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>.
3. <block> --> BEGIN <statements-list> END
4. <statements-list> --> <statement> <statements-list>
   |
   <empty>
5. <statement> --> <condition-statement> ENDIF ; | WHILE
   <conditional-expression> DO
   <statements-list> ENDWHILE ;
6. <condition-statement> --> <incomplete-condition-statement><alternative-part>
7. <incomplete-condition-statement> --> IF
   <conditional-expression> THEN <statements- list>
8. <alternative-part> --> ELSE<statements-list> |
   <empty>
9. <conditional-expression> -->
   <expression><comparison-operator>
   <expression>
10. <comparison-operator> --> <|
    <= |
    = |
    <> |
    >= |
    >
11. <expression> --> <variable-identifier> |
    <unsigned-integer>
12. <variable-identifier> --> <identifier>
13. <procedure-identifier> --> <identifier>
14. <identifier> --> <letter><string>
15. <string> --> <letter><string> |
    <digit><string> |
    <empty>
16. <unsigned-integer> --> <digit><digits-string>
17. <digits-string> --> <digit><digits-string> |
    <empty>
18. <digit> --> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
19. <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

```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

```cpp
#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
```

2. 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
```

3. 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

4. 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