Министерство образования Республики Беларусь
Учреждение образования
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

КАФЕДРА ИНФОРМАТИКИ

Отчёт по лабораторной работе № 3

По теме «Синтаксический анализатор»

Выполнил:
студент гр. 053502
Герчик А. В.

Проверил:
Гриценко Н. Ю.

Минск 2023

Содержание

1. Цель работы

Освоение работы с существующими синтаксическими анализаторами. Разработать свой собственный синтаксический анализатор, выбранного подмножества языка программирования. Построить синтаксическое дерево. Определить минимум 4 возможных синтаксических ошибки и показать их корректное выявление.

2. Результаты

2.1 АСД для тестовой программы test.cpp

На рисунках ниже представлен код программы test.cpp, на языке программирования C++. В програму входят простые математические операции (см. рисунок 2.1).

```
int main()
{
    float a, b, c, Y = 0.0;
    int n;
    cout << "Enter a : ";
    cin >> a;
    cout << "Enter b : ";
    cin >> b;
    cout << "Enter c : ";
    cin >> c;
    cout << "Enter N : ";
    cin >> n;
    switch (n)
    {
    case 2:
        Y = (b * c);
        break;
    case 56:
        Y = (b * c);
        break;
    case 7:
        Y = a + c;
        break;
    case 3:
        Y = a - (b * c);
        break;
    default:
        Y = a + (b * 3);
        break;
    }
    cout << "Y is : " << Y << endl;
    return 0;
}
```

Рисунок 2.1 - Код программы test.cpp

Результат выполнения программы синтаксического анализатора, а именно построенное абстрактное синтаксическое дерево изображено ниже (см. рисунок 2.2, рисунок 2.3, рисунок 2.4, рисунок 2.5)

Рисунок 2.2 - Первая часть АСД для test.cpp

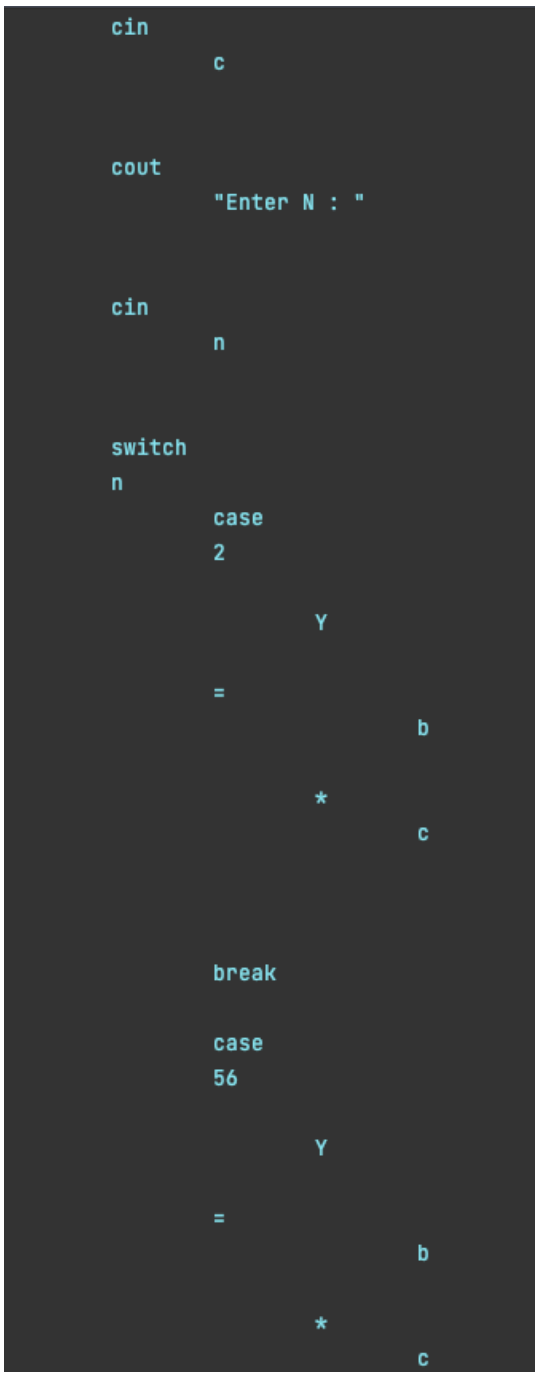

Рисунок 2.3 - Вторая часть АСД для test.cpp

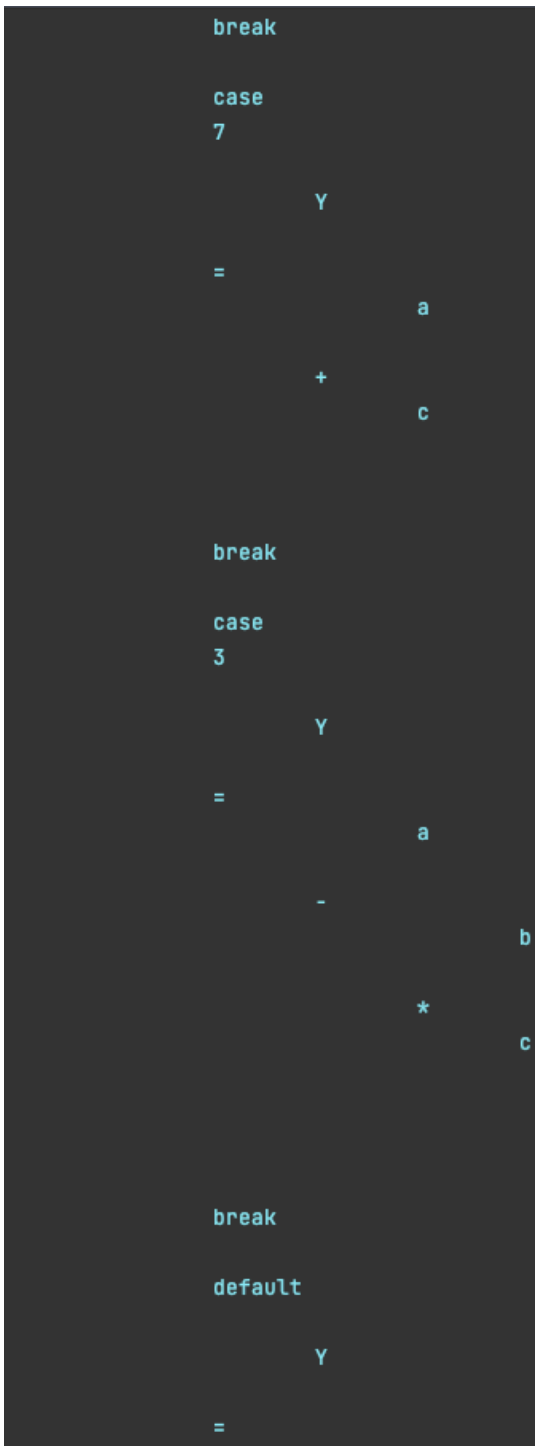

Рисунок 2.4 - Третья часть АСД для test.cpp

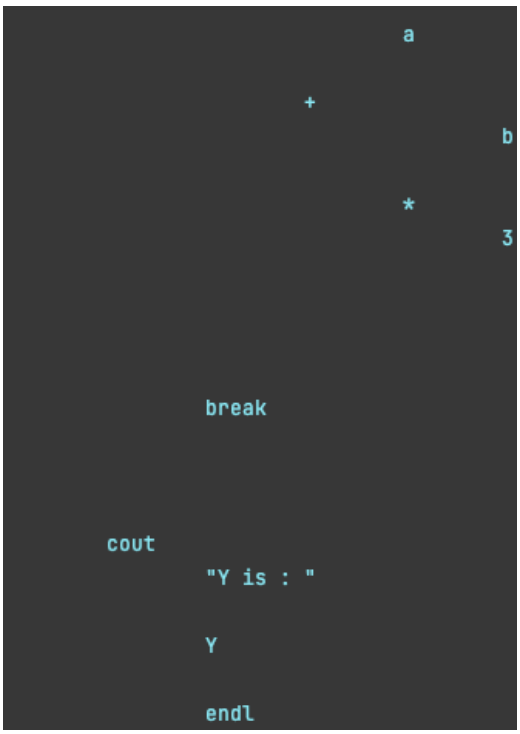

Рисунок 2.5 - Четвертая часть АСД для test.cpp

2.2 Примеры синтаксических ошибок

1) Пример с пропущенной запятой в объявлении переменных (см. рисунок 2.6, рисунок 2.7).

```
int main()
{
    float a b, c, Y = 0.0;
    int n;
```

Рисунок 2.6 - Пример кода с синтаксической ошибкой

```
"/Users/lnxd/Desktop/BSUIR/SIXTH TERM/MTran/LABS/DONE/LAB3/SyntaxAnalyzer/bin/Debug/net6.0/SyntaxAnalyzer"
Unhandled exception. System.Exception: after token 'a' needs token ';'
   at SyntaxAnalyzer.Parser.Require(List`1 tokenTypes) in /Users/lnxd/Desktop/BSUIR/SIXTH TERM/MTran/LABS/DONE/LAB3/SyntaxAnalyzer/Parser.cs:line 41
   at SyntaxAnalyzer.Parser.ParseExpression() in /Users/lnxd/Desktop/BSUIR/SIXTH TERM/MTran/LABS/DONE/LAB3/SyntaxAnalyzer/Parser.cs:line 412
   at SyntaxAnalyzer.Parser.ParseCode() in /Users/lnxd/Desktop/BSUIR/SIXTH TERM/MTran/LABS/DONE/LAB3/SyntaxAnalyzer/Parser.cs:line 588
   at SyntaxAnalyzer.Parser.ParseExpression() in /Users/lnxd/Desktop/BSUIR/SIXTH TERM/MTran/LABS/DONE/LAB3/SyntaxAnalyzer/Parser.cs:line 348
   at SyntaxAnalyzer.Parser.ParseCode() in /Users/lnxd/Desktop/BSUIR/SIXTH TERM/MTran/LABS/DONE/LAB3/SyntaxAnalyzer/Parser.cs:line 588
   at SyntaxAnalyzer.Program.Main() in /Users/lnxd/Desktop/BSUIR/SIXTH TERM/MTran/LABS/DONE/LAB3/SyntaxAnalyzer/Program.cs:line 188

Process finished with exit code 134.
```

Рисунок 2.7 - Реакция синтаксического анализатора на ошибку

2) Пример неправильного использования ключевых слов (см. рисунок 2.8, рисунок 2.9).

```
switch if (n)
{
case 2:
    Y = (b * c);
    break;
case 56:
    Y = (b * c);
    break;
case 7:
    Y = a + c;
    break;
case 3:
    Y = a - (b * c);
    break;
default:
    Y = a + (b * 3);
    break;
}
cout << "Y is : " << Y << endl;
```

Рисунок 2.8 - Пример кода с синтаксической ошибкой

```
"/Users/lnxd/Desktop/BSUIR/SIXTH TERM/MTran/LABS/DONE/LAB3/SyntaxAnalyzer/bin/Debug/net6.0/SyntaxAnalyzer"
Unhandled exception. System.Exception: after token 'switch' needs token '('
   at SyntaxAnalyzer.Parser.Require(List`1 tokenTypes) in /Users/lnxd/Desktop/BSUIR/SIXTH TERM/MTran/LABS/DONE/LAB3/SyntaxAnalyzer/Parser.cs:line 41
   at SyntaxAnalyzer.Parser.ParseExpression() in /Users/lnxd/Desktop/BSUIR/SIXTH TERM/MTran/LABS/DONE/LAB3/SyntaxAnalyzer/Parser.cs:line 518
   at SyntaxAnalyzer.Parser.ParseCode() in /Users/lnxd/Desktop/BSUIR/SIXTH TERM/MTran/LABS/DONE/LAB3/SyntaxAnalyzer/Parser.cs:line 588
   at SyntaxAnalyzer.Parser.ParseExpression() in /Users/lnxd/Desktop/BSUIR/SIXTH TERM/MTran/LABS/DONE/LAB3/SyntaxAnalyzer/Parser.cs:line 348
   at SyntaxAnalyzer.Parser.ParseCode() in /Users/lnxd/Desktop/BSUIR/SIXTH TERM/MTran/LABS/DONE/LAB3/SyntaxAnalyzer/Parser.cs:line 588
   at SyntaxAnalyzer.Program.Main() in /Users/lnxd/Desktop/BSUIR/SIXTH TERM/MTran/LABS/DONE/LAB3/SyntaxAnalyzer/Program.cs:line 188

Process finished with exit code 134.
```

Рисунок 2.9 - Реакция синтаксического анализатора на ошибку

3) Пример неправильного объявления функции (см. рисунок 2.10, рисунок 2.11).

```
int main(123)
{
    float a, b, c, Y = 0.0;
    int n;
    cout << "Enter a : ";
    cin >> a;
    cout << "Enter b : ";
    cin >> b;
    cout << "Enter c : ";
    cin >> c;
    cout << "Enter N : ";
    cin >> n;
```

Рисунок 2.10 - Пример кода с синтаксической ошибкой

```
"/Users/lnxd/Desktop/BSUIR/SIXTH TERM/MTran/LABS/DONE/LAB3/SyntaxAnalyzer/bin/Debug/net6.0/SyntaxAnalyzer"
Unhandled exception. System.Exception: After token '(' need token var type or ')'
   at SyntaxAnalyzer.Parser.ParseFunctionDefinition() in /Users/lnxd/Desktop/BSUIR/SIXTH TERM/MTran/LABS/DONE/LAB3/SyntaxAnalyzer/Parser.cs:line 203
   at SyntaxAnalyzer.Parser.ParseExpression() in /Users/lnxd/Desktop/BSUIR/SIXTH TERM/MTran/LABS/DONE/LAB3/SyntaxAnalyzer/Parser.cs:line 345
   at SyntaxAnalyzer.Parser.ParseCode() in /Users/lnxd/Desktop/BSUIR/SIXTH TERM/MTran/LABS/DONE/LAB3/SyntaxAnalyzer/Parser.cs:line 588
   at SyntaxAnalyzer.Program.Main() in /Users/lnxd/Desktop/BSUIR/SIXTH TERM/MTran/LABS/DONE/LAB3/SyntaxAnalyzer/Program.cs:line 188

Process finished with exit code 134.
```

Рисунок 2.11 - Реакция синтаксического анализатора на ошибку

4) Пример недопустимого присвоения (см. рисунок 2.12, рисунок 2.13).

```
int main()
{
    float a, b, c, Y = float;
    int n;
    cout << "Enter a : ";
    cin >> a;
```

Рисунок 2.12 - Пример кода с синтаксической ошибкой

```
"/Users/lnxd/Desktop/BSUIR/SIXTH TERM/MTran/LABS/DONE/LAB3/SyntaxAnalyzer/bin/Debug/net6.0/SyntaxAnalyzer"
Unhandled exception. System.Exception: After token '=' needs var or literal token
   at SyntaxAnalyzer.Parser.ParseVariableOrLiteral() in /Users/lnxd/Desktop/BSUIR/SIXTH TERM/MTran/LABS/DONE/LAB3/SyntaxAnalyzer/Parser.cs:line 108
   at SyntaxAnalyzer.Parser.ParseParentheses() in /Users/lnxd/Desktop/BSUIR/SIXTH TERM/MTran/LABS/DONE/LAB3/SyntaxAnalyzer/Parser.cs:line 121
   at SyntaxAnalyzer.Parser.ParseFormula() in /Users/lnxd/Desktop/BSUIR/SIXTH TERM/MTran/LABS/DONE/LAB3/SyntaxAnalyzer/Parser.cs:line 127
   at SyntaxAnalyzer.Parser.ParseExpression() in /Users/lnxd/Desktop/BSUIR/SIXTH TERM/MTran/LABS/DONE/LAB3/SyntaxAnalyzer/Parser.cs:line 406
   at SyntaxAnalyzer.Parser.ParseCode() in /Users/lnxd/Desktop/BSUIR/SIXTH TERM/MTran/LABS/DONE/LAB3/SyntaxAnalyzer/Parser.cs:line 588
   at SyntaxAnalyzer.Parser.ParseExpression() in /Users/lnxd/Desktop/BSUIR/SIXTH TERM/MTran/LABS/DONE/LAB3/SyntaxAnalyzer/Parser.cs:line 348
   at SyntaxAnalyzer.Parser.ParseCode() in /Users/lnxd/Desktop/BSUIR/SIXTH TERM/MTran/LABS/DONE/LAB3/SyntaxAnalyzer/Parser.cs:line 588
   at SyntaxAnalyzer.Program.Main() in /Users/lnxd/Desktop/BSUIR/SIXTH TERM/MTran/LABS/DONE/LAB3/SyntaxAnalyzer/Program.cs:line 188

Process finished with exit code 134.
```

Рисунок 2.13 - Реакция синтаксического анализатора на ошибку

Примечание. Код программы

```
    using LexicalAnalyzer.Functional;
    using SyntaxAnalyzer.Nodes;

    namespace SyntaxAnalyzer;

    internal class Parser
    {
        public Lexer Lexer { get; set; }
        public List<Token> Tokens { get; set; }
        public int Position { get; set; } = 0;
         public bool Switch { get; set; } = false; // to check
default in code

        public Parser(Lexer lexer, List<Token> tokens)
        {
            Lexer = lexer;
            Tokens = tokens;
        }

          public Token? Match(List<string> tokenTypes) // FIND
TOKEN
        {
            if (Position < Tokens.Count)
            {
                var token = Tokens[Position];

                if (tokenTypes.Contains(token.Identifier))
                {
                    Position++;
                    return token;
                }
            }

            return null;
        }

         public Token Require(List<string> tokenTypes) // REQUIRE
END-POINT OF BLOCK ETC.( ], }, ;, .....)
        {
            var token = Match(tokenTypes);
```

```
            if (token == null)
            {
                                throw new Exception($"after token
'{Tokens[Position - 1].Identifier}' needs token
'{tokenTypes[0]}'");
            }

            return token;
        }

        public List<string> GetVariables()
        {
            var variables = new List<string>();

            foreach (var elem in Lexer.VariablesTables.Values)
            {
                foreach (var elem2 in elem.Keys)
                {
                    variables.Add(elem2);
                }
            }

            return variables;
        }

        AbstractNode ParseVariableType()
        {
            var type = Match(Lexer.VariablesTypes);

            if (type != null)
            {
                return new VariableTypeNode(type);
            }

            throw new Exception($"After token '{Tokens[Position -
1].Identifier}' need var token");
        }


        AbstractNode ParseVariableOrLiteral()
        {
            var number = Match(Lexer.Literals.Keys.ToList());
```

```
            if (number != null)
            {
                return new LiteralNode(number);
            }

            var variable = Match(GetVariables());

            if (variable != null)
            {
                    var leftNode = new VariableNode(variable) as
AbstractNode;
                var @operator = Match(new List<string> { "[" });

                while (@operator != null)
                {
                    @operator.Identifier = "[]";
                    var rightNode = ParseFormula();
                    leftNode = new BinaryOperationNode(@operator,
leftNode, rightNode);
                    Require(new List<string> { "]" });
                    @operator = Match(new List<string> { "[" });
                }

                return leftNode;
            }

                 if (Tokens[Position].Identifier == "false" ||
Tokens[Position].Identifier == "true")
            {
                return new LiteralNode(new Token(Tokens[Position+
+].Identifier, "bool literal"));
            }

            throw new Exception($"After token '{Tokens[Position -
1].Identifier}' needs var or literal token");
        }

        public AbstractNode ParseParentheses()
        {
            if (Match(new List<string> { "(" }) != null)
            {
                var node = ParseFormula();
                Require(new List<string> { ")" });
```

```
                return node;
            }
            else
            {
                return ParseVariableOrLiteral();
            }
        }

        public AbstractNode ParseFormula()
        {
            var leftNode = ParseParentheses();
                                        var     @operator     =
Match(Lexer.CurrentOperations.Keys.ToList());
            @operator ??= Match(new List<string> { "[" });

            while (@operator != null)
            {
                if (@operator.Identifier == "[")
                {
                    @operator.Identifier = "[]";
                    var rightNode = ParseFormula();
                    leftNode = new BinaryOperationNode(@operator,
leftNode, rightNode);
                    Require(new List<string> { "]" });
                                                    @operator   =
Match(Lexer.Operations.Keys.ToList());
                }
                      else if (@operator.Identifier == "++" ||
@operator.Identifier == "--")
                {
                     leftNode = new UnaryOperationNode(@operator,
leftNode);
                                                    @operator   =
Match(Lexer.Operations.Keys.ToList());
                            @operator ??= Match(new List<string>
{ "[" });
                }
                else if (@operator.Identifier == "?")
                {
                    var ifBody = ParseFormula();
                    Require(new List<string> { ":" });
                    var elseBody = ParseFormula();
```

```
                        leftNode = new IfNode(leftNode, ifBody,
elseBody);
                                                    @operator   =
Match(Lexer.Operations.Keys.ToList());
                            @operator ??= Match(new List<string>
{ "[" });
                }
                      else if (@operator.Identifier == "<<" ||
@operator.Identifier == ">>")
                {
                    Position--;
                    break;
                }
                      else if (@operator.Identifier == "==" ||
@operator.Identifier == "!=" || @operator.Identifier == "<"
                          || @operator.Identifier == ">")
                {
                    var rightNode = ParseFormula();
                    leftNode = new BinaryOperationNode(@operator,
leftNode, rightNode);
                                                    @operator   =
Match(Lexer.Operations.Keys.ToList());
                            @operator ??= Match(new List<string>
{ "[" });
                }
                else
                {
                    var rightNode = ParseParentheses();

                    if (leftNode is BinaryOperationNode binary)
                    {
                                        binary.RightNode = new
BinaryOperationNode(@operator, binary.RightNode, rightNode);
                        leftNode = binary;
                    }
                    else
                    {
                                            leftNode = new
BinaryOperationNode(@operator, leftNode, rightNode);
                    }

                                                    @operator   =
Match(Lexer.Operations.Keys.ToList());
```

```
                        @operator ??= Match(new List<string>
{ "[" });
                }
            }

            return leftNode;
        }

        public List<Token> ParseFunctionDefinition()
        {
            var parameters = new List<Token>();

            if (Match(Lexer.VariablesTypes) == null)
            {
                if (Match(new List<string> { ")" }) != null)
                {
                    Position--;
                    return parameters;
                }

                throw new Exception(
                            $"After token '{Tokens[Position -
1].Identifier}' need token var type or ')'");
            }

            var parameter = Match(GetVariables());

            if (parameter == null)
            {
                        throw new Exception($"After token
'{Tokens[Position - 1].Identifier}' need token var");
            }

            parameters.Add(parameter);

            var keySymbol = Match(new List<string> { "," });

            while (keySymbol != null)
            {
                if (Match(Lexer.VariablesTypes) == null)
                {
                    throw new Exception(
```

```
                                $"After token '{Tokens[Position -
1].Identifier}' need token var type");
                }

                parameter = Match(GetVariables());
                parameters.Add(parameter!);
                keySymbol = Match(new List<string> { "," });
            }

            return parameters;
        }

        public List<AbstractNode> ParseCout()
        {
            var parameters = new List<AbstractNode>();

            var @operator = Match(new List<string> { "<<" });

            while (@operator != null)
            {
                if (Match(new List<string> { "endl" }) != null)
                {
                    Position--;
                           var temp = new KeyWordNode(Match(new
List<string> { "endl" })!);
                    parameters.Add(temp);
                    @operator = Match(new List<string> { "<<" });
                    continue;
                }

                var parameter = ParseFormula();
                parameters.Add(parameter);
                @operator = Match(new List<string> { "<<" });
            }

            if (parameters.Count == 0)
            {
                             throw new Exception($"After token
'{Tokens[Position - 1].Identifier}' need token '<<'");
            }

            return parameters;
        }
```

```
        public List<AbstractNode> ParseCin()
        {
            var parameters = new List<AbstractNode>();

            var @operator = Match(new List<string> { ">>" });

            while (@operator != null)
            {
                var parameter = ParseFormula();
                parameters.Add(parameter);
                @operator = Match(new List<string> { ">>" });
            }

            if (parameters.Count == 0)
            {
                                throw new Exception($"After token
'{Tokens[Position - 1].Identifier}' need token '>>'");
            }

            return parameters;
        }

        public List<AbstractNode> ParseFunctionParameters()
        {
            var parameters = new List<AbstractNode>();

            var parameter = ParseFormula();
            parameters.Add(parameter);
            var @operator = Match(new List<string> { "," });

            while (@operator != null)
            {
                parameter = ParseFormula();
                parameters.Add(parameter);
                @operator = Match(new List<string> { "," });
            }

            return parameters;
        }

        public AbstractNode ParseIfElse()
        {
```

```
            Require(new List<string> { "(" });
            var ifCondition = ParseFormula();
            Require(new List<string> { ")" });
            Require(new List<string> { "{" });
            var ifBody = ParseCode();
            Position--;
            Require(new List<string> { "}" });

            AbstractNode? elseBody = null;

            if (Match(new List<string> { "else" }) != null)
            {
                if (Match(new List<string> { "if" }) != null)
                {
                    elseBody = ParseIfElse();
                }
                else
                {
                    Require(new List<string> { "{" });
                    elseBody = ParseCode();
                    Position--;
                    Require(new List<string> { "}" });
                          return new IfNode(ifCondition, ifBody,
elseBody);
                }
            }

            return new IfNode(ifCondition, ifBody, elseBody);
        }

        public AbstractNode? ParseExpression()
        {
            if (Match(Lexer.VariablesTypes) != null)
            {
                                        var   functionToken   =
Match(Lexer.CurrentKeyWords.Keys.ToList());

                if (functionToken != null)
                {
                    if (functionToken.Type == "function")
                    {
                        Require(new List<string> { "(" });
```

```
                                        var parameters =
ParseFunctionDefinition();
                        Require(new List<string> { ")" });
                        Require(new List<string> { "{" });
                        var body = ParseCode();
                        Position--;
                        Require(new List<string> { "}" });
                           return new FunctionNode(functionToken,
parameters, body);
                    }
                    else
                    {
                        throw new Exception(
                                $"After token '{Tokens[Position -
1].Identifier}' need token var or token function");
                    }
                }

                var variableToken = Match(GetVariables());

                if (variableToken != null)
                {
                                        var leftNode = new
VariableNode(variableToken) as AbstractNode;
                         var @operator = Match(new List<string>
{ "," });

                    while (@operator != null)
                    {
                        variableToken = Match(GetVariables());

                        if (variableToken == null)
                        {
                            throw new Exception(
                                $"After token '{Tokens[Position -
1].Identifier}' need token var");
                        }

                                        var rightNode = new
VariableNode(variableToken);
                           leftNode = new BinaryOperationNode(new
Token("=", "operation"), leftNode, rightNode);
```

```
                                @operator = Match(new List<string>
{ "," });
                    }

                    @operator = Match(new List<string> { "=" });

                    if (@operator != null)
                    {
                         if (Match(new List<string> { "new" }) !=
null)
                        {
                                                    var  type  =
Match(Lexer.VariablesTypes);

                            if (type != null)
                            {
                                Position--;
                                            var  typeNode  =
ParseVariableType();
                                    Require(new List<string>
{ "[" });
                                var index = ParseFormula();
                                    Require(new List<string>
{ "]" });
                                    Require(new List<string>
{ ";" });
                                        var rightNode = new
BinaryOperationNode(new  Token("new",  "key  word"),  typeNode,
index);
                                                    return  new
BinaryOperationNode(@operator, leftNode, rightNode);
                            }

                            throw new Exception(
                                $"After token '{Tokens[Position -
1].Identifier}' need token var type");
                        }

                        var value = ParseFormula();
                        Require(new List<string> { ";" });
                              return new BinaryOperationNode(new
Token("=", "operation"), leftNode, value);
                    }
```

```
                    else
                    {
                        Require(new List<string> { ";" });
                        return null;
                    }
                }
            }

            if (Match(GetVariables()) != null)
            {
                Position--;
                var variableNode = ParseVariableOrLiteral();
                var list = Lexer.CurrentOperations.Keys.ToList();
                list.Add("[");
                var @operator = Match(list);

                if (@operator != null)
                {
                            if (@operator.Identifier == "++" ||
@operator.Identifier == "--")
                    {
                                            var unaryNode = new
UnaryOperationNode(@operator, variableNode);
                        Require(new List<string> { ";" });
                        return unaryNode;
                    }

                    if (@operator.Identifier == "[")
                    {
                        @operator.Identifier = "[]";
                        var rightNode = ParseFormula();
                                            variableNode = new
BinaryOperationNode(@operator, variableNode, rightNode);
                        Require(new List<string> { "]" });
                                                @operator =
Match(Lexer.Operations.Keys.ToList());
                    }

                    if (Match(new List<string> { "new" }) !=
null)
                    {
                        var type = Match(Lexer.VariablesTypes);
```

```
                        if (type != null)
                        {
                            Position--;
                            var typeNode = ParseVariableType();
                            Require(new List<string> { "[" });
                            var value = ParseFormula();
                            Require(new List<string> { "]" });
                            Require(new List<string> { ";" });
                               return new BinaryOperationNode(new
Token("new", "key word"), typeNode, value);
                        }

                        throw new Exception(
                                $"After token '{Tokens[Position -
1].Identifier}' need token var type");
                    }

                    var rightFormulaNode = ParseFormula();
                                        var binaryNode = new
BinaryOperationNode(@operator, variableNode, rightFormulaNode);
                    Require(new List<string> { ";" });
                    return binaryNode;
                }
                else
                {
                    Require(new List<string> { ";" });
                    return null;
                }

                            throw new Exception($"After token
'{Tokens[Position - 1].Identifier}' need token operator");
            }

               if (Match(Lexer.CurrentKeyWords.Keys.ToList()) !=
null)
            {
                Position--;
                                                var token =
Match(Lexer.CurrentKeyWords.Keys.ToList());

                switch (token!.Identifier)
                {
                    case "while":
```

```
                    Require(new List<string> { "(" });
                    var condition = ParseFormula();
                    Require(new List<string> { ")" });
                    Require(new List<string> { "{" });
                    var body = ParseCode();
                    Position--;
                    Require(new List<string> { "}" });
                    return new WhileNode(condition, body);
                case "cout":
                    var parameters = ParseCout();
                    Require(new List<string> { ";" });
                    return new CoutNode(parameters);
                case "cin":
                    var cin_parameters = ParseCin();
                    Require(new List<string> { ";" });
                    return new CinNode(cin_parameters);
                case "for":
                    Require(new List<string> { "(" });
                    Match(Lexer.VariablesTypes);
                    var first = ParseFormula();
                    Require(new List<string> { ";" });
                    var second = ParseFormula();
                    Require(new List<string> { ";" });
                    var third = ParseFormula();
                    Require(new List<string> { ")" });
                    Require(new List<string> { "{" });
                    var forBody = ParseCode();
                    Position--;
                    Require(new List<string> { "}" });
                     return new ForNode(first, second, third,
forBody);
                case "if":
                    return ParseIfElse();
                case "switch":
                    Require(new List<string> { "(" });
                    var variable = Match(GetVariables());

                    if (variable == null)
                    {
                        throw new Exception(
                            $"After token '{Tokens[Position -
1].Identifier}' need token var");
                    }
```

```
                        Require(new List<string> { ")" });
                        Require(new List<string> { "{" });
                        Switch = true;
                        var switchBody = ParseCode();
                        Switch = false;
                        Position--;
                        Require(new List<string> { "}" });
                                 return new SwitchNode(variable,
switchBody);
                    case "case":
                        if (Switch)
                        {
                                                var literalNode =
ParseVariableOrLiteral() as LiteralNode;

                            if (literalNode == null)
                            {
                                throw new Exception(
                                                    $"After token
{Tokens[Position - 1].Identifier} need token literal");
                            }

                            Require(new List<string> { ":" });
                                                 return new
CaseNode(literalNode.Literal);
                        }

                          throw new Exception("Unexpected token:
'case' without 'switch'");
                    case "default":
                        if (Switch)
                        {
                            Require(new List<string> { ":" });
                            return new KeyWordNode(token);
                        }

                          throw new Exception("Unexpected token:
'default' without 'switch'");
                    case "break":
                        Require(new List<string> { ";" });
                        return new KeyWordNode(token);
                    case "continue":
```

```
                        Require(new List<string> { ";" });
                        return new KeyWordNode(token);
                    default:
                        Require(new List<string> { "(" });
                                    var  functionParameters  =
ParseFunctionParameters();
                        Require(new List<string> { ")" });
                        Require(new List<string> { ";" });
                          return new FunctionExecutionNode(token,
functionParameters);
                }
            }

              throw new Exception($"Need var type, var, literal
{Position}");
        }

        public AbstractNode ParseCode()
        {
            var root = new StatementsNode();

            while (Position < Tokens.Count)
            {
                if (Match(new List<string> { "}" }) != null)
                {
                    return root;
                }

                var statementNode = ParseExpression();

                if (statementNode != null)
                {
                    root.AddNode(statementNode);
                }
            }

            return root;
        }
    }
```