Листинг программы (Анализирующей части)

Класс Scan.cs

```
public class Scan
{
    Dictionary<string, Token> state = new Dictionary<string, Token>();
    public string text;
    public string[] words;

    public Dictionary<string, int> keyValuePairs = new Dictionary<string, int>()
    {
        { "struct", 1 },
        { "int", 2},
        { "string", 3},
        { "double", 4},
        { "float", 5},
        { " ", 9},
        { "{", 10},
        { "}", 11},
        { ";", 12}
    };
    public Scan(string text)
```

```
{
    this.text = text;
}
public List<string> keywords = new List<string>();
public List<string> keyword = new List<string>();
public List<int> codes = new List<int>();
public List<Token> tokens = new List<Token>();
public List<string> errors = new List<string>();

public List<string> fortoken = new List<string>();
public void Tokenize()
{
    for (int i = 0; i < codes.Count; i++)
    {
        switch (codes[i])
        {
            case 1:
                tokens.Add(Token.STRUCT);
                fortoken.Add(keyword[i]);
                break;
            case 2:
            case 3:
            case 4:
```

```
            case 5:
                tokens.Add(Token.TYPE);
                fortoken.Add(keyword[i]);
                break;
            case 10:
                tokens.Add(Token.OPENBRACE);
                break;
            case 11:
                tokens.Add(Token.CLOSEBRACE);
                fortoken.Add(keyword[i]);
                break;
            case 12:
                tokens.Add(Token.SEM);
                fortoken.Add(keyword[i]);
                break;
            case 13:
                tokens.Add(Token.ID);
                fortoken.Add(keyword[i]);
                break;
            default:
                break;
        }
    }
```

```
}
public List<MyToken> myTokens = new List<MyToken>();
public void Lexic()
{
    int i = 0;
    int prevcode = 0;
    int line = 1;
    while (i < text.Length)
    {
        char c = text[i];
        if (c == '\n')
        {
            line++;
            i++;
            continue;
        }
        // Пробел
        if (char.IsWhiteSpace(c))
        {
            if (prevcode != 9 && (prevcode >= 1 && prevcode < 9))
            {
                codes.Add(keyValuePairs[" "]);
                keywords.Add("WhiteSpace");
```

```
            keyword.Add(" ");
        }
        i++;
        continue;
    }

    // Разделители { } ;
    if (IsDelimiter(c.ToString()))
    {
        int code = keyValuePairs[c.ToString()];
        codes.Add(code);
        if (code > 0 && code < 6)
            keywords.Add("keyword");
        else if (code == 10 || code == 11)
            keywords.Add("razdelitel");
        else if (code == 12)
            keywords.Add("end operator");
        myTokens.Add(new MyToken(  i, line, c.ToString()));
        keyword.Add(c.ToString());
        prevcode = code;
        i++;
        continue;
    }
```

```
            // Идентификатор или ключевое слово
            if (char.IsLetter(c) || c == '_')
            {
                string cleaned = "";
                int start = i;
                while (i < text.Length && !char.IsWhiteSpace(text[i]) &&
!IsDelimiter(text[i].ToString()))
                {
                    char ch = text[i];

                    if (char.IsLetterOrDigit(ch) || ch == '_')
                    {
                        cleaned += ch;
                    }
                    else
                    {
                        errors.Add(ch.ToString()); // символ с ошибкой — в отдельный
список
                    }

                    i++;
```

```
        }


        if (cleaned.Length > 0)
        {
            if (keyValuePairs.ContainsKey(cleaned))
            {
                int code = keyValuePairs[cleaned];
                codes.Add(code);
                keywords.Add("keyword");
                keyword.Add(cleaned);
                myTokens.Add(new MyToken(  start, line, cleaned));
                prevcode = code;
            }
            else
            {
                codes.Add(13);
                keywords.Add("Id");
                keyword.Add(cleaned);
                myTokens.Add(new MyToken(  start, line, cleaned));
                prevcode = 13;
            }
        }
```

```
            continue;
        }

        // Прочие символы — ошибка
        if (c != '{' && c != '}' && c != ';')
        {
            errors.Add(c.ToString()); // ошибка только в errors
        }

        i++;
    }
}


private bool IsSymbol(char c)
{
    // Символы, которые могут входить в "грязные" идентификаторы
    return "@$.#%&*!?\"\\|:[]/+-=<>()".Contains(c);
}

private bool IsDelimiter(string s)
{
    return keyValuePairs.ContainsKey(s) &&
```

```
                (s.Length == 1 && "{};,".Contains(s));

        }



        private bool letter(string word)

        {

            if (char.IsDigit(word[0]))

                return false;



            if (word.All(c => char.IsLetterOrDigit(c) || c == '_') && !Regex.IsMatch(word,
"[А-Яа-я]"))

            {

                return true;

            }

            return false;

        }

    }

}
```

Перечисление Token

```
public enum Token { START, STRUCT, OPENBRACE, TYPE, ID, SEM, CLOSEBRACE,
END, ERROR }
```

Перечисление ERRORTYPE

```
public enum ERRORTYPE { DELETE, ADD, CHANGE, NONE, START,ADDEND }
```

Класс Parse

```
public class Parse
```

```
{
    public Token GetNeedToken(List<Token> tokencurrent)
    {
        if (tokencurrent.Count != 0)
        {
            if (tokencurrent[tokencurrent.Count - 1] == Token.STRUCT)
            {
                return Token.ID;
            }
            else if (tokencurrent[tokencurrent.Count - 1] == Token.ID)
            {
                if (tokencurrent[tokencurrent.Count - 2] == Token.TYPE)
                {
                    return Token.SEM;
                }
                else if (tokencurrent[tokencurrent.Count - 2] == Token.STRUCT)
                {
                    return Token.OPENBRACE;
                }
            }
            else if (tokencurrent[tokencurrent.Count - 1] == Token.TYPE)
            {
                return Token.ID;
```

```
        }
        else if (tokencurrent[tokencurrent.Count - 1] == Token.OPENBRACE)
        {
            return Token.CLOSEBRACE;
        }
        else if (tokencurrent[tokencurrent.Count - 1] == Token.CLOSEBRACE)
        {
            return Token.SEM;
        }
        else if (tokencurrent[tokencurrent.Count - 1] == Token.SEM)
        {
            if (tokencurrent[tokencurrent.Count - 2] == Token.CLOSEBRACE)
            {
                return Token.STRUCT;
            }
            else
            {
                return Token.CLOSEBRACE;
            }

        }
        else
        {
```

```
                return Token.ERROR;

            }

        }

        else

        {

            return Token.STRUCT;

        }

        return Token.ERROR;

    }

    public List<string> Errors = new List<string>();

    public List<Token> tokens = new List<Token>();

    public List<Token> current_tokens = new List<Token>();

    public int Current_Position = 0;

    public Parse(List<Token> tok)

    {

        tokens = tok;

    }

    int absolute = 0;

    public List<MyToken> myTokens = new List<MyToken>();

    public int Parser(ERRORTYPE error, int current_token, ref TokenReturn token, int
errorCount = 0, int errorx = 0)

    {
```

```
        if (error == ERRORTYPE.DELETE)

        {

            token.error.Add(new MyToken($"Удалено {myTokens[current_token].Value}", myTokens[current_token].Position, myTokens[current_token].Line));

            current_token++;

            errorCount++;

        }

        else if (error == ERRORTYPE.CHANGE)

        {

            token.error.Add(new MyToken($"Ожидалось \"{GetNeedToken(token.tokens)}\" получили \"{myTokens[current_token].Value}\"", myTokens[current_token].Position, myTokens[current_token].Line));

            token.tokens.Add(GetNeedToken(token.tokens));

            current_token++;

            errorCount++;

        }

        else if (error == ERRORTYPE.ADD)

        {

            token.error.Add(new MyToken($"Ожидалось \"{GetNeedToken(token.tokens)}\"", myTokens[current_token].Position, myTokens[current_token].Line));

            token.tokens.Add(GetNeedToken(token.tokens));

            errorCount++;

        }
```

```
        else if(error == ERRORTYPE.ADDEND){

            token.error.Add(new MyToken($"Ожидалось \"{GetNeedToken(token.tokens)}\"", myTokens[myTokens.Count-1].Position+3, myTokens[myTokens.Count-1].Line));

            token.tokens.Add(GetNeedToken(token.tokens));

            errorCount++;

        }

        else if (error == ERRORTYPE.NONE)

        {

            token.tokens.Add(tokens[current_token]);

            current_token++;

        }



        if (tokens.Count > current_token)

        {

            if (GetNeedToken(token.tokens) != tokens[current_token])

            {

                if ((GetNeedToken(token.tokens) == Token.CLOSEBRACE && tokens[current_token] == Token.TYPE) ||

                    ((GetNeedToken(token.tokens) == Token.CLOSEBRACE && tokens[current_token] == Token.TYPE)))

                {

                    return Parser(ERRORTYPE.NONE, current_token, ref token, errorCount);

                }
```

```
            else

            {

                int minErrors = int.MaxValue;



                var token1 = new TokenReturn(token.tokens, token.error);



                int errorsDelete = Parser(ERRORTYPE.DELETE, current_token, ref token1,
errorCount);

                if (errorsDelete < minErrors)

                    minErrors = errorsDelete;

                var token2 = new TokenReturn(token.tokens, token.error);

                int errorsChange = Parser(ERRORTYPE.CHANGE, current_token, ref
token2, errorCount);

                if (errorsChange <= minErrors)

                {

                    minErrors = errorsChange;

                    token1 = new TokenReturn(token2.tokens, token2.error);

                }



                var token3 = new TokenReturn(token.tokens, token.error);

                int errorsAdd = Parser(ERRORTYPE.ADD, current_token, ref token3,
errorCount);

                if (errorsAdd <= minErrors)

                {
```

```
                minErrors = errorsAdd;
                token1 = new TokenReturn(token3.tokens, token3.error);
            }
            token = token1;
            return minErrors;
        }
    }
    else
    {
        return Parser(ERRORTYPE.NONE, current_token, ref token, errorCount);
    }
}
else
{
    int ob = 0;
    int obp =0;
    for (int i = 0; i < token.tokens.Count; i++)
    {
        if (token.tokens[i] == Token.OPENBRACE)
        {
            ob++;
            obp = 1;
        }
```

```
            if (token.tokens[i] == Token.CLOSEBRACE)

            {

                ob--;

            }



        }

        if (ob > 0 || obp == 0)

        {

            var token3 = new TokenReturn(token.tokens, token.error);

            int errorsAdd = Parser(ERRORTYPE.ADDEND, current_token, ref token3,
errorCount);

            token = token3;

            return errorsAdd;

        }

        if (token.tokens[token.tokens.Count-1] == Token.CLOSEBRACE)

        {

            var token3 = new TokenReturn(token.tokens, token.error);

            int errorsAdd = Parser(ERRORTYPE.ADDEND, current_token, ref token3,
errorCount);

            token = token3;

            return errorsAdd;

        }
```

```
        }

        return errorCount;

    }

}
```

Класс MyToken

```
public class MyToken

{

    public string Value { get; set; }

    public int Line { get; set; }

    public int Position { get; set; }

    public string mess { get; set; }

    public MyToken(int pos, int lin, string val)

    {

        Value = val;

        Line = lin;

        Position = pos;

    }

    public MyToken(string mess, int pos, int lin)

    {

        Line = lin;

        Position = pos;

        this.mess = mess;

    }

}
```

Класс TokenReturn

```
public class TokenReturn
{
    public List<Token> tokens { get; set; } = new List<Token>();
    public List<MyToken> error { get; set; } = new List<MyToken>();

    public TokenReturn() { }

    public TokenReturn(List<Token> tokens, List<MyToken> error)
    {
        this.tokens = new List<Token>(tokens);
        this.error = new List<MyToken>(error);
    }
}
```