutput)

{

result.Add(new LexicalAnalizerOutput { code = lexCode, lexem = currLexem, row = i });

}

}

}

}

if (WorkDone != null) WorkDone(result, errors, constants, identifiers);

}

private void SkipComment(string[] lines, ref int i, ref int j)

{

int entry\_i = i;

int entry\_j = j;

string currentLine = lines[i];

char currentSymbol = currentLine[j];

j++;

if (j < currentLine.Length)

{

currentSymbol = currentLine[j];

entry\_j = j;

}

else // error ??

{

errors.Add(new Error { message = "\*\*Error\*\* BegCom symbol without '\*'", row = i, pos = j });

j++; // Analize method will iterate to next row

//entry\_i = i;

//entry\_j = j;

return;

}

if (currentSymbol == (char)commentSymbol[0]) // if (\*

{

j++;

for (int k = i; k < lines.Count(); k++)

{

currentLine = lines[k];

while (j < currentLine.Length - 1)

{

currentSymbol = currentLine[j];

char nextSymbol = currentLine[j + 1];

j++;

if (currentSymbol == commentSymbol[0] && nextSymbol == endCom[0]) // end of Comment found

{

i = k;

j += 1; // skip "\*)"

return;

}

}

j = 0;

}

// ERROR end of comment not found

i = entry\_i; // skip begCom and continue parsing

j = entry\_j;

errors.Add(new Error { message = "\*\*Error\*\* End of comment not found", row = i, pos = j });

return;

}

else

{

// ERROR ("Do u mean comment? '\*' missing")

errors.Add(new Error { message = "\*\*Error\*\* Do u mean comment? '\*' missing", row = i, pos = j });

}

}

private string makeBuffer(string currentLine, attrType type, ref int j) // makes buffer for constant or identifier

{

string buffer = "";

char currentSymbol = currentLine[j];

buffer += currentSymbol.ToString();

while (++j < currentLine.Length)

{

currentSymbol = currentLine[j];

int symbolAttr = GetSymbolAttr(currentSymbol);

// if type == constant it takes only digits, if identifier it takes letters or digits starting from second symbol

if (symbolAttr == attributesTypes[type] || symbolAttr == attributesTypes[attrType.constant])

buffer += currentSymbol.ToString();

else break;

}

return buffer;

}

private int CheckIdentifier(string buffer) // returns lexCode of identifier in buffer

{

int lexCode = 0;

if (keyWords.Count() != 0)

{

if (keyWords.Any(x => x.keyWord == buffer))

{

lexCode = keyWords.First(x => x.keyWord == buffer).id;

return lexCode;

}

}

if (identifiers.Count() != 0)

{

if (identifiers.Any(x => x.name == buffer))

{

lexCode = identifiers.First(x => x.name == buffer).id;

return lexCode;

}

else

{

// creates new identifier with id = maxId + 1

Identifier identifier = new Identifier(buffer, identifierType.user, identifiers.OrderByDescending(x => x.id).First().id + 1);

identifiers.Add(identifier);

lexCode = identifier.id;

return lexCode;

}

}

else

{

Identifier identifier = new Identifier(buffer, identifierType.user);

identifiers.Add(identifier);

lexCode = identifier.id;

return lexCode;

}

}

private int CheckConst(string buffer) // returns lexCode, if not present in constants returns new id

{

int lexCode = 0;

if (constants.Count() == 0)

{

Constant constant = new Constant(Convert.ToInt32(buffer));

constants.Add(constant);

lexCode = constant.id;

}

else

{

if (!constants.Any(x => x.value == Convert.ToInt32(buffer))) // if no consts has the same value

{

// creates new const with id = maxId + 1

Constant constant = new Constant(Convert.ToInt32(buffer), constants.OrderByDescending(x => x.id).First().id + 1);

constants.Add(constant);

lexCode = constant.id;

}

else lexCode = constants.First(x => x.value == Convert.ToInt32(buffer)).id; // if exists get id

}

return lexCode;

}

private int GetSymbolAttr(char symbol)

{

if (attributes.Count != 0)

return attributes.First(x => x.symbol == symbol).type;

else

return attributesTypes[attrType.invalid];

}

public List<Constant> GetConstants()

{

return constants;

}

public List<Identifier> GetIdentifiers()

{

return identifiers;

}

}

}