**Національний технічний університет України**

**“Київський політехнічний інститут”**

**Факультет прикладної математики**

**Кафедра системного програмування і спеціалізованих комп’ютерних систем**

**ЛАБОРАТОРНА РОБОТА №2**

***з дисципліни***

***“*** **ОСНОВИ ПРОЕКТУВАННЯ ТРАНСЛЯТОРІВ*”***

**ТЕМА: “** **РОЗРОБКА ГЕНЕРАТОРА КОДУ”**

**Група: КВ-11**

**Виконала: Нестерук А.О.**

**Оцінка:**

**Київ – 2024**

# **Постановка задачі**

1. Разробити програму генератора коду (ГК) для подмножини мови програмування SIGNAL, заданої за варіантом.
2. Програма генератора коду має забезпечувати:
   * читання дерева розбору та таблиць, створених синтаксичним аналізатором, що було розроблено в розрахунково-графічній роботі;
   * виявлення семантичних помилок;
   * генерацію коду та/або побудову внутрішніх таблиць для гене- рації коду.
3. Зкомпонувати повний компілятор, що складається з розробле- них раніше лексичного та синтаксичного аналізаторів і генератора коду, який забезпечує наступне:
   * генерацію коду та/або побудову внутрішніх таблиць для гене- рації коду;
   * формування лістингу вхідної програми з повідомленнями про лексичні, синтаксичні та семантичні помилки.

### ***Варіант 16***

1. <signal-program> --> <program>
2. <program> --> PROGRAM <procedure-identifier> ;

<block>.

1. <block> --> BEGIN <statements-list> END
2. <statements-list> --> <statement> <statements-list>

|

<empty>

1. <statement> --> <condition-statement> ENDIF ; | WHILE <conditional-expression> DO

<statements-list> ENDWHILE ;

1. <condition-statement> --> <incomplete-condition- statement><alternative-part>
2. <incomplete-condition-statement> --> IF

<conditional-expression> THEN <statements- list>

1. <alternative-part> --> ELSE<statements-list> |

<empty>

1. <conditional-expression> -->

<expression><comparison-operator>

<expression>

1. <comparison-operator> --> < |

<= |

= |

<> |

>= |

>

1. <expression> --> <variable-identifier> |

<unsigned-integer>

1. <variable-identifier> --> <identifier>
2. <procedure-identifier> --> <identifier>
3. <identifier> --> <letter><string>
4. <string> --> <letter><string> |

<digit><string> |

<empty>

1. <unsigned-integer> --> <digit><digits-string>
2. <digits-string> --> <digit><digits-string> |

<empty>

1. <digit> --> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
2. <letter> --> A | B | C | D | ... | Z

# **Код програми**

**Token.h**

#**pragma once**

**using** **namespace** std;

**struct** Token

{

**int** Code;

**int** Row;

**int** Column;

string Name;

};

**Token.cpp**

#include "Lexer.h"

#include "Token.h"

**void** Lexer::Add\_Token(**int** Code, **int** Row, **int** Column, string Name)

{

Token symbol;

symbol.Code = Code;

symbol.Row = Row;

symbol.Column = Column;

symbol.Name = Name;

Lexems.push\_back(symbol);

}

**Lexer.h**

#**pragma once**

#include <string>

#include <vector>

#include <iostream>

#include <iomanip>

#include <fstream>

#include "Token.h"

**using** **namespace** std;

#define DM\_BASE 301

#define KW\_BASE 401

#define CONST\_BASE 501

#define IDN\_BASE 1001

#define ERR\_BASE 2001

**class** Lexer

{

**private**:

fstream Input\_File;

vector <Token> Lexems;

vector <Token> Key\_Words;

vector <Token> Delimiters;

vector <string> Identifiers;

vector <string> Constants;

**char** symbol;

**int** Row;

**int** Save\_Row;

**int** Column;

**int** Save\_Column;

**int** Lex\_Errors\_Counter;

**int** Identifiers\_Counter;

**int** Constants\_Counter;

**int** Attributes[128];

**bool** ERROR\_FLAG;

**private**:

**void** Input\_SYMBOL();

**void** Add\_Token(**int** Code, **int** Row, **int** Column, string Name);

**void** INP();

**void** WS();

**void** DIG();

**void** LET();

**void** BCOM();

**void** COM(string Symbols\_Buffer);

**void** ECOM(string Symbols\_Buffer);

**void** DM1();

**void** DM2();

**void** ERR(string Symbols\_Buffer);

**void** Form\_Information\_table();

**void** Set\_Symbol\_Attributes();

**int** Search\_Identifiers\_table(string Identifier);

**int** Search\_Key\_Words\_table(string KKey\_Word);

**int** Search\_Constants\_table(string Constant);

**int** Search\_Delimiters\_table(string Delimiter);

**public**:

Lexer();

**void** Start\_lex\_analysis(string File);

**void** Listing\_Lexer(string File);

};

**lexer.cpp**

#include "Lexer.h"

Lexer::Lexer() : Row(1), Column(0), Save\_Row(1), Save\_Column(1), symbol(' '), ERROR\_FLAG(0)

{

Form\_Information\_table();

Set\_Symbol\_Attributes();

}

**void** Lexer::Start\_lex\_analysis(string File)

{

Input\_File.open(File);

**if** (!Input\_File.is\_open())

{

cout << "Failed to open!!! (" << File << ")" << endl;

**return**;

}

**else** **if** (Input\_File.peek() == '\0')

{

cout << "File is empty!!! (" << File << ")" << endl;

**return**;

}

**else** {

INP();

Listing\_Lexer(File);

}

}

**Information\_Table.cpp**

#include "Lexer.h"

**int** Lexer::Search\_Identifiers\_table(string Identifier)

{

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

{

**if** (Identifiers[i] == Identifier)

**return** i;

}

**return** -1;

}

**int** Lexer::Search\_Key\_Words\_table(string Key\_Word)

{

**for** (**int** i = 0; i < Key\_Words.size(); i++)

{

**if** (Key\_Words[i].Name == Key\_Word)

**return** i;

}

**return** -1;

}

**int** Lexer::Search\_Delimiters\_table(string Delimiter)

{

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

{

**if** (Delimiters[i].Name == Delimiter)

**return** i;

}

**return** -1;

}

**int** Lexer::Search\_Constants\_table(string Constant)

{

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

{

**if** (Constants[i] == Constant)

**return** i;

}

**return** -1;

}

**void** Lexer::Form\_Information\_table()

{

Identifiers\_Counter = IDN\_BASE;

Constants\_Counter = CONST\_BASE;

Lex\_Errors\_Counter = ERR\_BASE;

Token symbol;

string Key\_Words\_Array[]{ "PROGRAM", "BEGIN", "END", "ENDIF", "WHILE", "DO", "ENDWHILE", "IF", "THEN", "ELSE" };

**for** (**int** i = 0; i < size(Key\_Words\_Array); i++)

{

symbol.Name = Key\_Words\_Array[i];

symbol.Code = KW\_BASE + i;

Key\_Words.push\_back(symbol);

}

string Double\_Delimiters\_Array []{ "<=", "<>", ">=" };

**for** (**int** i = 0; i < size(Double\_Delimiters\_Array); i++)

{

symbol.Name = Double\_Delimiters\_Array [i];

symbol.Code = DM\_BASE + i;

Delimiters.push\_back(symbol);

}

}

**void** Lexer::Set\_Symbol\_Attributes()

{

**for** (**int** i = 0; i < 128; i++)

{

**if** ((i == 8) || (i == 9) || (i == 10) || (i == 13) || (i == 32))

Attributes[i] = 0;

**else** **if** ((i > 47) && (i < 58))

Attributes[i] = 1;

**else** **if** ((i > 64) && (i < 91))

Attributes[i] = 2;

**else** **if** ((i == '=') || (i == '.') || (i == ';'))

Attributes[i] = 3;

**else** **if** ((i == '<') || (i == '>'))

Attributes[i] = 4;

**else** **if** (i == '(')

Attributes[i] = 5;

**else**

Attributes[i] = 6;

}

}

**Lexer\_scan.cpp**

#include "Lexer.h"

**void** Lexer::INP()

{

Input\_SYMBOL();

**while** (!Input\_File.eof())

{

**switch** (Attributes[symbol])

{

**case** 0:

WS();

**break**;

**case** 1:

DIG();

**break**;

**case** 2:

LET();

**break**;

**case** 3:

DM1();

**break**;

**case** 4:

DM2();

**break**;

**case** 5:

BCOM();

**break**;

**case** 6:

ERR("");

**break**;

}

}

}

**void** Lexer::Input\_SYMBOL() {

symbol = Input\_File.get();

**if** (symbol == '\n')

{

Row++;

Column = 0;

}

**else** **if** (symbol == '\t')

Column += 4;

**else**

Column++;

}

**void** Lexer::WS()

{

**do**

{

Input\_SYMBOL();

} **while** (Attributes[symbol] == 0);

}

**void** Lexer::DIG()

{

Save\_Row = Row;

Save\_Column = Column;

string symbols\_Buffer = "";

**while** ((!Input\_File.eof()) && (Attributes[symbol] == 1))

{

symbols\_Buffer += symbol;

Input\_SYMBOL();

}

**int** searching\_result = Search\_Constants\_table(symbols\_Buffer);

**if** (searching\_result == -1)

{

Add\_Token(Constants\_Counter, Save\_Row, Save\_Column, symbols\_Buffer);

Constants.push\_back(symbols\_Buffer);

Constants\_Counter++;

}

**else**

Add\_Token(searching\_result + CONST\_BASE, Save\_Row, Save\_Column, symbols\_Buffer);

}

**void** Lexer::LET()

{

Save\_Row = Row;

Save\_Column = Column;

string symbols\_Buffer = "";

**while** ((!Input\_File.eof()) && ((Attributes[symbol] == 2) || (Attributes[symbol] == 1)))

{

symbols\_Buffer += symbol;

Input\_SYMBOL();

}

**int** searching\_result = Search\_Key\_Words\_table(symbols\_Buffer);

**if** (searching\_result == -1)

{

searching\_result = Search\_Identifiers\_table(symbols\_Buffer);

**if** (searching\_result == -1)

{

Add\_Token(Identifiers\_Counter, Save\_Row, Save\_Column, symbols\_Buffer);

Identifiers.push\_back(symbols\_Buffer);

Identifiers\_Counter++;

}

**else**

Add\_Token(searching\_result + IDN\_BASE, Save\_Row, Save\_Column, symbols\_Buffer);

}

**else**

Add\_Token(Key\_Words[searching\_result].Code, Save\_Row, Save\_Column, Key\_Words[searching\_result].Name);

}

**void** Lexer::BCOM()

{

string symbols\_Buffer;

symbols\_Buffer += symbol;

Input\_SYMBOL();

**if** (symbol == '\*')

COM(symbols\_Buffer);

**else**

ERR(symbols\_Buffer);

}

**void** Lexer::COM(string Symbols\_Buffer)

{

Symbols\_Buffer += symbol;

Input\_SYMBOL();

**if** (symbol == '\*')

{

ECOM(Symbols\_Buffer);

}

**else** **if** (Input\_File.eof()) {

ERR(Symbols\_Buffer);

}

**else** {

COM(Symbols\_Buffer);

}

}

**void** Lexer::ECOM(string Symbols\_Buffer)

{

Symbols\_Buffer += symbol;

Input\_SYMBOL();

**if** (symbol == ')') {

Input\_SYMBOL();

**return**;

}

**else** **if** (symbol == '\*') {

ECOM(Symbols\_Buffer);

}

**else** **if** (Input\_File.eof()) {

Symbols\_Buffer += symbol;

ERR(Symbols\_Buffer);

}

**else** {

COM(Symbols\_Buffer);

}

}

**void** Lexer::DM1()

{

string Symbols\_Buffer = "";

Symbols\_Buffer += symbol;

Add\_Token(symbol, Row, Column, Symbols\_Buffer);

Input\_SYMBOL();

**return**;

}

**void** Lexer::DM2()

{

Save\_Row = Row;

Save\_Column = Column;

string Symbols\_Buffer = "";

Symbols\_Buffer += symbol;

**int** searching\_result;

**if** (symbol == '<') {

Input\_SYMBOL();

Symbols\_Buffer += symbol;

searching\_result = Search\_Delimiters\_table(Symbols\_Buffer);

**if** (searching\_result == -1)

Add\_Token('<', Save\_Row, Save\_Column, "<");

**else**

{

Add\_Token(Delimiters[searching\_result].Code, Save\_Row, Save\_Column, Delimiters[searching\_result].Name);

Input\_SYMBOL();

}

}

**else** **if** (symbol == '>') {

Input\_SYMBOL();

Symbols\_Buffer += symbol;

searching\_result = Search\_Delimiters\_table(Symbols\_Buffer);

**if** (searching\_result == -1)

Add\_Token('>', Save\_Row, Save\_Column, ">");

**else**

{

Add\_Token(Delimiters[searching\_result].Code, Save\_Row, Save\_Column, Delimiters[searching\_result].Name);

Input\_SYMBOL();

}

}

}

**void** Lexer::ERR(string Symbols\_Buffer)

{

Add\_Token(Lex\_Errors\_Counter, Save\_Row, Save\_Column, Symbols\_Buffer + symbol);

Lex\_Errors\_Counter++;

ERROR\_FLAG = 1;

Input\_SYMBOL();

}

**Parser.cpp**

#include "Parser.h"

**void** Parser::Start\_syntax\_analysis(string File) {

Parse\_tree = Initialization\_Tree();

**if** (Program(Parse\_tree)) {

Tree\_Listing(File);

cout << "Syntax analysis completed successfully" << endl;

}

}

Parser::Parser(Lexer& object) : Lexem\_Index(0), ErrorString(""), Key\_Words(object.Key\_Words), Lexems(object.Lexems),Check\_IF(0),Check\_WHILE(0)

{

}

Tree\_Node\* Parser::Initialization\_Tree() {

Tree\_Node\* root = **new** Tree\_Node;

root->Code = -1;

root->Down = NULL;

root->Row = 0;

root->Column = 0;

root->Is\_Terminal = **false**;

root->Name = "";

root->Right = NULL;

root->NonTerminal\_name = "<signal-program>";

**return** root;

}

**Parser.h**

#**pragma once**

#include <string>

#include <vector>

#include <iostream>

#include <iomanip>

#include <fstream>

#include "Lexer.h"

**using** **namespace** std;

**struct** Tree\_Node {

**int** Code;

string Name;

**int** Row;

**int** Column;

string NonTerminal\_name;

**bool** Is\_Terminal;

Tree\_Node\* Right;

Tree\_Node\* Down;

};

**class** Parser

{

**private**:

vector <Token> Lexems;

vector <Token> Key\_Words;

**int** Lexem\_Index;

//bool Check\_IF;

//bool Check\_WHILE;

string TreeString;

string ErrorString;

**int** Check\_WHILE;

**int** Check\_IF;

**private**:

**bool** Program(Tree\_Node\* Parser\_Tree);

**bool** Block(Tree\_Node\* Parser\_Tree);

**bool** Statements\_list(Tree\_Node\* Parser\_Tree);

**bool** Statement(Tree\_Node\* Parser\_Tree);

**bool** Condition\_statement(Tree\_Node\* Parser\_Tree);

**bool** Condition\_expression(Tree\_Node\* Parser\_Tree);

**bool** Incomplete\_conditionstatement(Tree\_Node\* Parser\_Tree);

**bool** Alternative\_part(Tree\_Node\* Parser\_Tree);

**bool** Comparison\_operator(Tree\_Node\* Parser\_Tree);

**bool** Expression(Tree\_Node\* Parser\_Tree);

**bool** Identifier(Tree\_Node\* Parser\_Tree);

**void** Check\_Dot\_End();

**void** Check\_End\_Lexems();

**bool** Check\_After\_File();

**void** Errors(**int** Row, **int** Column, string message);

**void** Write\_Tree(Tree\_Node\* Root, **const** string space, ofstream& outputFile);

**void** Tree\_Listing(string File);

Tree\_Node\* Initialization\_Tree();

Tree\_Node\* Add\_New\_Tree\_Node(**int** code,**int** row, **int** column, string Name, string Func, **bool** isterminal);

**public**:

Tree\_Node\* Parse\_tree;

Parser(Lexer& object);

**void** Start\_syntax\_analysis(string File);

};

**Parsing.cpp**

#include "Parser.h"

**bool** Parser::Program(Tree\_Node\* Parser\_Tree) {

Check\_End\_Lexems();

Tree\_Node\* Current\_Node = Parser\_Tree;

Current\_Node = Current\_Node->Down = Add\_New\_Tree\_Node(-1, Lexems[Lexem\_Index].Row, Lexems[Lexem\_Index].Column, "","<program>", **false**);

**if** (Lexems[Lexem\_Index].Code != 401) {

Errors(Lexems[Lexem\_Index].Row, Lexems[Lexem\_Index].Column, ": Keyword \"PROGRAM\" is missed\n");

}

Current\_Node->Down = Add\_New\_Tree\_Node(Lexems[Lexem\_Index].Code, Lexems[Lexem\_Index].Row, Lexems[Lexem\_Index].Column, Lexems[Lexem\_Index].Name, "", **true**);

Current\_Node = Current\_Node->Down;

Lexem\_Index++;

Check\_End\_Lexems();

Current\_Node->Right = Add\_New\_Tree\_Node(-1, Lexems[Lexem\_Index].Row, Lexems[Lexem\_Index].Column, "", "<procedureidentifier>",**false**);

Current\_Node = Current\_Node->Right;

Identifier(Current\_Node);

Lexem\_Index++;

Check\_End\_Lexems();

**if** (Lexems[Lexem\_Index].Code != ';') {

Errors(Lexems[Lexem\_Index].Row, Lexems[Lexem\_Index].Column, ": \";\" is missed\n");

}

Current\_Node->Right = Add\_New\_Tree\_Node(Lexems[Lexem\_Index].Code, Lexems[Lexem\_Index].Row, Lexems[Lexem\_Index].Column, Lexems[Lexem\_Index].Name, "", **true**);

Current\_Node = Current\_Node->Right;

Lexem\_Index++;

Check\_End\_Lexems();

Current\_Node->Right = Add\_New\_Tree\_Node(-1, Lexems[Lexem\_Index].Row, Lexems[Lexem\_Index].Column, "", "<block>",**false**);

Current\_Node = Current\_Node->Right;

Check\_End\_Lexems();

Block(Current\_Node);

Lexem\_Index++;

Check\_Dot\_End();

**if** (Lexems[Lexem\_Index].Code != '.') {

Errors(Lexems[Lexem\_Index].Row, Lexems[Lexem\_Index].Column, ": \".\" is missed\n");

}

Current\_Node->Right = Add\_New\_Tree\_Node(Lexems[Lexem\_Index].Code, Lexems[Lexem\_Index].Row, Lexems[Lexem\_Index].Column, Lexems[Lexem\_Index].Name, "", **true**);

Lexem\_Index++;

**if** (Check\_After\_File()) {

Errors(-1, -1, "Unexpected symbol out the program!\n");

}

**else** {

**return** **true**;

}

}

**bool** Parser::Block(Tree\_Node\* Parser\_Tree) {

Check\_End\_Lexems();

Tree\_Node\* Current\_Node = Parser\_Tree;

**if** (Lexems[Lexem\_Index].Code != 402) {

Errors(Lexems[Lexem\_Index].Row, Lexems[Lexem\_Index].Column, ": Keyword \"BEGIN\" is missed\n");

}

Current\_Node->Down = Add\_New\_Tree\_Node(Lexems[Lexem\_Index].Code, Lexems[Lexem\_Index].Row, Lexems[Lexem\_Index].Column, Lexems[Lexem\_Index].Name, "", **true**);

Current\_Node = Current\_Node->Down;

Lexem\_Index++;

Check\_End\_Lexems();

Current\_Node->Right = Add\_New\_Tree\_Node(-1, Lexems[Lexem\_Index].Row, Lexems[Lexem\_Index].Column, "", "<statements-list>",**false**);

Current\_Node = Current\_Node->Right;

Statements\_list(Current\_Node);

Lexem\_Index++;

Check\_End\_Lexems();

**if** (Lexems[Lexem\_Index].Code != 403) {

Errors(Lexems[Lexem\_Index].Row, Lexems[Lexem\_Index].Column, ": Keyword \"END\" ismissed\n");

}

Current\_Node->Right = Add\_New\_Tree\_Node(Lexems[Lexem\_Index].Code, Lexems[Lexem\_Index].Row, Lexems[Lexem\_Index].Column, Lexems[Lexem\_Index].Name, "", **true**);

**return** **true**;

}

**bool** Parser::Statements\_list(Tree\_Node\* Parser\_Tree) {

Check\_End\_Lexems();

Tree\_Node\* Current\_Node = Parser\_Tree;

Tree\_Node\* empty\_Current\_Node = Current\_Node;

Current\_Node->Down = Add\_New\_Tree\_Node(-1, Lexems[Lexem\_Index].Row, Lexems[Lexem\_Index].Column, "", "<statement>",**false**);

Current\_Node = Current\_Node->Down;

**if** (Statement(Current\_Node)) {

Lexem\_Index++;

Check\_End\_Lexems();

Current\_Node->Right = Add\_New\_Tree\_Node(-1, Lexems[Lexem\_Index].Row, Lexems[Lexem\_Index].Column, "","<statements-list>", **false**);

Current\_Node = Current\_Node->Right;

Statements\_list(Current\_Node);

**return** **true**;

}

**else** {

empty\_Current\_Node->Down = Add\_New\_Tree\_Node(-1, Lexems[Lexem\_Index].Row, Lexems[Lexem\_Index].Column, "","<empty>", **false**);

empty\_Current\_Node = empty\_Current\_Node->Down;

Lexem\_Index--;

**return** **true**;

}

}

**bool** Parser::Statement(Tree\_Node\* Parser\_Tree) {

Check\_End\_Lexems();

Tree\_Node\* Current\_Node = Parser\_Tree;

**int** currentCode = Lexems[Lexem\_Index].Code;

**if** (currentCode == 405) { // while

//Check\_WHILE = true;

Check\_WHILE++;

Current\_Node->Down = Add\_New\_Tree\_Node(Lexems[Lexem\_Index].Code, Lexems[Lexem\_Index].Row, Lexems[Lexem\_Index].Column, Lexems[Lexem\_Index].Name, "", **true**);

Current\_Node = Current\_Node->Down;

Lexem\_Index++;

Check\_End\_Lexems();

Current\_Node->Right = Add\_New\_Tree\_Node(-1, Lexems[Lexem\_Index].Row, Lexems[Lexem\_Index].Column, "","<conditional-expression>", **false**);

Current\_Node = Current\_Node->Right;

Condition\_expression(Current\_Node);

Lexem\_Index++;

Check\_End\_Lexems();

**if** (Lexems[Lexem\_Index].Code != 406) {

Errors(Lexems[Lexem\_Index].Row, Lexems[Lexem\_Index].Column, ": Keyword \"DO\" is missed\n");

}

Current\_Node->Right = Add\_New\_Tree\_Node(Lexems[Lexem\_Index].Code, Lexems[Lexem\_Index].Row, Lexems[Lexem\_Index].Column, Lexems[Lexem\_Index].Name, "", **true**);

Current\_Node = Current\_Node->Right;

Lexem\_Index++;

Check\_End\_Lexems();

Current\_Node->Right = Add\_New\_Tree\_Node(-1, Lexems[Lexem\_Index].Row, Lexems[Lexem\_Index].Column, "","<statements-list>", **false**);

Current\_Node = Current\_Node->Right;

Statements\_list(Current\_Node);

Lexem\_Index++;

Check\_End\_Lexems();

**if** (Lexems[Lexem\_Index].Code != 407) {

Errors(Lexems[Lexem\_Index].Row, Lexems[Lexem\_Index].Column, ": Keyword \"ENDWHILE\" is missed\n");

}

//Check\_WHILE = false;

Check\_WHILE--;

Current\_Node->Right = Add\_New\_Tree\_Node(Lexems[Lexem\_Index].Code, Lexems[Lexem\_Index].Row, Lexems[Lexem\_Index].Column, Lexems[Lexem\_Index].Name, "", **true**);

Current\_Node = Current\_Node->Right;

Lexem\_Index++;

Check\_End\_Lexems();

**if** (Lexems[Lexem\_Index].Code != ';') {

Errors(Lexems[Lexem\_Index].Row, Lexems[Lexem\_Index].Column, ": \";\" is missed\n");

}

**else** {

Current\_Node->Right = Add\_New\_Tree\_Node(Lexems[Lexem\_Index].Code, Lexems[Lexem\_Index].Row, Lexems[Lexem\_Index].Column, Lexems[Lexem\_Index].Name, "", **true**);

Current\_Node = Current\_Node->Right;

**return** **true**;

}

}

**else** **if** (currentCode == 408) { // if

//Check\_IF = true;

Check\_IF++;

Current\_Node->Down = Add\_New\_Tree\_Node(-1, Lexems[Lexem\_Index].Row, Lexems[Lexem\_Index].Column, "", "<conditionstatement>",**false**);

Current\_Node = Current\_Node->Down;

Condition\_statement(Current\_Node);

Lexem\_Index++;

Check\_End\_Lexems();

**if** (Lexems[Lexem\_Index].Code != 404) {

Errors(Lexems[Lexem\_Index].Row, Lexems[Lexem\_Index].Column, ": Keyword \"ENDIF\" is missed\n");

}

//Check\_IF = false;

Check\_IF--;

Current\_Node->Right =Add\_New\_Tree\_Node(Lexems[Lexem\_Index].Code, Lexems[Lexem\_Index].Row, Lexems[Lexem\_Index].Column, Lexems[Lexem\_Index].Name, "", **true**);

Current\_Node = Current\_Node->Right;

Lexem\_Index++;

Check\_End\_Lexems();

Current\_Node->Right = Add\_New\_Tree\_Node(Lexems[Lexem\_Index].Code, Lexems[Lexem\_Index].Row, Lexems[Lexem\_Index].Column, Lexems[Lexem\_Index].Name, "", **true**);

Current\_Node = Current\_Node->Right;

**if** (Lexems[Lexem\_Index].Code != ';') {

Errors(Lexems[Lexem\_Index].Row, Lexems[Lexem\_Index].Column, ": \";\" is missed\n");

}

**else** {

**return** **true**;

}

}

**else** **if** (Lexems[Lexem\_Index].Code == 410) { // ELSE

**if** (Check\_IF == **false**) {

Errors(Lexems[Lexem\_Index].Row, Lexems[Lexem\_Index].Column, ": Incorrect syntax for statement block\n");

}

**return** **false**;

}

**else** **if** (Lexems[Lexem\_Index].Code == 407) { //ENDWHILE

**if** (Check\_WHILE == **false**) {

Errors(Lexems[Lexem\_Index].Row, Lexems[Lexem\_Index].Column, ": Incorrect syntax for statement block\n");

}

**return** **false**;

}

**else** **if** (Lexems[Lexem\_Index].Code == 404) { // ENDIF

**if** (Check\_IF == **false**) {

Errors(Lexems[Lexem\_Index].Row, Lexems[Lexem\_Index].Column, ": Incorrect syntax for statement block\n");

}

**return** **false**;

}

**else** **if** (Lexems[Lexem\_Index].Code == 403) { // END

**return** **false**;

}

**else** **if** (Lexems[Lexem\_Index].Code == ';') {

**if** (Check\_IF == **false** || Check\_WHILE == **false**) {

Errors(Lexems[Lexem\_Index].Row, Lexems[Lexem\_Index].Column, ": Incorrect syntax for statement block\n");

}

**return** **false**;

}

**else** **if** (Lexems[Lexem\_Index].Code != '.') {

Errors(Lexems[Lexem\_Index].Row, Lexems[Lexem\_Index].Column, ": Incorrect syntax for statement block\n");

}

**else** **return** **false**;

}

**bool** Parser::Condition\_statement(Tree\_Node\* Parser\_Tree) {

Check\_End\_Lexems();

Tree\_Node\* Current\_Node = Parser\_Tree;

Current\_Node->Down = Add\_New\_Tree\_Node(-1, Lexems[Lexem\_Index].Row, Lexems[Lexem\_Index].Column, "", "<incompletecondition-statement>", **false**);

Current\_Node = Current\_Node->Down;

Incomplete\_conditionstatement(Current\_Node);

Lexem\_Index++;

Current\_Node->Right = Add\_New\_Tree\_Node(-1, Lexems[Lexem\_Index].Row, Lexems[Lexem\_Index].Column, "", "<alternativepart>",**false**);

Current\_Node = Current\_Node->Right;

Alternative\_part(Current\_Node);

**return** **true**;

}

**bool** Parser::Incomplete\_conditionstatement(Tree\_Node\* Parser\_Tree)

{

Check\_End\_Lexems();

Tree\_Node\* Current\_Node = Parser\_Tree;

Current\_Node->Down = Add\_New\_Tree\_Node(Lexems[Lexem\_Index].Code, Lexems[Lexem\_Index].Row, Lexems[Lexem\_Index].Column, Lexems[Lexem\_Index].Name, "", **true**);

Current\_Node = Current\_Node->Down;

Lexem\_Index++;

Check\_End\_Lexems();

Current\_Node->Right = Add\_New\_Tree\_Node(-1, Lexems[Lexem\_Index].Row, Lexems[Lexem\_Index].Column, "","<conditional-expression>", **false**);

Current\_Node = Current\_Node->Right;

Condition\_expression(Current\_Node);

Lexem\_Index++;

Check\_End\_Lexems();

**if** (Lexems[Lexem\_Index].Code != 409) {

Errors(Lexems[Lexem\_Index].Row, Lexems[Lexem\_Index].Column, ": Keyword \"THEN\" is missed\n");

}

Current\_Node->Right = Add\_New\_Tree\_Node(Lexems[Lexem\_Index].Code, Lexems[Lexem\_Index].Row, Lexems[Lexem\_Index].Column, Lexems[Lexem\_Index].Name, "", **true**);

Current\_Node = Current\_Node->Right;

Lexem\_Index++;

Check\_End\_Lexems();

Current\_Node->Right = Add\_New\_Tree\_Node(-1, Lexems[Lexem\_Index].Row, Lexems[Lexem\_Index].Column, "","<statements-list>", **false**);

Current\_Node = Current\_Node->Right;

Statements\_list(Current\_Node);

**return** **true**;

}

**bool** Parser::Alternative\_part(Tree\_Node\* Parser\_Tree) {

Check\_End\_Lexems();

Tree\_Node\* Current\_Node = Parser\_Tree;

**if** (Lexems[Lexem\_Index].Code == 410) {

Current\_Node->Down =Add\_New\_Tree\_Node(Lexems[Lexem\_Index].Code, Lexems[Lexem\_Index].Row, Lexems[Lexem\_Index].Column, Lexems[Lexem\_Index].Name, "", **true**);

Current\_Node = Current\_Node->Down;

Lexem\_Index++;

Check\_End\_Lexems();

Current\_Node->Right = Add\_New\_Tree\_Node(-1, Lexems[Lexem\_Index].Row, Lexems[Lexem\_Index].Column, "","<statements-list>", **false**);

Current\_Node = Current\_Node->Right;

Statements\_list(Current\_Node);

**return** **true**;

}

**else** {

Current\_Node->Down = Add\_New\_Tree\_Node(-1, Lexems[Lexem\_Index].Row, Lexems[Lexem\_Index].Column, "", "<empty>",**false**);

Lexem\_Index--;

**return** **true**;

}

}

**bool** Parser::Condition\_expression(Tree\_Node\* Parser\_Tree) {

Check\_End\_Lexems();

Tree\_Node\* Current\_Node = Parser\_Tree;

Current\_Node->Down = Add\_New\_Tree\_Node(-1, Lexems[Lexem\_Index].Row, Lexems[Lexem\_Index].Column, "", "<expression>",**false**);

Current\_Node = Current\_Node->Down;

Expression(Current\_Node);

Lexem\_Index++;

Current\_Node->Right = Add\_New\_Tree\_Node(-1, Lexems[Lexem\_Index].Row, Lexems[Lexem\_Index].Column, "", "<comparison-operator>", **false**);

Current\_Node = Current\_Node->Right;

Comparison\_operator(Current\_Node);

Lexem\_Index++;

Current\_Node->Right = Add\_New\_Tree\_Node(-1, Lexems[Lexem\_Index].Row, Lexems[Lexem\_Index].Column, "", "<expression>", **false**);

Current\_Node = Current\_Node->Right;

Expression(Current\_Node);

**return** **true**;

}

**bool** Parser::Expression(Tree\_Node\* Parser\_Tree) {

Check\_End\_Lexems();

Tree\_Node\* Current\_Node = Parser\_Tree;

**int** lexemCode = Lexems[Lexem\_Index].Code;

**if** ((lexemCode >= CONST\_BASE && lexemCode < IDN\_BASE) || (lexemCode >= IDN\_BASE && lexemCode < ERR\_BASE)) {

string nodeType = (lexemCode >= CONST\_BASE && lexemCode < IDN\_BASE) ? "<unsigned-integer>" : "<variable-identifier>";

Current\_Node->Down = Add\_New\_Tree\_Node(-1, Lexems[Lexem\_Index].Row, Lexems[Lexem\_Index].Column, "", nodeType,**false**);

Current\_Node = Current\_Node->Down;

**if** (lexemCode >= IDN\_BASE && lexemCode < ERR\_BASE) {

Current\_Node->Down = Add\_New\_Tree\_Node(-1, Lexems[Lexem\_Index].Row, Lexems[Lexem\_Index].Column, "","<identifier>", **false**);

Current\_Node = Current\_Node->Down;

}

Current\_Node->Down = Add\_New\_Tree\_Node(lexemCode, Lexems[Lexem\_Index].Row, Lexems[Lexem\_Index].Column, Lexems[Lexem\_Index].Name, "", **true**);

Current\_Node = Current\_Node->Down;

**return** **true**;

}

Errors(Lexems[Lexem\_Index].Row, Lexems[Lexem\_Index].Column, ": Variable identifier or unsigned integer ismissed\n");

}

**bool** Parser::Comparison\_operator(Tree\_Node\* Parser\_Tree) {

Check\_End\_Lexems();

Tree\_Node\* Current\_Node = Parser\_Tree;

**if** ((Lexems[Lexem\_Index].Code == '<') || (Lexems[Lexem\_Index].Code == '=') || (Lexems[Lexem\_Index].Code == '>') || (Lexems[Lexem\_Index].Code >= DM\_BASE && Lexems[Lexem\_Index].Code < KW\_BASE)) {

Current\_Node->Down = Add\_New\_Tree\_Node(Lexems[Lexem\_Index].Code, Lexems[Lexem\_Index].Row, Lexems[Lexem\_Index].Column, Lexems[Lexem\_Index].Name, "", **true**);

Current\_Node = Current\_Node->Down;

**return** **true**;

}

Errors(Lexems[Lexem\_Index].Row, Lexems[Lexem\_Index].Column, ": Comparison operator is missed\n");

}

**bool** Parser::Identifier(Tree\_Node\* Parser\_Tree) {

Check\_End\_Lexems();

Tree\_Node\* Current\_Node = Parser\_Tree;

**if** ((Lexems[Lexem\_Index].Code >= IDN\_BASE) && (Lexems[Lexem\_Index].Code < ERR\_BASE)) {

Current\_Node->Down = Add\_New\_Tree\_Node(-1, Lexems[Lexem\_Index].Row, Lexems[Lexem\_Index].Column, "", "<identifier>", **false**);

Current\_Node = Current\_Node->Down;

Current\_Node->Down = Add\_New\_Tree\_Node(Lexems[Lexem\_Index].Code, Lexems[Lexem\_Index].Row, Lexems[Lexem\_Index].Column, Lexems[Lexem\_Index].Name, "", **true**);

**return** **true**;

}

Errors(Lexems[Lexem\_Index].Row, Lexems[Lexem\_Index].Column, ": Identifier is missed\n");

}

Tree\_Node\* Parser::Add\_New\_Tree\_Node(**int** code,**int** row, **int** column, string name, string NonTerminal\_name, **bool** Is\_Terminal) {

Tree\_Node\* newTree\_Node = **new** Tree\_Node;

newTree\_Node->Code = code;

newTree\_Node->Down = NULL;

newTree\_Node->Row = row;

newTree\_Node->Column = column;

newTree\_Node->Is\_Terminal = Is\_Terminal;

newTree\_Node->Name = name;

newTree\_Node->Right = NULL;

newTree\_Node->NonTerminal\_name = **new** **char**[255];

newTree\_Node->NonTerminal\_name = NonTerminal\_name;

**return** newTree\_Node;

}

**void** Parser::Errors(**int** Row = -1, **int** Column = -1, string message = "") {

**if** (Row == -1 && Column == -1) {

ErrorString = "Syntactic error: " + message;

}

**else** {

ErrorString = "Line " + to\_string(Row) + " Column " + to\_string(Column) + message;

}

cout << ErrorString << endl;

exit(1);

}

**void** Parser::Check\_End\_Lexems() {

**if** (Lexem\_Index >= Lexems.size()) {

Errors(-1, -1 ,"Unexpected end of the file!\n");

}

}

**bool** Parser::Check\_After\_File() {

**return** Lexem\_Index < Lexems.size();

}

**void** Parser::Check\_Dot\_End() {

**if** (Lexem\_Index >= Lexems.size()) {

Errors(-1, -1, ": \".\" is missed\n");

}

}

**Generator.cpp**

#include "Generator.h"

**void** Code\_Generator::Start\_code\_generation(string File) {

program(Parse\_tree);

Code\_Generation\_Listing(File);

}

Code\_Generator::Code\_Generator(Parser& object): Error\_String(""), Code\_line\_string(""), Label\_Counter(0), Label(0), Parse\_tree(object.Parse\_tree)

{

}

**Generator.h**

#**pragma once**

#include <string>

#include <vector>

#include <iostream>

#include <iomanip>

#include <fstream>

#include "Lexer.h"

#include "Parser.h"

**class** Code\_Generator {

**private**:

Tree\_Node\* Parse\_tree;

string Code\_line\_string;

string Error\_String;

**int** Label;

**int** Label\_Counter;

**int** temp\_label;

string Name\_of\_Program;

**void** Code\_Generation\_Listing(string File);

**void** program(Tree\_Node\* Parser\_Tree);

**void** statements\_list(Tree\_Node\* Parser\_Tree);

**void** statement(Tree\_Node\* Parser\_Tree);

**void** condition\_statement(Tree\_Node\* Parser\_Tree);

**void** conditional\_expression(Tree\_Node\* Parser\_Tree);

**void** incomplete\_condition\_statement(Tree\_Node\* Parser\_Tree);

**void** alternative\_part(Tree\_Node\* Parser\_Tree);

**void** comparison\_operator(Tree\_Node\* Parser\_Tree);

**void** expression(Tree\_Node\* Parser\_Tree);

**public**:

Code\_Generator(Parser& object);

**void** Start\_code\_generation(string File);

};

**Generating.cpp**

#include "Generator.h"

**void** Code\_Generator::program(Tree\_Node\* Parser\_Tree)

{

Tree\_Node\* Current\_Node = Parser\_Tree;

Name\_of\_Program = Current\_Node->Down->Down->Right->Down->Down->Name;

Current\_Node = Current\_Node->Down->Down->Right->Right->Right->Down->Right;

**if** (Current\_Node->NonTerminal\_name == "<statements-list>") {

Code\_line\_string.append("code SEGMENT\n\tASSUME cs:code\t\n" + Name\_of\_Program + ":\n");

statements\_list(Current\_Node);

Code\_line\_string.append("mov ah, 4ch\nint 21h\ncode ends\nend " + Name\_of\_Program);

**return**;

}

Error\_String = "Code Generation Error: Row " + to\_string(Current\_Node->Row) + " column " + to\_string(Current\_Node->Column) + "\n";

**return**;

}

**void** Code\_Generator::statements\_list(Tree\_Node\* Parser\_Tree) {

Tree\_Node\* Current\_Node = Parser\_Tree;

Current\_Node = Current\_Node->Down;

**if** (Current\_Node ->NonTerminal\_name == "<empty>") {

Code\_line\_string.append("\tnop\n");

}

**if** (Current\_Node->NonTerminal\_name == "<statement>") {

statement(Current\_Node);

Current\_Node = Current\_Node->Right;

**if** (Current\_Node->NonTerminal\_name == "<statements-list>") {

statements\_list(Current\_Node);

}

}

}

**void** Code\_Generator::statement(Tree\_Node\* Parser\_Tree) {

Tree\_Node\* Current\_Node = Parser\_Tree;

Current\_Node = Current\_Node->Down;

**if** (Current\_Node->Code == 405) {

Label = Label\_Counter;

Code\_line\_string.append("?L" + to\_string(Label) + ":\n");

**int** while\_Label = Label;

Label++;

Label\_Counter++;

Current\_Node = Current\_Node->Right;

**if** (Current\_Node->NonTerminal\_name == "<conditional-expression>") {

conditional\_expression(Current\_Node);

Label++;

Label\_Counter++;

Current\_Node = Current\_Node->Right->Right;

**if** (Current\_Node->NonTerminal\_name == "<statements-list>") {

statements\_list(Current\_Node);

Current\_Node = Current\_Node->Right;

**if** (Current\_Node->Code == 407) {

Label = while\_Label;

Code\_line\_string.append("\tjmp ?L" + to\_string(Label) + "\n");

Label++;

Code\_line\_string.append("?L" + to\_string(Label) + ":\tnop\n");

}

}

}

}

**if** (Current\_Node->NonTerminal\_name == "<conditionstatement>") {

condition\_statement(Current\_Node);

}

}

**void** Code\_Generator::condition\_statement(Tree\_Node\* Parser\_Tree) {

Tree\_Node\* Current\_Node = Parser\_Tree;

Current\_Node = Current\_Node->Down;

Label = Label\_Counter;

**int** if\_Label = Label;

**if** (Current\_Node->NonTerminal\_name == "<incompletecondition-statement>") {

incomplete\_condition\_statement(Current\_Node);

Label = if\_Label;

Code\_line\_string.append("?L" + to\_string(Label) + ":\tnop\n");

Label++;

Current\_Node = Current\_Node->Right;

**if** (Current\_Node->NonTerminal\_name == "<alternativepart>") {

alternative\_part(Current\_Node);

Code\_line\_string.append("?L" + to\_string(Label) + ":\tnop\n");

}

}

}

**void** Code\_Generator::incomplete\_condition\_statement(Tree\_Node\* Parser\_Tree) {

Tree\_Node\* Current\_Node = Parser\_Tree;

Current\_Node = Current\_Node->Down->Right;

**if** (Current\_Node->NonTerminal\_name == "<conditional-expression>") {

conditional\_expression(Current\_Node);

Label\_Counter++;

Label++;

**int** inc\_temp\_Label = Label;

Label++;

Label\_Counter++;

Current\_Node = Current\_Node->Right->Right;

**if** (Current\_Node->NonTerminal\_name == "<statements-list>") {

statements\_list(Current\_Node);

Label = inc\_temp\_Label;

Code\_line\_string.append("\tjmp ?L" + to\_string(Label) + "\n");

}

}

}

**void** Code\_Generator::alternative\_part(Tree\_Node\* Parser\_Tree) {

Tree\_Node\* Current\_Node = Parser\_Tree;

Current\_Node = Current\_Node->Down;

**if** (Current\_Node->NonTerminal\_name == "<empty>") {

Code\_line\_string.append("\tnop\n");

}

**else** **if**( Current\_Node->Code == 410) {

**int** ap\_temp\_label = Label;

Label = Label\_Counter;

Current\_Node = Current\_Node->Right;

**if** (Current\_Node->NonTerminal\_name == "<statements-list>"){

statements\_list(Current\_Node);

Label = ap\_temp\_label;

}

}

}

**void** Code\_Generator::conditional\_expression(Tree\_Node\* Parser\_Tree){

Tree\_Node\* Current\_Node = Parser\_Tree;

Current\_Node = Current\_Node->Down;

**if** (Current\_Node->NonTerminal\_name == "<expression>" && Current\_Node->Right->Right->NonTerminal\_name == "<expression>") {

expression(Current\_Node);

Current\_Node = Current\_Node->Right;

**if** (Current\_Node->NonTerminal\_name == "<comparison-operator>") {

comparison\_operator(Current\_Node);

}

}

}

**void** Code\_Generator::comparison\_operator(Tree\_Node\* Parser\_Tree) {

Tree\_Node\* Current\_Node = Parser\_Tree;

Current\_Node = Current\_Node->Down;

**int** find = 0;

**if** (Current\_Node->Name == "<")

{

Code\_line\_string.append("\tjge ?L" + to\_string(Label) + "\n");

find = 1;

}

**if** (Current\_Node->Name == "=")

{

Code\_line\_string.append("\tlne ?L" + to\_string(Label) + "\n");

find = 1;

}

**if** (Current\_Node->Name == ">")

{

Code\_line\_string.append("\tjle ?L" + to\_string(Label) + "\n");

find = 1;

}

**if** (Current\_Node->Name == "<=")

{

Code\_line\_string.append("\tjg ?L" + to\_string(Label) + "\n");

find = 1;

}

**if** (Current\_Node->Name == "<>")

{

Code\_line\_string.append("\tje ?L" + to\_string(Label) + "\n");

find = 1;

}

**if** (Current\_Node->Name == ">=")

{

Code\_line\_string.append("\tjl ?L" + to\_string(Label) + "\n");

find = 1;

}

**else** **if** (find == 0) {

Error\_String = "Error generate comparation operator : Row " + to\_string(Current\_Node->Row) + " column " + to\_string(Current\_Node->Column) + "\n";

exit(-1);

}

}

**void** Code\_Generator::expression(Tree\_Node\* Parser\_Tree) {

Tree\_Node\* Expression\_Node\_1 = Parser\_Tree->Down;

Tree\_Node\* Expression\_Node\_2 = Parser\_Tree->Right->Right->Down;

**if** (Expression\_Node\_1->NonTerminal\_name == "<variable-identifier>" && Expression\_Node\_2->NonTerminal\_name == "<variable-identifier>") {

**if** (Expression\_Node\_1->Down->Down->Name != Name\_of\_Program && Expression\_Node\_2->Down->Down->Name != Name\_of\_Program) {

Code\_line\_string.append("\tmov ax, " + Expression\_Node\_1->Down->Down->Name + "\n");

Code\_line\_string.append("\tmov bx, " + Expression\_Node\_2->Down->Down->Name + "\n");

Code\_line\_string.append("\tcmp ax, bx\n");

}

**else** {

Code\_line\_string.append("\terror: <variable-identifier> can`t equal name of program \n");

Error\_String = "Error generate comparation operator : Row " + to\_string(Expression\_Node\_1->Row) + " column " + to\_string(Expression\_Node\_1->Column) + "\n";

**return**;

}

}

**else** **if**(Expression\_Node\_1->NonTerminal\_name == "<unsigned-integer>" && Expression\_Node\_2->NonTerminal\_name == "<unsigned-integer>") {

Code\_line\_string.append("\tmov ax, " + Expression\_Node\_1->Down->Name + "\n");

Code\_line\_string.append("\tmov bx, " + Expression\_Node\_2->Down->Name + "\n");

Code\_line\_string.append("\tcmp ax, bx\n");

}

**else** **if** (Expression\_Node\_1->NonTerminal\_name == "<variable-identifier>" && Expression\_Node\_2->NonTerminal\_name == "<unsigned-integer>") {

**if** (Expression\_Node\_1->Down->Down->Name != Name\_of\_Program) {

Code\_line\_string.append("\tmov ax, " + Expression\_Node\_1->Down->Down->Name + "\n");

Code\_line\_string.append("\tmov bx, " + Expression\_Node\_2->Down->Name + "\n");

Code\_line\_string.append("\tcmp ax, bx\n");

}

**else** {

Code\_line\_string.append("\terror: <variable-identifier> can`t equal name of program \n");

Error\_String = "Error generate comparation operator : Row " + to\_string(Expression\_Node\_1->Row) + " column " + to\_string(Expression\_Node\_1->Column) + "\n";

**return**;

}

}

**else** **if** (Expression\_Node\_1->NonTerminal\_name == "<unsigned-integer>" && Expression\_Node\_2->NonTerminal\_name == "<variable-identifier>") {

**if** (Expression\_Node\_2->Down->Down->Name != Name\_of\_Program) {

Code\_line\_string.append("\tmov ax, " + Expression\_Node\_1->Down->Name + "\n");

Code\_line\_string.append("\tmov bx, " + Expression\_Node\_2->Down->Down->Name + "\n");

Code\_line\_string.append("\tcmp ax, bx\n");

}

**else** {

Code\_line\_string.append("\terror: <variable-identifier> can`t equal name of program \n");

Error\_String = "Error generate comparation operator : Row " + to\_string(Expression\_Node\_2->Row) + " column " + to\_string(Expression\_Node\_2->Column) + "\n";

**return**;

}

}

}

**OUTPUT.cpp**

#include "Lexer.h"

#include "Parser.h"

#include "Generator.h"

**void** Lexer::Listing\_Lexer(string File)

{

File = "Lexer\_Listing\_" + File;

ofstream File\_Output(File);

File\_Output << setw(10) << left << "Row" << setw(10) << left << "Column" << setw(10) << left << "Code" << setw(20) << left << "Lexem" << endl << endl;

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

{

**if** (Lexems[i].Code < ERR\_BASE)

File\_Output << setw(10) << left << Lexems[i].Row << setw(10) << left << Lexems[i].Column << setw(10) << left << Lexems[i].Code << setw(20) << left << Lexems[i].Name << endl;

}

File\_Output << endl << endl;

**if** (ERROR\_FLAG == 1)

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

**for** (**int** j = ERR\_BASE; j < Lex\_Errors\_Counter; j++)

**if** (Lexems[i].Code == j)

File\_Output << "Lexer: Error " << Lexems[i].Code << "(Row " << Lexems[i].Row << ", Column " << Lexems[i].Column << "): ImLexem\_Indexible characters combination: " << Lexems[i].Name << endl;

cout << "Lexical analysis completed successfuly" << endl << endl;

File\_Output.close();

}

**void** Parser::Tree\_Listing(string File) {

TreeString = "";

File = "Syntax\_Listing\_" + File;

ofstream File\_Output(File);

**if** (!ErrorString.empty())

File\_Output << ErrorString << endl;

**else** {

Write\_Tree(Parse\_tree, "", File\_Output);

File\_Output << TreeString << endl;

}

}

**void** Parser::Write\_Tree(Tree\_Node\* Root, **const** string space, ofstream& File\_Output) {

**while** (Root != **nullptr**) {

**if** (Root->Is\_Terminal)

File\_Output << space << to\_string(Root->Code) << " " << Root->Name << "\n";

**else**

File\_Output << space << Root->NonTerminal\_name << "\n";

**if** (Root->Down != **nullptr**) {

string newSpace = space + " ";

Write\_Tree(Root->Down, newSpace, File\_Output);

}

Root = Root->Right;

}

}

**void** Code\_Generator::Code\_Generation\_Listing(string File)

{

File = "Code\_Generation\_Listing\_" + File;

ofstream File\_Output(File);

**if** (!Error\_String.empty()) {

File\_Output << Error\_String << endl;

cout << "Error while generating code" << endl << endl;

}

**else** {

cout << "Code generation completed successfuly" << endl << endl;

File\_Output << Code\_line\_string << endl << endl;

}

File\_Output.close();

}

**Main.cpp**

#include "Lexer.h"

#include "parser.h"

#include "Generator.h"

**using** **namespace** std;

**int** main()

{

string filename = "test\_true4.sig";

Lexer Lab1;

Lab1.Start\_lex\_analysis(filename);

Parser RGR(Lab1);

RGR.Start\_syntax\_analysis(filename);

Code\_Generator Lab2(RGR);

Lab2.Start\_code\_generation(filename);

**return** 0;

}

**Тестування програми**

1. **True-тест**

PROGRAM LAB2OPT;

BEGIN

(\*TES ()\*\*T FOR

\*\*LAB)\*)

WHILE VAR1 = (\*ABO B \*A\*) VAR4 DO

IF 24 <= 56

THEN

ELSE

WHILE VAR1 = VAR4 DO ENDWHILE;

ENDIF;

ENDWHILE;

END

code SEGMENT

ASSUME cs:code

LAB2OPT:

?L0:

mov ax, VAR1

mov bx, VAR4

cmp ax, bx

lne ?L1

mov ax, 24

mov bx, 56

cmp ax, bx

jg ?L2

nop

jmp ?L3

?L2: nop

?L4:

mov ax, VAR1

mov bx, VAR4

cmp ax, bx

lne ?L5

nop

jmp ?L4

?L5: nop

nop

?L3: nop

nop

jmp ?L0

?L1: nop

nop

mov ah, 4ch

int 21h

code ends

end LAB2OPT

1. **True-тест**

PROGRAM LAB2;

BEGIN

IF 10 > 2

THEN

WHILE VAR1 <> VAR2 DO ENDWHILE;

ELSE

IF 2 = LAB22

THEN

ELSE

WHILE VAR1 <= VAR2

DO

IF 10 >= 2 THEN ENDIF;

WHILE VAR16644 <> VAR6634342 DO ENDWHILE;

ENDWHILE;

ENDIF;

WHILE 4 = 56

DO

WHILE VAR166 <> VAR662 DO ENDWHILE;

ENDWHILE;

ENDIF;

WHILE VAR1 <= VAR2

DO

IF 10 >= 2 THEN ENDIF;

WHILE VAR16644 <> VAR6634342 DO ENDWHILE;

ENDWHILE;

IF 10 > 2 THEN ENDIF;

END.

code SEGMENT

ASSUME cs:code

LAB2:

mov ax, 10

mov bx, 2

cmp ax, bx

jle ?L0

?L2:

mov ax, VAR1

mov bx, VAR2

cmp ax, bx

je ?L3

nop

jmp ?L2

?L3: nop

nop

jmp ?L1

?L0: nop

mov ax, 2

mov bx, LAB22

cmp ax, bx

lne ?L4

nop

jmp ?L5

?L4: nop

?L6:

mov ax, VAR1

mov bx, VAR2

cmp ax, bx

jg ?L7

mov ax, 10

mov bx, 2

jl ?L8

nop

jmp ?L9

?L8: nop

nop

?L9: nop

?L10:

mov ax, VAR16644

mov bx, VAR6634342

cmp ax, bx

je ?L11

nop

jmp ?L10

?L11: nop

nop

jmp ?L6

?L7: nop

nop

?L5: nop

?L12:

mov ax, 4

mov bx, 56

cmp ax, bx

lne ?L13

?L14:

mov ax, VAR166

mov bx, VAR662

cmp ax, bx

je ?L15

nop

jmp ?L14

?L15: nop

nop

jmp ?L12

?L13: nop

nop

?L1: nop

?L16:

mov ax, VAR1

mov bx, VAR2

cmp ax, bx

jg ?L17

mov ax, 10

mov bx, 2

cmp ax, bx

jl ?L18

nop

jmp ?L19

?L18: nop

nop

?L19: nop

?L20:

mov ax, VAR16644

mov bx, VAR6634342

cmp ax, bx

je ?L21

nop

jmp ?L20

?L21: nop

nop

jmp ?L16

?L17: nop

mov ax, 10

mov bx, 2

cmp ax, bx

jle ?L22

nop

jmp ?L23

?L22: nop

nop

?L23: nop

nop

mov ah, 4ch

int 21h

code ends

end LAB2

1. **False-тест**

PROGRAM OPT2;

BEGIN

(\*TES ()\*\*T FOR

\*\*LAB)\*)

WHILE VAR1 = VAR2 DO

IF 24 <= OPT2

THEN

WHILE VAR1 = VAR2 DO ENDWHILE;

ENDIF;

ENDWHILE;

END.

**Lexical analysis completed successfuly**

**Syntax analysis completed successfully**

**Error while generating code**

code SEGMENT

ASSUME cs:code

OPT2:

?L0:

mov ax, VAR1

mov bx, VAR2

cmp ax, bx

lne ?L1

error: <variable-identifier> can`t equal name of program

jg ?L2

?L4:

mov ax, VAR1

mov bx, VAR2

cmp ax, bx

lne ?L5

nop

jmp ?L4

?L5: nop

nop

jmp ?L3

?L2: nop

nop

?L3: nop

nop

jmp ?L0

?L1: nop

nop

mov ah, 4ch

int 21h

code ends

end OPT2

Error generate comparation operator : Row 7 column 22

1. **False-тест**

PROGRAM OPT2;

BEGIN

WHILE OPT2 = OPT2 DO

ENDWHILE;

END.

**Lexical analysis completed successfuly**

**Syntax analysis completed successfully**

**Error while generating code**

code SEGMENT

ASSUME cs:code

OPT2:

?L0:

error: <variable-identifier> can`t equal name of program

lne ?L1

nop

jmp ?L0

?L1: nop

nop

mov ah, 4ch

int 21h

code ends

end OPT2

Error generate comparation operator : Row 3 column 16