Министерство науки и высшего образования Российской Федерации

федеральное государственное автономное

образовательное учреждение высшего образования

«Национальный исследовательский университет

«Московский институт электронной техники»

институт системной и программной инженерии

и информационных технологий»

Курс «Теория алгоритмических языков и компиляторов»

Лабораторная работа № 4

Разработка транслятора заданных конструкций языка СИ++. Разработка синтаксического анализатора, обнаруживающего максимальное число ошибок.

Выполнил:

Студент группы ПИН-51Д

Джугели Дмитрий Александрович

Москва, 2025

Задание на лабораторную работу

Написать синтаксический анализатор, обнаруживающий наибольшее число ошибок, для приведённой ниже грамматики (данная грамматика является упрощённым вариантом грамматики языка С):

<program>: <type> ’main’ ‘(‘ ‘)’ ‘{‘ **<statement>** ‘}’

<type>: ‘int’

| ‘bool’

| ‘void’

<statement>:

| <declaration> ‘;’

| ‘{‘ **<statement>** ‘}’

| <for> <statement>

| <if> <statement>

| <return>

<declaration>: <type> <identifier> <assign>

<identifier>: <character><id_end>

<character>: ‘a’ | ‘b’ | ‘c’ | ‘d’ | ‘e’ | ‘f’ | ‘g’ | ‘h’ | ‘i' | ‘j’ | ‘k’ | ‘l’ | ‘m’ | ‘n’ | ‘o’ | ‘p’ | ‘q’ | ‘r’ | ‘s’ | ‘t’ | ‘u’ | ‘v’ | ‘w’ | ‘x’ | ‘y’ | ‘z’ | ‘A’ | ‘B’ | ‘C’ | ‘D’ | ‘E’ | ‘F’ | ‘G’ | ‘H’ | ‘I’ | ‘J’ | ‘K’ | ‘L’ | ‘M’ | ‘N’ | ‘O’ | ‘P’ | ‘Q’ | ‘R’ | ‘S’ | ‘T’ | ‘U’ | ‘V’ | ‘W’ | ‘X’ | ‘Y’ | ‘Z’ | ‘_’

<id_end>:

| <character><id_end>

<assign>:

| ‘=’ <assign_end>

<assign_end>: <identifier>

| <number>

<number>: <digit><number_end>

<digit>: ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’

<number_end>:

| <digit><number_end>

<for>: ‘for’ ‘(‘ **<declaration>** ‘;’ **<bool_expression>** ‘;’ ‘)’

<bool_expression>: <identifier> <relop> <identifier>

| <number> <relop> <identifier>

<relop>: ‘<’ | ‘>’ | ‘==’ | ‘!=’

<if>: ‘if’ ‘(‘ **<bool_expression>** ‘)’

<return>: ‘return’ **<number>** ‘;’

Данная грамматика записана с помощью вариации РБНФ. Нетерминалы заключены в **<>**, терминал (или последовательность терминалов) – в **‘’**. Символ **|** обозначает альтернативу (т.е. «или»). Запись **<N1><N2>** означает, конкатенацию нетерминалов, а запись **<N1> <N2>** – последовательную запись нетерминалов, разделённых пробельными символами (пробельные символы аналогичны таковым в грамматике языка C). Запись вида **<N1>: | <N2>** означает, что нетерминал N1 можно заменить либо нетерминалом N2, либо пустой строкой. Стартовым нетерминалом грамматики является **<program>.** Назовём язык, описываемый данной граматикой, C-light.

Входные данные: файл с программой на языке C-light

Выходные данные:

1. **Задание минимум:** количество обнаруженных ошибок и сообщения о месте их возникновения
2. **Задание максимум:** количество обнаруженных ошибок, сообщения о месте их возникновения и о их типе, файл с таблицей, аналогичной таблице из примера 4. Востановление после ошибок реализовать в «режиме паники»

В качестве **дополнения** может быть реализован механизм восстановления после ошибок в режиме фразы, а также сгенерирован файл с таблицей предиктивного анализа.

Код:

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;

namespace SyntaxAnalyzer
{
    public enum TokenType
    {
        // Ключевые слова
        INT, BOOL, VOID, MAIN, FOR, IF, RETURN,

        // Идентификаторы и литералы
        IDENTIFIER, NUMBER,

        // Операторы и разделители
        LPAREN, RPAREN, LBRACE, RBRACE, SEMICOLON, ASSIGN,
        LESS, GREATER, EQUAL, NOT_EQUAL,

        // Конец файла
        EOF,

        // Ошибочный токен
        ERROR
    }

    public class Token
    {
        public TokenType Type { get; set; }
        public string Value { get; set; }
        public int Line { get; set; }
        public int Position { get; set; }

        public Token(TokenType type, string value, int line, int position)
        {
            Type = type;
            Value = value;
            Line = line;
            Position = position;
        }

        public override string ToString()
        {
            return $"{Type} '{Value}' at line {Line}, position {Position}";
        }
    }

    public class Lexer
    {
        private readonly string _input;
        private int _position;
        private int _line;
        private int _linePosition;

        public Lexer(string input)
        {
            _input = input;
            _position = 0;
            _line = 1;
            _linePosition = 1;
        }
```

```
        private char Current
        {
            get { return _position < _input.Length ? _input[_position] : '\0'; }
        }

        private void Advance()
        {
            if (Current == '\n')
            {
                _line++;
                _linePosition = 1;
            }
            else
            {
                _linePosition++;
            }
            _position++;
        }

        private void SkipWhitespace()
        {
            while (char.IsWhiteSpace(Current))
            {
                Advance();
            }
        }

        public Token NextToken()
        {
            SkipWhitespace();

            if (_position >= _input.Length)
                return new Token(TokenType.EOF, "", _line, _linePosition);

            char current = Current;
            int startLine = _line;
            int startPos = _linePosition;

            // Ключевые слова и идентификаторы
            if (char.IsLetter(current) || current == '_')
            {
                string identifier = ReadIdentifier();
                switch (identifier.ToLower())
                {
                    case "int":
                        return new Token(TokenType.INT, identifier, startLine, startPos);
                    case "bool":
                        return new Token(TokenType.BOOL, identifier, startLine, startPos);
                    case "void":
                        return new Token(TokenType.VOID, identifier, startLine, startPos);
                    case "main":
                        return new Token(TokenType.MAIN, identifier, startLine, startPos);
                    case "for":
                        return new Token(TokenType.FOR, identifier, startLine, startPos);
                    case "if":
                        return new Token(TokenType.IF, identifier, startLine, startPos);
                    case "return":
                        return new Token(TokenType.RETURN, identifier, startLine, startPos);
                    default:
                        return new Token(TokenType.IDENTIFIER, identifier, startLine,
startPos);
                }
            }

            // Числа
```

```
            if (char.IsDigit(current))
            {
                string number = ReadNumber();
                return new Token(TokenType.NUMBER, number, startLine, startPos);
            }

            // Операторы и разделители
            switch (current)
            {
                case '(':
                    Advance();
                    return new Token(TokenType.LPAREN, "(", startLine, startPos);
                case ')':
                    Advance();
                    return new Token(TokenType.RPAREN, ")", startLine, startPos);
                case '{':
                    Advance();
                    return new Token(TokenType.LBRACE, "{", startLine, startPos);
                case '}':
                    Advance();
                    return new Token(TokenType.RBRACE, "}", startLine, startPos);
                case ';':
                    Advance();
                    return new Token(TokenType.SEMICOLON, ";", startLine, startPos);
                case '=':
                    Advance();
                    if (Current == '=')
                    {
                        Advance();
                        return new Token(TokenType.EQUAL, "==", startLine, startPos);
                    }
                    return new Token(TokenType.ASSIGN, "=", startLine, startPos);
                case '<':
                    Advance();
                    return new Token(TokenType.LESS, "<", startLine, startPos);
                case '>':
                    Advance();
                    return new Token(TokenType.GREATER, ">", startLine, startPos);
                case '!':
                    Advance();
                    if (Current == '=')
                    {
                        Advance();
                        return new Token(TokenType.NOT_EQUAL, "!=", startLine, startPos);
                    }
                    break;
            }

            // Неизвестный символ
            Advance();
            return new Token(TokenType.ERROR, current.ToString(), startLine, startPos);
        }

        private string ReadIdentifier()
        {
            int start = _position;
            while (char.IsLetterOrDigit(Current) || Current == '_')
            {
                Advance();
            }
            return _input.Substring(start, _position - start);
        }

        private string ReadNumber()
        {
```

```
            int start = _position;
            while (char.IsDigit(Current))
            {
                Advance();
            }
            return _input.Substring(start, _position - start);
        }

        public override string ToString()
        {
            return _input;
        }
    }

    public class SyntaxError : Exception
    {
        public int Line { get; }
        public int Position { get; }
        public string Expected { get; }
        public string Found { get; }

        public SyntaxError(string message, int line, int position, string expected = "",
string found = "")
            : base(message)
        {
            Line = line;
            Position = position;
            Expected = expected;
            Found = found;
        }
    }

    public class Parser
    {
        private Lexer _lexer;
        private Token _currentToken;
        private readonly List<SyntaxError> _errors;
        private readonly List<string> _symbolTable;

        public Parser(Lexer lexer)
        {
            _lexer = lexer;
            _currentToken = _lexer.NextToken();
            _errors = new List<SyntaxError>();
            _symbolTable = new List<string>();
        }

        public List<SyntaxError> Errors
        {
            get { return _errors; }
        }

        public List<string> SymbolTable
        {
            get { return _symbolTable; }
        }

        private void Eat(TokenType expectedType)
        {
            if (_currentToken.Type == expectedType)
            {
                _currentToken = _lexer.NextToken();
            }
            else
            {
```

```
                throw new SyntaxError($"Expected {expectedType}, but found
{_currentToken.Type}",
                    _currentToken.Line, _currentToken.Position, expectedType.ToString(),
_currentToken.Type.ToString());
            }
        }

        private void ErrorRecovery(TokenType[] syncTokens)
        {
            _errors.Add(new SyntaxError("Syntax error", _currentToken.Line,
_currentToken.Position));

            // Режим паники: пропускаем токены до синхронизации
            while (_currentToken.Type != TokenType.EOF &&
!syncTokens.Contains(_currentToken.Type))
            {
                _currentToken = _lexer.NextToken();
            }
        }

        public void Parse()
        {
            try
            {
                Console.WriteLine("=== DEBUG: TOKEN STREAM ===");
                // Сохраняем текущее состояние
                Lexer debugLexer = new Lexer(_lexer.ToString());
                Token debugToken = debugLexer.NextToken();

                while (debugToken.Type != TokenType.EOF)
                {
                    Console.WriteLine(debugToken);
                    debugToken = debugLexer.NextToken();
                }
                Console.WriteLine(debugToken); // EOF
                Console.WriteLine("============================");

                Program();
            }
            catch (SyntaxError ex)
            {
                _errors.Add(ex);
                ErrorRecovery(new[] { TokenType.EOF });
            }
        }

        private void Program()
        {
            try
            {
                Console.WriteLine($"Program(): current token = {_currentToken}");
                Type();
                Eat(TokenType.MAIN);
                Eat(TokenType.LPAREN);
                Eat(TokenType.RPAREN);
                Eat(TokenType.LBRACE);
                Statement();
                Eat(TokenType.RBRACE);
            }
            catch (SyntaxError ex)
            {
                _errors.Add(ex);
                ErrorRecovery(new[] { TokenType.EOF });
            }
        }
```

```
        private void Type()
        {
            Console.WriteLine($"Type(): current token = {_currentToken}");

            switch (_currentToken.Type)
            {
                case TokenType.INT:
                    Console.WriteLine("Found INT type");
                    Eat(TokenType.INT);
                    break;
                case TokenType.BOOL:
                    Console.WriteLine("Found BOOL type");
                    Eat(TokenType.BOOL);
                    break;
                case TokenType.VOID:
                    Console.WriteLine("Found VOID type");
                    Eat(TokenType.VOID);
                    break;
                default:
                    Console.WriteLine($"ERROR: Expected type, but got {_currentToken}");
                    throw new SyntaxError("Expected type (int, bool, or void)",
                        _currentToken.Line, _currentToken.Position);
            }
        }

        private void Statement()
        {
            try
            {
                Console.WriteLine($"Statement(): current token = {_currentToken}");

                switch (_currentToken.Type)
                {
                    case TokenType.INT:
                    case TokenType.BOOL:
                    case TokenType.VOID:
                        Console.WriteLine("Processing declaration statement");
                        Declaration();
                        Eat(TokenType.SEMICOLON);
                        break;
                    case TokenType.LBRACE:
                        Console.WriteLine("Processing block statement");
                        Eat(TokenType.LBRACE);
                        Statement();
                        Eat(TokenType.RBRACE);
                        break;
                    case TokenType.FOR:
                        Console.WriteLine("Processing for statement");
                        For();
                        Statement();
                        break;
                    case TokenType.IF:
                        Console.WriteLine("Processing if statement");
                        If();
                        Statement();
                        break;
                    case TokenType.RETURN:
                        Console.WriteLine("Processing return statement");
                        Return();
                        break;
                    default:
                        Console.WriteLine($"ERROR: Expected statement, but got
{_currentToken}");
                        throw new SyntaxError("Expected statement",
```

```
                            _currentToken.Line, _currentToken.Position);
                }
            }
            catch (SyntaxError ex)
            {
                _errors.Add(ex);
                ErrorRecovery(new[] { TokenType.SEMICOLON, TokenType.RBRACE, TokenType.RETURN,
                    TokenType.INT, TokenType.BOOL, TokenType.VOID, TokenType.FOR, TokenType.IF
});
            }
        }

        private void Declaration()
        {
            try
            {
                Console.WriteLine($"Declaration(): current token = {_currentToken}");
                Type();
                Identifier();
                Assign();
            }
            catch (SyntaxError ex)
            {
                _errors.Add(ex);
                ErrorRecovery(new[] { TokenType.SEMICOLON });
            }
        }

        private void Identifier()
        {
            Console.WriteLine($"Identifier(): current token = {_currentToken}");

            if (_currentToken.Type == TokenType.IDENTIFIER)
            {
                if (!_symbolTable.Contains(_currentToken.Value))
                {
                    _symbolTable.Add(_currentToken.Value);
                }
                Eat(TokenType.IDENTIFIER);
            }
            else
            {
                Console.WriteLine($"ERROR: Expected identifier, but got {_currentToken}");
                throw new SyntaxError("Expected identifier",
                    _currentToken.Line, _currentToken.Position);
            }
        }

        private void Assign()
        {
            Console.WriteLine($"Assign(): current token = {_currentToken}");

            if (_currentToken.Type == TokenType.ASSIGN)
            {
                Eat(TokenType.ASSIGN);
                AssignEnd();
            }
            // Эпсилон-правило - ничего не делаем
        }

        private void AssignEnd()
        {
            Console.WriteLine($"AssignEnd(): current token = {_currentToken}");

            if (_currentToken.Type == TokenType.IDENTIFIER)
```

```
            {
                Identifier();
            }
            else if (_currentToken.Type == TokenType.NUMBER)
            {
                Number();
            }
            else
            {
                Console.WriteLine($"ERROR: Expected identifier or number after '=', but got
{_currentToken}");
                throw new SyntaxError("Expected identifier or number after '='",
                    _currentToken.Line, _currentToken.Position);
            }
        }

        private void Number()
        {
            Console.WriteLine($"Number(): current token = {_currentToken}");

            if (_currentToken.Type == TokenType.NUMBER)
            {
                Eat(TokenType.NUMBER);
            }
            else
            {
                Console.WriteLine($"ERROR: Expected number, but got {_currentToken}");
                throw new SyntaxError("Expected number",
                    _currentToken.Line, _currentToken.Position);
            }
        }

        private void For()
        {
            try
            {
                Console.WriteLine($"For(): current token = {_currentToken}");
                Eat(TokenType.FOR);
                Eat(TokenType.LPAREN);
                Declaration();
                Eat(TokenType.SEMICOLON);
                BoolExpression();
                Eat(TokenType.SEMICOLON);
                Eat(TokenType.RPAREN);
            }
            catch (SyntaxError ex)
            {
                _errors.Add(ex);
                ErrorRecovery(new[] { TokenType.LBRACE, TokenType.INT, TokenType.BOOL,
                    TokenType.VOID, TokenType.FOR, TokenType.IF, TokenType.RETURN });
            }
        }

        private void BoolExpression()
        {
            try
            {
                Console.WriteLine($"BoolExpression(): current token = {_currentToken}");

                if (_currentToken.Type == TokenType.IDENTIFIER)
                {
                    Identifier();
                }
                else if (_currentToken.Type == TokenType.NUMBER)
                {
```

```
                    Number();
                }
                else
                {
                    Console.WriteLine($"ERROR: Expected identifier or number in boolean
expression, but got {_currentToken}");
                    throw new SyntaxError("Expected identifier or number in boolean
expression",
                        _currentToken.Line, _currentToken.Position);
                }

                Relop();

                // Исправление: второй операнд тоже может быть числом или идентификатором
                if (_currentToken.Type == TokenType.IDENTIFIER)
                {
                    Identifier();
                }
                else if (_currentToken.Type == TokenType.NUMBER)
                {
                    Number();
                }
                else
                {
                    Console.WriteLine($"ERROR: Expected identifier or number in boolean
expression, but got {_currentToken}");
                    throw new SyntaxError("Expected identifier or number in boolean
expression",
                        _currentToken.Line, _currentToken.Position);
                }
            }
            catch (SyntaxError ex)
            {
                _errors.Add(ex);
                ErrorRecovery(new[] { TokenType.SEMICOLON });
            }
        }

        private void Relop()
        {
            Console.WriteLine($"Relop(): current token = {_currentToken}");

            switch (_currentToken.Type)
            {
                case TokenType.LESS:
                    Eat(TokenType.LESS);
                    break;
                case TokenType.GREATER:
                    Eat(TokenType.GREATER);
                    break;
                case TokenType.EQUAL:
                    Eat(TokenType.EQUAL);
                    break;
                case TokenType.NOT_EQUAL:
                    Eat(TokenType.NOT_EQUAL);
                    break;
                default:
                    Console.WriteLine($"ERROR: Expected relational operator, but got
{_currentToken}");
                    throw new SyntaxError("Expected relational operator (<, >, ==, !=)",
                        _currentToken.Line, _currentToken.Position);
            }
        }

        private void If()
```

```
        {
            try
            {
                Console.WriteLine($"If(): current token = {_currentToken}");
                Eat(TokenType.IF);
                Eat(TokenType.LPAREN);
                BoolExpression();
                Eat(TokenType.RPAREN);
            }
            catch (SyntaxError ex)
            {
                _errors.Add(ex);
                ErrorRecovery(new[] { TokenType.LBRACE, TokenType.INT, TokenType.BOOL,
                    TokenType.VOID, TokenType.FOR, TokenType.IF, TokenType.RETURN });
            }
        }

        private void Return()
        {
            try
            {
                Console.WriteLine($"Return(): current token = {_currentToken}");
                Eat(TokenType.RETURN);
                Number();
                Eat(TokenType.SEMICOLON);
            }
            catch (SyntaxError ex)
            {
                _errors.Add(ex);
                ErrorRecovery(new[] { TokenType.SEMICOLON, TokenType.RBRACE, TokenType.INT,
                    TokenType.BOOL, TokenType.VOID, TokenType.FOR, TokenType.IF,
TokenType.RETURN });
            }
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            if (args.Length == 0)
            {
                Console.WriteLine("Usage: SyntaxAnalyzer <input_file>");
                Console.WriteLine("Testing with sample input...");

                // Тестовый ввод
                string testInput = "int main() { int x = 5; return 0; }";
                Console.WriteLine($"Test input: {testInput}");

                TestWithInput(testInput);
                return;
            }

            try
            {
                string input = File.ReadAllText(args[0]);
                Console.WriteLine($"Input file: {args[0]}");
                Console.WriteLine($"File content: '{input}'");

                Lexer lexer = new Lexer(input);
                Parser parser = new Parser(lexer);

                parser.Parse();
```

```
                Console.WriteLine($"\nAnalysis completed. Found {parser.Errors.Count}
error(s):");
                Console.WriteLine("==================================================");

                foreach (var error in parser.Errors)
                {
                    Console.WriteLine($"Line {error.Line}, Position {error.Position}:
{error.Message}");
                    if (!string.IsNullOrEmpty(error.Expected))
                    {
                        Console.WriteLine($"  Expected: {error.Expected}");
                    }
                    if (!string.IsNullOrEmpty(error.Found))
                    {
                        Console.WriteLine($"  Found: {error.Found}");
                    }
                }

                Console.WriteLine("\nSymbol Table:");
                Console.WriteLine("=============");
                foreach (var symbol in parser.SymbolTable)
                {
                    Console.WriteLine(symbol);
                }

                // Сохранение результатов в файл
                SaveResultsToFile(parser, Path.ChangeExtension(args[0], ".analysis.txt"));
            }
            catch (Exception ex)
            {
                Console.WriteLine($"Error: {ex.Message}");
            }
        }

        static void TestWithInput(string input)
        {
            Lexer lexer = new Lexer(input);
            Parser parser = new Parser(lexer);

            parser.Parse();

            Console.WriteLine($"\nAnalysis completed. Found {parser.Errors.Count} error(s):");
            Console.WriteLine("==================================================");

            foreach (var error in parser.Errors)
            {
                Console.WriteLine($"Line {error.Line}, Position {error.Position}:
{error.Message}");
                if (!string.IsNullOrEmpty(error.Expected))
                {
                    Console.WriteLine($"  Expected: {error.Expected}");
                }
                if (!string.IsNullOrEmpty(error.Found))
                {
                    Console.WriteLine($"  Found: {error.Found}");
                }
            }

            Console.WriteLine("\nSymbol Table:");
            Console.WriteLine("=============");
            foreach (var symbol in parser.SymbolTable)
            {
                Console.WriteLine(symbol);
            }
        }
```

```
        static void SaveResultsToFile(Parser parser, string filename)
        {
            using (StreamWriter writer = new StreamWriter(filename))
            {
                writer.WriteLine("SYNTAX ANALYSIS RESULTS");
                writer.WriteLine("======================");
                writer.WriteLine($"Total errors: {parser.Errors.Count}");
                writer.WriteLine();

                if (parser.Errors.Count > 0)
                {
                    writer.WriteLine("ERROR DETAILS:");
                    writer.WriteLine("==============");
                    foreach (var error in parser.Errors)
                    {
                        writer.WriteLine($"Line {error.Line}, Position {error.Position}:
{error.Message}");
                        if (!string.IsNullOrEmpty(error.Expected))
                        {
                            writer.WriteLine($"  Expected: {error.Expected}");
                        }
                        if (!string.IsNullOrEmpty(error.Found))
                        {
                            writer.WriteLine($"  Found: {error.Found}");
                        }
                    }
                    writer.WriteLine();
                }

                writer.WriteLine("SYMBOL TABLE:");
                writer.WriteLine("=============");
                foreach (var symbol in parser.SymbolTable)
                {
                    writer.WriteLine(symbol);
                }
            }

            Console.WriteLine($"Results saved to: {filename}");
        }
    }
}
```

Тестовые данные:

int main() {

int x = 5;

}

Работа программы:

```
C:\Users\ant_daddy\OneDrive\Документы\lr4>TA4.exe simple.txt
Input file: simple.txt
File content: 'int main() {
    int x = 5;
}
'
=== DEBUG: TOKEN STREAM ===
INT 'int' at line 1, position 1
MAIN 'main' at line 1, position 5
LPAREN '(' at line 1, position 9
RPAREN ')' at line 1, position 10
LBRACE '{' at line 1, position 12
INT 'int' at line 2, position 5
IDENTIFIER 'x' at line 2, position 9
ASSIGN '=' at line 2, position 11
NUMBER '5' at line 2, position 13
SEMICOLON ';' at line 2, position 14
RBRACE '}' at line 3, position 1
EOF '' at line 4, position 1
==========================
Program(): current token = INT 'int' at line 1, position 1
Type(): current token = INT 'int' at line 1, position 1
Found INT type
Statement(): current token = INT 'int' at line 2, position 5
Processing declaration statement
Declaration(): current token = INT 'int' at line 2, position 5
Type(): current token = INT 'int' at line 2, position 5
Found INT type
Identifier(): current token = IDENTIFIER 'x' at line 2, position 9
Assign(): current token = ASSIGN '=' at line 2, position 11
AssignEnd(): current token = NUMBER '5' at line 2, position 13
Number(): current token = NUMBER '5' at line 2, position 13
```

Тестовые данные:

```
int main() {
    x = 5;          //нет типа
    int number;     //объявление без присваивания
    return ;      //нет числа
    if(x > 5) {     //условный оператор
        return 1;
    }
}
```

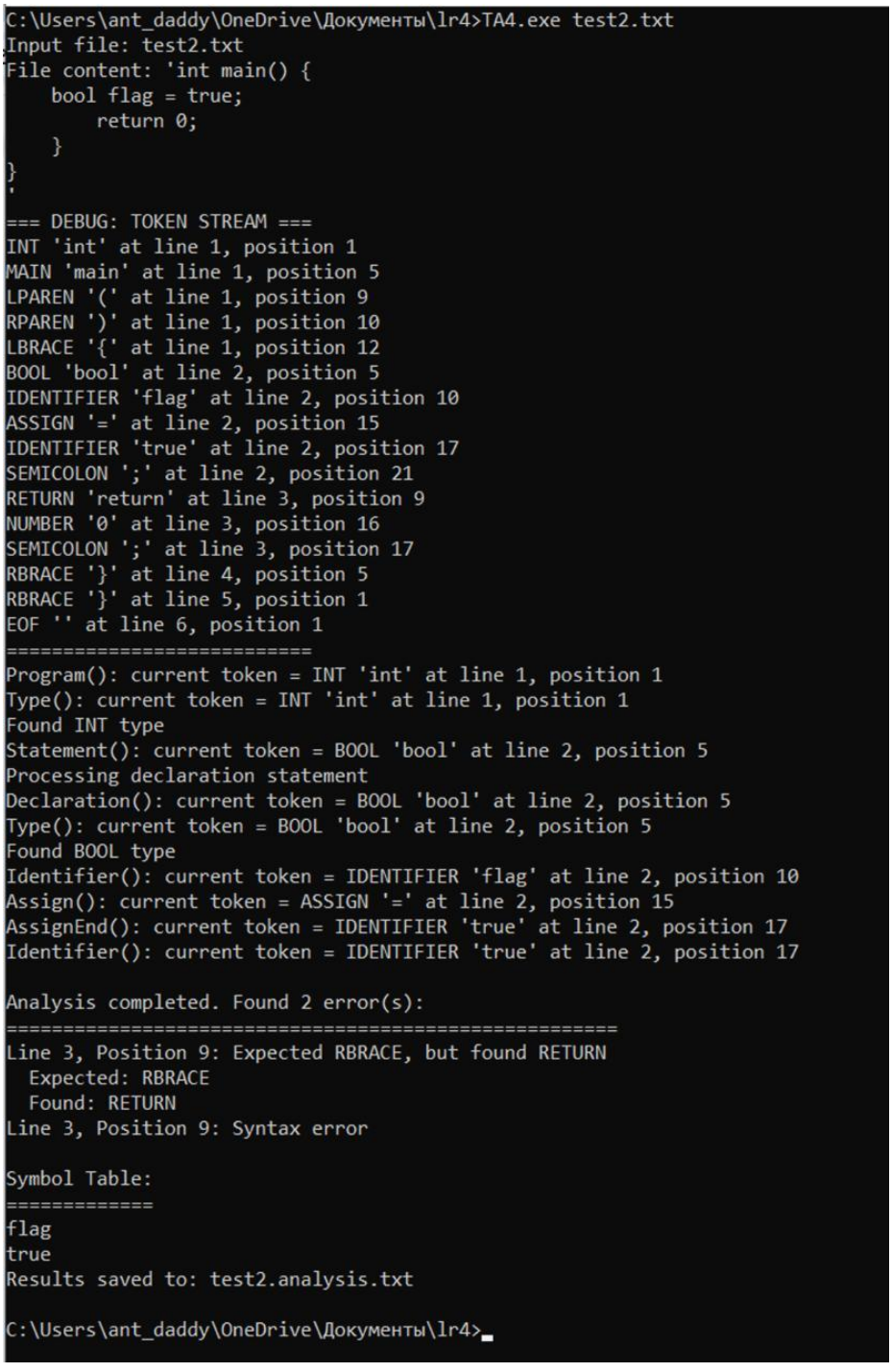
Работа программы:

```
C:\Users\ant_daddy\OneDrive\Документы\lr4>TA4.exe error.txt
Input file: error.txt
File content: 'int main() {
    x = 5;
    int number;
    return ;
    if(x > 5) {
        return 1;
    }
}
'
=== DEBUG: TOKEN STREAM ===
INT 'int' at line 1, position 1
MAIN 'main' at line 1, position 5
LPAREN '(' at line 1, position 9
RPAREN ')' at line 1, position 10
LBRACE '{' at line 1, position 12
IDENTIFIER 'x' at line 2, position 5
ASSIGN '=' at line 2, position 7
NUMBER '5' at line 2, position 9
SEMICOLON ';' at line 2, position 10
INT 'int' at line 3, position 5
IDENTIFIER 'number' at line 3, position 9
SEMICOLON ';' at line 3, position 15
RETURN 'return' at line 4, position 5
SEMICOLON ';' at line 4, position 12
IF 'if' at line 5, position 5
LPAREN '(' at line 5, position 7
IDENTIFIER 'x' at line 5, position 8
GREATER '>' at line 5, position 10
NUMBER '5' at line 5, position 12
RPAREN ')' at line 5, position 13
LBRACE '{' at line 5, position 15
RETURN 'return' at line 6, position 9
NUMBER '1' at line 6, position 16
SEMICOLON ';' at line 6, position 17
RBRACE '}' at line 7, position 5
RBRACE '}' at line 8, position 1
EOF '' at line 9, position 1
==========================
Program(): current token = INT 'int' at line 1, position 1
Type(): current token = INT 'int' at line 1, position 1
Found INT type
Statement(): current token = IDENTIFIER 'x' at line 2, position 5
ERROR: Expected statement, but got IDENTIFIER 'x' at line 2, position 5

Analysis completed. Found 4 error(s):
======================================================
```

Тестовые данные:

```
int main() {
    bool flag = true;
        return 0;
    }
}
```

Работа программы:

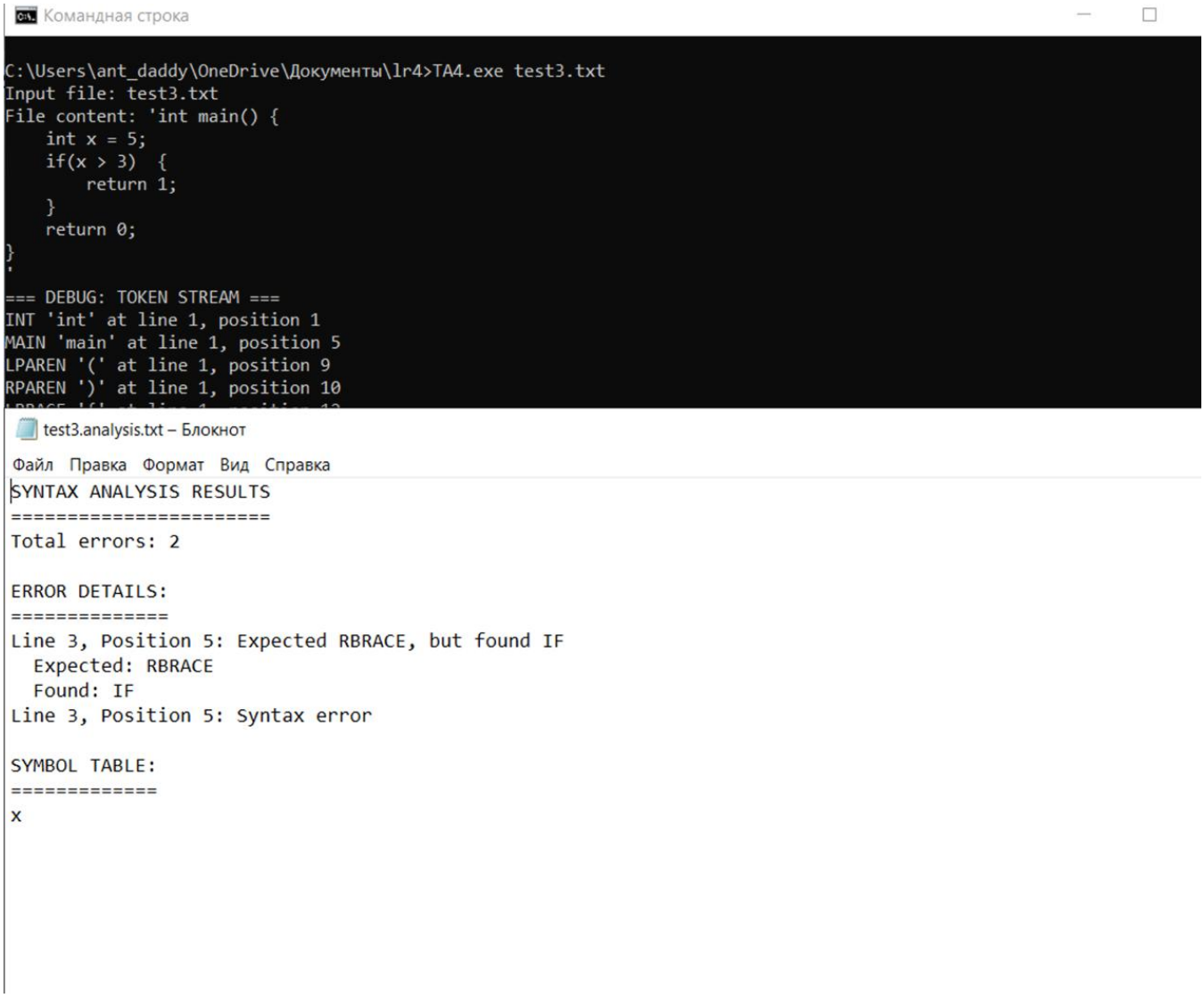


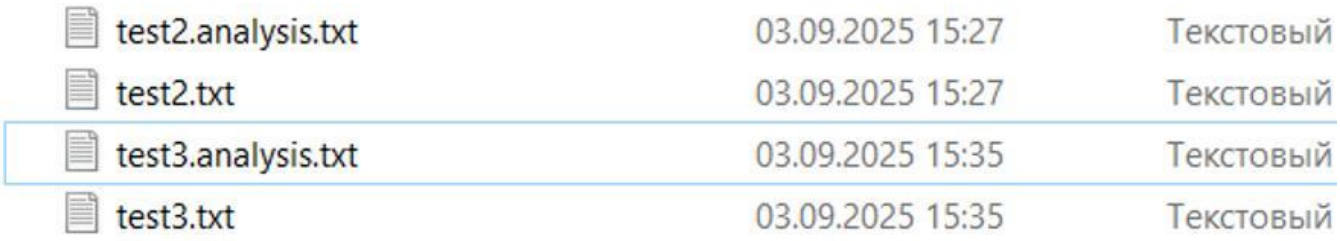
```
C:\Users\ant_daddy\OneDrive\Документы\lr4>TA4.exe test2.txt
Input file: test2.txt
File content: 'int main() {
    bool flag = true;
        return 0;
    }
}
'

=== DEBUG: TOKEN STREAM ===
INT 'int' at line 1, position 1
MAIN 'main' at line 1, position 5
LPAREN '(' at line 1, position 9
RPAREN ')' at line 1, position 10
LBRACE '{' at line 1, position 12
BOOL 'bool' at line 2, position 5
IDENTIFIER 'flag' at line 2, position 10
ASSIGN '=' at line 2, position 15
IDENTIFIER 'true' at line 2, position 17
SEMICOLON ';' at line 2, position 21
RETURN 'return' at line 3, position 9
NUMBER '0' at line 3, position 16
SEMICOLON ';' at line 3, position 17
RBRACE '}' at line 4, position 5
RBRACE '}' at line 5, position 1
EOF '' at line 6, position 1
==========================
Program(): current token = INT 'int' at line 1, position 1
Type(): current token = INT 'int' at line 1, position 1
Found INT type
Statement(): current token = BOOL 'bool' at line 2, position 5
Processing declaration statement
Declaration(): current token = BOOL 'bool' at line 2, position 5
Type(): current token = BOOL 'bool' at line 2, position 5
Found BOOL type
Identifier(): current token = IDENTIFIER 'flag' at line 2, position 10
Assign(): current token = ASSIGN '=' at line 2, position 15
AssignEnd(): current token = IDENTIFIER 'true' at line 2, position 17
Identifier(): current token = IDENTIFIER 'true' at line 2, position 17

Analysis completed. Found 2 error(s):
==================================================
Line 3, Position 9: Expected RBRACE, but found RETURN
  Expected: RBRACE
  Found: RETURN
Line 3, Position 9: Syntax error

Symbol Table:
=============
flag
true
Results saved to: test2.analysis.txt

C:\Users\ant_daddy\OneDrive\Документы\lr4>
```

Тестовые данные:

```
int main() {
    int x = 5;
    if(x  {
        return 1;
    }
    return 0;
}
```

Работа программы:




test2.analysis.txt 03.09.2025 15:27 Текстовый
test2.txt 03.09.2025 15:27 Текстовый
test3.analysis.txt 03.09.2025 15:35 Текстовый
test3.txt 03.09.2025 15:35 Текстовый