**РОЗРАХУНКОВО-ГРАФІЧНА РОБОТА**

З дисципліни «Інженерія програмного забезпечення»

На тему: «Розробка синтаксичного аналізатора»

Варіант #8

|  |  |
| --- | --- |
| **Виконав**  Студент групи КВ-21  Комарницький О.Б.  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | **Перевірила**  Доцент кафедри СП і СКС  Бояринова Ю. Є.  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |

**Варіант #8**

Низхідний розбір за алгоритмом рекурсивного спуску

**Мета лабораторної роботи**

Метою синтаксичної роботи “Розробка синтаксичного аналізатора” є засвоєння теоретичного матеріалу та набуття практичного досвіду і практичних навичок розробки синтаксичних аналізаторів (парсерів)

**Постановка задачі**

1. Розробити програму синтаксичного аналізатора для підмножини мови програмування SIGNAL згідно граматики за варіантом.
2. Програма має забезпечувати наступні дії:
   * + - Читання рядка лексем та таблиць, згенерованих лексичним аналізатором, який було розроблено в лабораторній роботі “Розробка лексичного аналізатора”.
       - Синтаксичний аналіз (розбір) програми, поданої рядком лексем (алгоритм синтаксичного аналізатора вибирається за варіантом)
       - Побудова дерева розбору
       - Формування таблиць ідентифікаторів та різних констант х повною інформацією, необхідною для генерування коду
       - Формування лістингу вхідної програми з повідомленнями про лексичні та синтаксичні помилки.
3. Для програмування може бути використана довільна алгоритмічна мова програмування високого рівня. Якщо обрана мова програмування має конструкції або бібліотеки для роботи з регулярними виразами, то використання цих конструкцій та/або бібліотек строго заборонено.
4. Входом СА має бути наступне:
   * + - Закодований рядок лексем.
       - Таблиці ідентифікаторів, числових, символьних та рядкових констант (якщо це передбачено граматикою варіанту), згенеровані лексичним аналізатором.
       - Вхідна програма підмножини мови програмування SIGNAL згідно з варіантом (необхідна для формування лістингу програми)
5. Виходом синтаксичного аналізатора має бути наступне:

* Дерево розбору вхідної програми;
* Таблиці ідентифікаторів та різних констант з повною інформацією, необхідною для генерування коду;
* Лістинг вхідної програми з повідомленнями про лексичні та синтаксичні помилки.

**Зміст звіту**

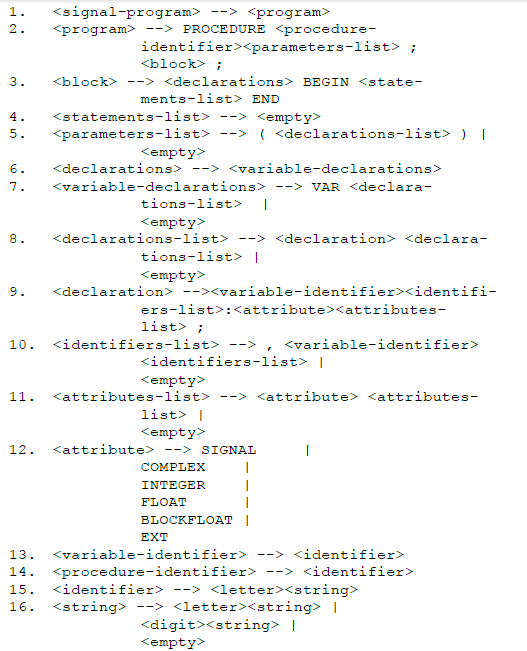
Зміст оформлюється згідно вимог до розрахунково-графічної роботи і має мати наступне:

* + - Титульний аркуш;
    - Індивідуальне завдання згідно до варіанту;
    - Перетворену граматику (якщо це необхідно для реалізації даного алгоритму СА)
    - Таблиці та граф автомату синтаксичного аналізатора (крім варіантів з синтаксичним за методом рекурсивного спуску)
    - Лістинг програми синтаксичного аналізатора мовою програмування
    - Контрольні приклади, необхідні для демонстрації всіх конструкцій заданої граматики, а також всіх можливих помилкових ситуацій
    - Опис кожного контрольного прикладу має містити вхідний рядок лексем (результат роботи ЛА, розробленого в лабораторній роботі “Розробка лексичного аналізатора”), згенеровані таблиці, а також зображення побудованого дерева розбору.
    - В якості графічної частини розрахунково-графічної роботи включити до звіту рисунки дерев розбору, побудованих для всіх контрольних прикладів (мінімум 5 прикладів і дерев)

До звіту на довільному носії інформації мають бути додані:

* + - Файл з текстом звіту;
    - Проект працюючої програми мовою програмування;
    - Завантажувальний модуль програми з необхідними файлами даних.

**Граматика мови**

 Для побудови СА визначимо назви процедур, що відповідають нетерміналам граматики таким чином:

<program>🡪 PRG

<procedure-identifier > 🡪 PRCIDN

<parameters-list>🡪 PRMLST

<block> 🡪 BLK

<declarations> 🡪 DCLS

<declarations-list> 🡪 DCLLST

<variable-declarations> 🡪 VARDCLS

<declaration> 🡪 DCL

<variable-identifier> 🡪 VAR

<identifiers-list> 🡪 IDNLST

<attribute> 🡪 ATTR

<attributes-list> 🡪 ATTRLST

<procedure-identifier> 🡪 PRIDN

<identifier> 🡪 IDN

**Лістинг програми СА**

**Оголошення класу Parser:**

#include "Scanner.h"

using namespace std;

class Parser

{

public:

Parser(list<int> , list<\_word\_>, list<\_word\_> );

~Parser();

void PRG();

void PRCIDN(int , int );

void IDN(int, int );

void PRMLST(int , int );

void DCLLST(int , int );

void DCL(int , int );

void VAR(int , int );

void ATTR(int, int );

void BLK(int , int);

void DCLS(int , int );

void VARDCLS(int , int );

void ERR(char \* );

private:

list<\_word\_> IDENTIFIERS;

list<\_word\_> KEYS;

list<int> codedLine;

int searchIdentifier(string);

string getWordByKey(int, list<\_word\_>);

void printTreeElement(int, string);

};

**Опис функцій класу Parser:**

#include "Parser.h"

Parser::Parser(list<int> \_codedLine, list<\_word\_> \_IDENTIFIERS, list<\_word\_> \_KEYS)

{

codedLine = \_codedLine;

IDENTIFIERS = \_IDENTIFIERS;

KEYS = \_KEYS;

}

Parser::~Parser()

{}

string Parser::getWordByKey(int code, list<\_word\_> table)

{

list<\_word\_>::iterator it;

for (it = table.begin(); it != table.end(); ++it)

{

if ((\*it).Code == code) return (\*it).Word;

}

return "";

}

int Parser::searchIdentifier(string word)

{

list<\_word\_>::iterator it;

for (int i = 0; i < (int)word.size(); i++)

word[i] = toupper(word[i]);

int index = 0;

for (it = IDENTIFIERS.begin(); it != IDENTIFIERS.end(); ++it)

{

if ((\*it).Word == word) return (\*it).Code;

index++;

}

return -1;

}

void Parser::PRG()

{

int tabCounter = 0;

printTreeElement(tabCounter, "<program>");

if (getWordByKey(codedLine.front(), KEYS) != "PROCEDURE")

ERR("SYNTAX ERR: Expected 'PROCEDURE'");

printTreeElement(tabCounter + 1, "PROCEDURE");

codedLine.pop\_front();

PRCIDN(codedLine.front(), tabCounter + 1);

PRMLST(codedLine.front(), tabCounter + 1);

BLK(codedLine.front(), tabCounter + 1);

if ((char)codedLine.front() != ';')

ERR("SYNTAX ERR: Expected ';' in the end of program");

codedLine.pop\_front();

if (!codedLine.empty())

ERR("SYNTAX ERR: Found symbols after 'END;'");

printTreeElement(tabCounter, "</program>");

}

void Parser::BLK(int currentLexem, int tabCounter)

{

printTreeElement(tabCounter, "<block>");

DCLS(currentLexem, tabCounter + 1);

if (getWordByKey(codedLine.front(), KEYS) != "BEGIN")

ERR("SYNTAX ERR: Expected 'BEGIN'");

printTreeElement(tabCounter, "BEGIN");

codedLine.pop\_front();

if (getWordByKey(codedLine.front(), KEYS) != "END")

ERR("SYNTAX ERR: Expected 'END'");

printTreeElement(tabCounter, "END");

codedLine.pop\_front();

printTreeElement(tabCounter, "</block>");

}

void Parser::DCLS(int currentLexem, int tabCounter)

{

printTreeElement(tabCounter, "<declarations-list>");

VARDCLS(currentLexem, tabCounter + 1);

printTreeElement(tabCounter, "</declarations-list>");

}

void Parser::VARDCLS(int currentLexem, int tabCounter)

{

printTreeElement(tabCounter, "<var-declarations>");

if (getWordByKey(codedLine.front(), KEYS) != "VAR")

ERR("SYNTAX ERR: Expected 'VAR'");

codedLine.pop\_front();

DCLLST(codedLine.front(), tabCounter + 1);

printTreeElement(tabCounter, "</var-declarations>");

}

void Parser::PRCIDN(int currentLexem, int tabCounter)

{

printTreeElement(tabCounter, "<procedure-identifier>");

IDN(currentLexem, tabCounter + 1);

printTreeElement(tabCounter, "</procedure-identifier>");

}

void Parser::IDN(int currentLexem, int tabCounter)

{

printTreeElement(tabCounter, "<identifier>");

if (searchIdentifier(getWordByKey(currentLexem, IDENTIFIERS)) < 0)

ERR("SYNTAX ERR: Expected parameter identifier");

printTreeElement(tabCounter + 1, getWordByKey(currentLexem, IDENTIFIERS));

codedLine.pop\_front();

printTreeElement(tabCounter, "</identifier>");

}

void Parser::PRMLST(int currentLexem, int tabCounter)

{

printTreeElement(tabCounter, "<parameters-list>");

if ((char)currentLexem == '(')

{

codedLine.pop\_front();

DCLLST(codedLine.front(), tabCounter + 1);

}

else

ERR("SYNTAX ERR: Expected '(' in parameters list");

if ((char)codedLine.front() == ')')

codedLine.pop\_front();

else

ERR("SYNTAX ERR: Expected ')' parameters list");

if ((char)codedLine.front() == ';')

codedLine.pop\_front();

else

ERR("SYNTAX ERR: Expected ';' parameters list");

printTreeElement(tabCounter, "</parameters-list>");

}

void Parser::DCLLST(int currentLexem, int tabCounter)

{

printTreeElement(tabCounter, "<declarations-list>");

DCL(currentLexem, tabCounter + 1);

while ((char)codedLine.front() == ';')

{

codedLine.pop\_front();

DCL(codedLine.front(), tabCounter + 1);

}

printTreeElement(tabCounter, "</declarations-list>");

}

void Parser::DCL(int currentLexem, int tabCounter)

{

printTreeElement(tabCounter, "<declaration>");

VAR(currentLexem, tabCounter + 1);

while ((char)codedLine.front() == ',')

{

codedLine.pop\_front();

IDN(codedLine.front(), tabCounter + 1);

}

if ((char)codedLine.front() == ':')

{

codedLine.pop\_front();

ATTR(codedLine.front(), tabCounter + 1);

while ((char)codedLine.front() == ' ')

ATTR(codedLine.front(), tabCounter + 1);

}

else

ERR("SYNTAX ERR: Expected ':' in declaration");

printTreeElement(tabCounter, "</declaration>");

}

void Parser::ATTR(int currentLexem, int tabCounter)

{

printTreeElement(tabCounter, "<attribute>");

string currentType = getWordByKey(currentLexem, KEYS);

if (currentType != "SIGNAL" && currentType != "COMPLEX" && currentType != "INTEGER"

&& currentType != "FLOAT" && currentType != "BLOCKFLOAT" && currentType != "EXT")

ERR("SYNTAX ERR: Expected attribute");

else

{

printTreeElement(tabCounter + 1, currentType);

codedLine.pop\_front();

}

printTreeElement(tabCounter, "</attribute>");

}

void Parser::VAR(int currentLexem, int tabCounter)

{

printTreeElement(tabCounter, "<var>");

IDN(currentLexem, tabCounter + 1);

printTreeElement(tabCounter, "</var>");

}

void Parser::ERR(char \* message)

{

cout << message << endl;

exit(0);

}

void Parser::printTreeElement(int tabCounter, string word)

{

string tabs = "";

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

tabs += '\t';

cout << tabs << word << endl;

}

**Програма СА:**

#include "Scanner.h"

#include "Parser.h"

int main()

{

Scanner scanner = Scanner();

if (!scanner.setFile("Program.txt"))

{

cout << "File can't be opened" << endl;

return 0;

}

list<int> codedLine = scanner.processFile();

cout << "All identifiers:" << endl;

scanner.showIdentifers();

cout << "Encoded line:" << endl;

for each (int symbol in codedLine)

cout << symbol << " ";

cout << endl;

Parser parser = Parser(codedLine, scanner.getIDENTIFIERSTable(), scanner.getKEYSTable());

parser.PRG();

cout << "Run successfully ..." << endl;

return 0;

}

**Тестування СА:**

* Перевіримо коректний код програми:

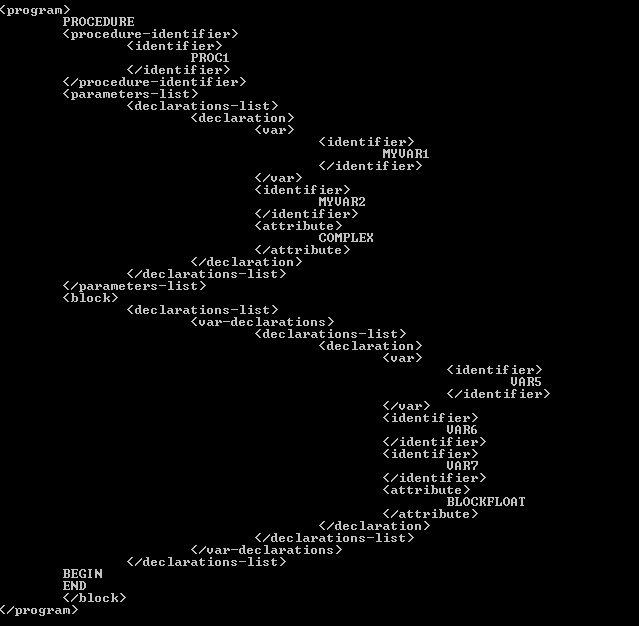
PROCEDURE Proc1(MYVAR1, MYVAR2: COMPLEX);

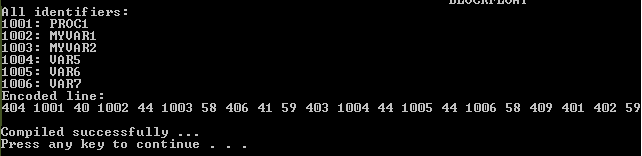
VAR var5, var6, var7: BLOCKFLOAT

BEGIN

END;

Дерево розбору програми:



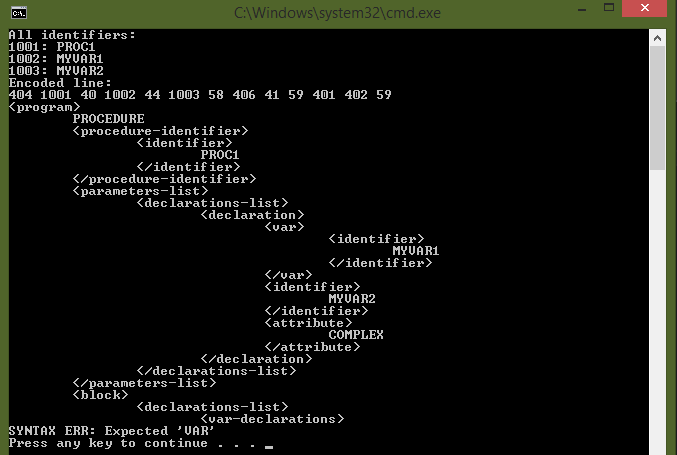


* Дослідимо випадок відсутності опису змінних:

PROCEDURE Proc1(MYVAR1, MYVAR2: COMPLEX);

BEGIN

END;



Виявлена синтаксична помилка ‘VAR’

* Дослідимо випадок відсутнього атрибута змінних:

PROCEDURE Proc1(MYVAR1, MYVAR2);

VAR var5, var6, var7: BLOCKFLOAT

BEGIN

END;



Виявлена синтаксична помилка ‘:’