**Листинг программы (Анализирующей части)**

Класс Token

|  |
| --- |
| /// <summary> Коды токенов </summary>  public enum CODE  {  ERROR, // Ошибка  INT, // int  UINT, // uint  FLOAT32, // float32  FLOAT64, // float64  FUNC, // func  RETURN, // return  IDENTIFIER, // Идентификатор  DELIMITER, // Разделитель (пробел)  PLUS, // +  MINUS, // -  MULTIPLY, // \*  DIVIDE, // /  LBRACE, // {  RBRACE, // }  LPAREN, // (  RPAREN, // )  COMMA, // ,  UNSIGNED\_INT, // Беззнаковое целое число  END // ; (конец оператора)  }  /// <summary> Класс токена </summary>  public class Token  {  public CODE Code { get; set; }  private string \_token;  private int \_line;  private int \_startColumn;  private int \_endColumn;  public Token(CODE code, string token, int line, int startColumn, int endColumn, int startIndex, int endIndex)  {  Code = code;  \_token = token;  \_line = line;  \_startColumn = startColumn;  \_endColumn = endColumn;  StartIndex = startIndex;  EndIndex = endIndex;  }  public string TokenValue => \_token;  public int Line => \_line;  public int StartColumn => \_startColumn;  public int EndColumn => \_endColumn;  public int StartIndex { get; set; }  public int EndIndex { get; set; }  public override string ToString()  {  return $"Code: {Enum.GetName(typeof(CODE), Code),-12} | Token: {\_token,-15} | Line: {\_line,3} | Start: {\_startColumn,3} | End: {\_endColumn,3}";  }  } |

Класс BaseParser

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Text;  using System.Threading.Tasks;  namespace Compiler.Analysis  {  /// <summary> Класс для представления ошибок </summary>  public class ErrorEntry  {  public string Message { get; set; }  public int Line { get; set; }  public int Column { get; set; }  public Token Token { get; set; }  public int Priority { get; set; }  public ErrorEntry(string message, Token token, int priority)  {  Message = message;  Line = token.Line;  Column = token.StartColumn;  Token = token;  Priority = priority;  }  public override string ToString()  {  return Message + " | " + Line.ToString() + " | " + Column.ToString();  }  }  /// <summary> Делегат для проверки условия </summary>  public delegate bool Condition();  /// <summary> Базовый класс для всех парсеров, кроме главного </summary>  class BaseParser  {  // Используются только в парсере  protected List<Token> \_tokens;  protected int \_index;  protected List<ErrorEntry> \_errors;  protected string \_text;  protected int Index { get { return \_index; } set { \_index = value; } }  protected Token Token { get { return \_tokens[Index]; } }  protected bool IsNotEndList => \_index < \_tokens.Count - 1;  // Используются другими парсерами, как результат работы.  public int NextPosition { get; protected set; }  public List<Token> Tokens { get { return \_tokens; } }  public List<ErrorEntry> Errors { get { return \_errors; } }  // Мы не смотрим уже на просмотренные токены и не меняем изначальный список  // Убираем токены с прошлого парсера  public BaseParser(List<Token> tokens, int pos, string text)  {  \_text = text;  \_tokens = new List<Token>() { };  \_errors = new List<ErrorEntry>() { };  \_index = 0;  // Скопируем только нужные нам токены  for (int i = 0; i < tokens.Count; i++)  {  if (i >= pos)  {  \_tokens.Add(tokens[i]);  }  }  \_text = text;  }  ///<summary> Добавление ошибок в список ошибок </summary>  protected void AddError(string message, int priority = 0, Token token = null)  {  \_errors.Add(new ErrorEntry(message, token ?? Token, priority));  }  ///<summary> Сбор ошибок </summary>  protected string CollectError(int startPos, int endPos, bool moveIndex = true, Condition condition = null)  {  // Устанавливаем condition по умолчанию (всегда true)  condition = condition ?? (() => true);  SkipSpace();  string errorValue = "";  Token startToken = \_tokens[startPos], endToken = \_tokens[startPos];  for (int i = startPos; i < endPos && condition(); i++)  {  if (moveIndex) Index++;  endToken = \_tokens[i];  errorValue += \_tokens[i].TokenValue;  }  return errorValue;  }  // Вспомогательный функционал  ///<summary> Пропуск пробелов </summary>  protected void SkipSpace()  {  if (Token.Code == CODE.DELIMITER) Index++;  }  ///<summary> Проверяет, что заданный токен является Типом </summary>  protected bool Type(Token token)  {  if (token.Code == CODE.INT || token.Code == CODE.UINT || token.Code == CODE.FLOAT32 || token.Code == CODE.FLOAT64) return true;  else return false;  }  ///<summary> Проверяет, что текущий токен является Типом </summary>  protected bool Type()  {  if (Token.Code == CODE.INT || Token.Code == CODE.UINT || Token.Code == CODE.FLOAT32 || Token.Code == CODE.FLOAT64) return true;  else return false;  }  }  } |

Класс FuncHeadParser

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Text;  using System.Threading.Tasks;  namespace Compiler.Analysis  {  class FuncHeadParser : BaseParser  {  public FuncHeadParser(List<Token> tokens, int pos, string text) : base(tokens, pos, text)  {  }  public bool FindLparen { get; set; } = false;  public bool FindFunc { get; set; } = false;  public int LparenPos { get; set; }  public int FuncPos { get; set; }  private int endPos = 0;  public List<ErrorEntry> Parse()  {  BeforeParse();  Func();  return \_errors;  }  private void BeforeParse()  {  \_tokens.Add(new Token(CODE.DELIMITER, " ", \_tokens[\_tokens.Count - 1].Line, \_tokens[\_tokens.Count - 1].EndColumn + 1, \_tokens[\_tokens.Count - 1].EndColumn + 1, \_tokens[\_tokens.Count - 1].EndIndex + 1, \_tokens[\_tokens.Count - 1].EndIndex + 1));  while (Index < \_tokens.Count - 1 && Token.Code != CODE.LPAREN)  {  Index++;  }  if (Token.Code == CODE.LPAREN)  {  FindLparen = true;  LparenPos = Index;  }  Index = 0;  while (Index < \_tokens.Count - 1 && Token.Code != CODE.FUNC)  {  Index++;  }  if (Token.Code == CODE.FUNC)  {  FindFunc = true;  FuncPos = Index;  }  if (LparenPos > 10 || LparenPos < FuncPos)  {  FindLparen = false;  }  if (FindLparen) endPos = LparenPos;  else endPos = \_tokens.Count;  Index = 0;  NextPosition = endPos + 1;  }  private void Func()  {  if (!FindFunc)  {  AddError($"Ожидалось: ключевое слово func.");  Index = 1;  }  else  {  if (FuncPos != 0)  {  AddError($"Ожидалось: ключевое слово func.");  }  Index = FuncPos + 1;  if (Token.Code != CODE.DELIMITER)  {  AddError($"Ожидалось: пробел после func.");  }  Index++;  }  FuncName();  }  private void FuncName()  {  SkipSpace();  string errorValue;  Condition condition = () => Token.Code != CODE.IDENTIFIER;  Token startToken = Token;  // Проверяем имя функции  errorValue = CollectError(Index, endPos, true, condition);  if (Token.Code != CODE.IDENTIFIER)  {  AddError($"Ожидалось: идентификатор (имя функции).", 0,startToken);  }  else  {  if (errorValue != "") AddError($"Лишняя последовательность символов.", 0, startToken);  }  AfterFuncName();  }  private void AfterFuncName()  {  Index++;  SkipSpace();  string errorValue;  Token startToken = Token;  if (FindLparen)  {  errorValue = CollectError(Index, endPos);  if (errorValue != "") AddError($"Лишняя последовательность символов.", 0, startToken);  }  else  {  AddError($"Ожидалось: ( после имени функции.", 1);  NextPosition = Index;  }  }  }  } |

Класс ArgumentsParser

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Text;  using System.Threading.Tasks;  namespace Compiler.Analysis  {  enum ArgumentsStatus  {  FIND\_RPAREN, // Нашли )  FIND\_LBRACE, // Нашли {  FIND\_RETURN, // Нашли return  FIND\_END, // Нашли ;  NOT\_FIND // Ничего не нашли  }  /// <summary>  /// Класс для парсинга аргументов.  /// Помимо ошибок непосредственно в аргументах он выделяет так же и ошибки в оконцовке аргументов.  /// </summary>  class ArgumentsParser : BaseParser  {  public ArgumentsParser(List<Token> tokens, int pos, string text) : base(tokens, pos, text) { }  public ArgumentsStatus Status { get; set; }  public int EndPos { get; set; }  public bool FindAgrsType { get; set; } = false;  int TypePos { get; set; }  // Местоположение различных важных токенов  bool FindRparen{ get; set; } = false;  bool FindLbrace { get; set; } = false;  bool FindReturn { get; set; } = false;  bool FindEnd { get; set; }=false;  int RparenPos { get; set; }  int LbracePos { get; set; }  int ReturnPos { get; set; }  int ENDPos { get; set; }  public List<ErrorEntry> Parse()  {  if (\_tokens.Count == 0) return \_errors;  FindArgumentsEnd();  Arguments();  return \_errors;  }  /// <summary> Функция, которая ищет конец наших аргументов </summary>  private void FindArgumentsEnd()  {  Status = ArgumentsStatus.NOT\_FIND;  Index++;  EndPos = Index;  // Постараемся найти конец наших аргументов. Сначала без ошибок - )  while (IsNotEndList && Token.Code != CODE.RPAREN && Token.Code != CODE.LBRACE && Token.Code != CODE.RETURN && Token.Code != CODE.END)  {  Index++;  }  if (Token.Code == CODE.RPAREN)  {  EndPos = Index;  Status = ArgumentsStatus.FIND\_RPAREN;  FindRparen = true;  RparenPos = Index;  }  Index = 0;  //Если не вышло ищем {  while (IsNotEndList && Token.Code != CODE.LBRACE)  {  Index++;  }  if (Token.Code == CODE.LBRACE)  {  EndPos = Index;  Status = ArgumentsStatus.FIND\_LBRACE;  FindLbrace = true;  LbracePos = Index;  }  Index = 0;  //Если не вышло ищем return  while (IsNotEndList && Token.Code != CODE.RETURN)  {  Index++;  }  if (Token.Code == CODE.RETURN)  {  EndPos = Index;  Status = ArgumentsStatus.FIND\_RETURN;  FindReturn = true;  ReturnPos = Index;  }  Index = 0;  while (IsNotEndList && Token.Code != CODE.END)  {  Index++;  }  if (Token.Code == CODE.END)  {  EndPos = Index;  Status = ArgumentsStatus.FIND\_END;  FindEnd = true;  ENDPos = Index;  }    if(FindRparen) EndPos = RparenPos;  else if (FindLbrace) EndPos = LbracePos;  else if (FindReturn) EndPos = ReturnPos;  else if (FindEnd) EndPos = EndPos;  else EndPos = \_tokens.Count - 1;  }  // Проверка аргументов  private void Arguments()  {  // Начитаем перебирать аргументы  CODE lastCode = CODE.COMMA;  string errorValue;  Index = 0;  Token startErrorToken;  while (Index < EndPos && !Type())  {  errorValue = "";  SkipSpace();  // Значит что мы сейчас ждем индификатор  if (lastCode == CODE.COMMA)  {  if (Token.Code == CODE.IDENTIFIER)  {  lastCode = CODE.IDENTIFIER;  Index++;  continue;  }  else  {  startErrorToken = Token;  errorValue = CollectError(Index, EndPos, true, () => !Type() && Token.Code != CODE.IDENTIFIER && Token.Code != CODE.COMMA);  AddError($"Ожидалось: индификатор.", 0, startErrorToken);  if (Type())  {  FindAgrsType = true;  TypePos = Index;  break;  }  else if (Token.Code == CODE.IDENTIFIER)  {  lastCode = CODE.IDENTIFIER;  Index++;  }  else if (Token.Code == CODE.COMMA)  {  lastCode = CODE.COMMA;  Index++;  }  }  }  else if (lastCode == CODE.IDENTIFIER)  {  if (Token.Code == CODE.COMMA)  {  lastCode = CODE.COMMA;  Index++;  continue;  }  if (Type())  {  FindAgrsType = true;  TypePos = Index;  Index++;  break;  }  else  {  startErrorToken = Token;  errorValue = CollectError(Index, EndPos, true, () => !Type() && Token.Code != CODE.IDENTIFIER && Token.Code != CODE.COMMA);  AddError($"Ожидалось: запятая или тип.", 0, startErrorToken);  if (Type())  {  FindAgrsType = true;  TypePos = Index;  Index++;  break;  }  else if (Token.Code == CODE.IDENTIFIER)  {  lastCode = CODE.IDENTIFIER;  Index++;  }  else if (Token.Code == CODE.COMMA)  {  lastCode = CODE.COMMA;  Index++;  }  }  }  }  if (lastCode != CODE.IDENTIFIER) AddError($"Ожидалось: индификатор.", 0, \_tokens[Index]);  NextPosition = Index + 1;  if (IsNotEndList && Type())  {  FindAgrsType = true;  TypePos = Index;  Index++;  }  if (FindAgrsType && TypePos < EndPos - 1)  {  SkipSpace();  startErrorToken = \_tokens[TypePos + 1];  errorValue = CollectError(Index, EndPos);  AddError($"Лишняя последовательность символов.", 0, startErrorToken);  }  else if (!FindAgrsType)  {  Index = EndPos;  SkipSpace();  errorValue = CollectError(Index, EndPos);  startErrorToken = Token;  AddError($"Ожидалось: тип аргументов.", 1, startErrorToken);  }  EndArguments();  }  // Проверка конца агрументов  private void EndArguments()  {  Token startErrorToken;  string errorValue;  // Проверяем конец и пытаемся найти переход на выражение    NextPosition = Index + 1;  if (FindReturn) NextPosition = ReturnPos + 1;  else if (FindLbrace) NextPosition = LbracePos + 1;  else if (FindRparen) NextPosition = RparenPos + 1;  else NextPosition = Index + 1;    if (!FindRparen) AddError("Ожидалось: ).", 3, \_tokens[FindAgrsType ? TypePos + 1 : Index]);    if (!FindLbrace && !FindRparen) AddError("Ожидалось: {.", 2, \_tokens[FindAgrsType ? TypePos + 2 : Index]);  else if (EndPos + 1 < \_tokens.Count && !FindLbrace) AddError("Ожидалось: {.", 2, \_tokens[FindAgrsType ? EndPos + 1 : Index + 1]);  if (!FindReturn)  {  int buf = NextPosition;  if (!FindLbrace && !FindRparen && FindAgrsType) buf = TypePos + 1;  Index = buf;  if (IsNotEndList) AddError("Ожидалось: ключевое слово return.", 3, \_tokens[Index]);  Index++;  NextPosition = Index;  }  if (!FindEnd)  {    AddError("Ожидалось: ;", 0, \_tokens[\_tokens.Count - 1]);  }  if (FindRparen && FindLbrace && LbracePos - RparenPos != 1) AddError("Лишняя последовательность символов.", 0, \_tokens[RparenPos + 1]);  if (FindLbrace && FindReturn && ReturnPos - LbracePos != 1) AddError("Лишняя последовательность символов.", 0, \_tokens[LbracePos + 1]);  if (NextPosition >= \_tokens.Count) NextPosition = \_tokens.Count - 1;  }  }  } |

Класс ExpressionParser

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Text;  using System.Threading.Tasks;  namespace Compiler.Analysis  {  class ExpressionParser : BaseParser  {  enum StatuExp { OPERTION, OPERAND, LPAREN, RPAREN }  public ExpressionParser(List<Token> tokens, int pos, string text) : base(tokens, pos, text)  {  }  public List<ErrorEntry> Parse()  {  SkipSpace();  Console.WriteLine(Token);  EXPRESSION();  return \_errors;  }  public bool Operation()  {  if (Token.Code == CODE.PLUS || Token.Code == CODE.MINUS || Token.Code == CODE.MULTIPLY || Token.Code == CODE.DIVIDE) return true;  else return false;  }  public void EXPRESSION()  {  int countLPAREN = 0;  int countRPAREN = 0;  int ENDPos = 0;  int RBRACEPos = 0;  int endPos = 0;  bool findEND = false;  bool findRBRACE = false;  while (IsNotEndList && Token.Code != CODE.RBRACE && Token.Code != CODE.END)  {  Index++;  }  if (IsNotEndList && Token.Code == CODE.RBRACE)  {  findRBRACE = true;  RBRACEPos = Index;  }  Index = 0;  while (IsNotEndList && Token.Code != CODE.END)  {  Index++;  }  if (Token.Code == CODE.END)  {  findEND = true;  ENDPos = Index;  }  Index = 0;  if (findRBRACE) endPos = RBRACEPos;  else if (findEND) endPos = ENDPos;  else endPos = \_tokens.Count;  StatuExp status = StatuExp.OPERTION;  while (Index < endPos)  {  SkipSpace();  if (Index >= endPos) break;  if (status == StatuExp.OPERTION)  {  if (Token.Code == CODE.LPAREN)  {  countLPAREN++;  status = StatuExp.LPAREN;  Index++;  continue;  }  else if (Token.Code == CODE.IDENTIFIER || Token.Code == CODE.UNSIGNED\_INT)  {  status = StatuExp.OPERAND;  Index++;  continue;  }  else  {  string errorValue = "";  while (Index < endPos && !Operation() && Token.Code != CODE.LPAREN && Token.Code != CODE.RPAREN && Token.Code != CODE.IDENTIFIER && Token.Code != CODE.UNSIGNED\_INT)  {  errorValue += Token.TokenValue;  Index++;  }  AddError($"Ожидалось: операнд");  if (Operation()) { status = StatuExp.OPERTION; Index++; continue; }  if (Token.Code == CODE.LPAREN) { status = StatuExp.LPAREN; Index++; countLPAREN++; continue; }  if (Token.Code == CODE.RPAREN) { status = StatuExp.RPAREN; Index++; countRPAREN++; continue; }  if (Token.Code == CODE.IDENTIFIER || Token.Code == CODE.UNSIGNED\_INT) { status = StatuExp.OPERAND; Index++; continue; }  }  }  else if (status == StatuExp.LPAREN)  {  if (Token.Code == CODE.IDENTIFIER || Token.Code == CODE.UNSIGNED\_INT)  {  status = StatuExp.OPERAND;  Index++;  continue;  }  else  {  string errorValue = "";  while (Index < endPos && !Operation() && Token.Code != CODE.LPAREN && Token.Code != CODE.RPAREN && Token.Code != CODE.IDENTIFIER && Token.Code != CODE.UNSIGNED\_INT)  {  errorValue += Token.TokenValue;  Index++;  }  AddError($"Ожидалось: операнд.");  if (Operation()) { status = StatuExp.OPERTION; Index++; continue; }  if (Token.Code == CODE.LPAREN) { status = StatuExp.LPAREN; Index++; countLPAREN++; continue; }  if (Token.Code == CODE.RPAREN) { status = StatuExp.RPAREN; Index++; countRPAREN++; continue; }  if (Token.Code == CODE.IDENTIFIER || Token.Code == CODE.UNSIGNED\_INT) { status = StatuExp.OPERAND; Index++; continue; }  }  }  else if (status == StatuExp.OPERAND)  {  if (Operation())  {  status = StatuExp.OPERTION;  Index++;  continue;  }  else if (Token.Code == CODE.RPAREN)  {  status = StatuExp.RPAREN;  countRPAREN++;  Index++;  continue;  }  else  {  string errorValue = "";  while (Index < endPos && !Operation() && Token.Code != CODE.LPAREN && Token.Code != CODE.RPAREN && Token.Code != CODE.IDENTIFIER && Token.Code != CODE.UNSIGNED\_INT)  {  errorValue += Token.TokenValue;  Index++;  }  AddError($"Ожидалось: арифметическая операция или ).");  if (Operation()) { status = StatuExp.OPERTION; Index++; continue; }  if (Token.Code == CODE.LPAREN) { status = StatuExp.LPAREN; Index++; countLPAREN++; continue; }  if (Token.Code == CODE.RPAREN) { status = StatuExp.RPAREN; Index++; countRPAREN++; continue; }  if (Token.Code == CODE.IDENTIFIER || Token.Code == CODE.UNSIGNED\_INT) { status = StatuExp.OPERAND; Index++; continue; }  }  }  else  {  if (Operation())  {  status = StatuExp.OPERTION;  Index++;  continue;  }  else  {  string errorValue = "";  while (Index < endPos && !Operation() && Token.Code != CODE.LPAREN && Token.Code != CODE.RPAREN && Token.Code != CODE.IDENTIFIER && Token.Code != CODE.UNSIGNED\_INT)  {  errorValue += Token.TokenValue;  Index++;  }  AddError($"Ожидалось: арифметическая операция.");  if (Operation()) { status = StatuExp.OPERTION; Index++; continue; }  if (Token.Code == CODE.LPAREN) { status = StatuExp.LPAREN; Index++; countLPAREN++; continue; }  if (Token.Code == CODE.RPAREN) { status = StatuExp.RPAREN; Index++; countRPAREN++; continue; }  if (Token.Code == CODE.IDENTIFIER || Token.Code == CODE.UNSIGNED\_INT) { status = StatuExp.OPERAND; Index++; continue; }  }  }  }  if (status != StatuExp.OPERAND && status != StatuExp.RPAREN)  {  AddError($"Ожидалось: операнд или ).", 0, \_tokens[Index]);  }  if (countLPAREN > countRPAREN)  {  AddError($"Ожидалось: ).");  }  if (countRPAREN > countLPAREN)  {  int \_ = Index;  Index = 0;  AddError($"Ожидалось: (.");  Index = \_;  }  if (!findRBRACE)  {  AddError("Ожидалось: }.");  return;  }  if (findEND && findRBRACE && ENDPos - RBRACEPos != 1)  {  AddError($"Лишняя последовательность символов." , 0, \_tokens[RBRACEPos + 1]);  }  if (findRBRACE && !findEND && \_tokens[RBRACEPos + 1].Code != CODE.DELIMITER)  {  AddError($"Лишняя последовательность символов.", 0, \_tokens[RBRACEPos + 1]);  }  }  }  } |

Класс Parser

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Text;  using System.Threading.Tasks;  namespace Compiler.Analysis  {  /// <summary>  /// Является главным парсером. По сути просто вызывает сканер, удаляет ERROR и вызывает другие парсеры.  /// </summary>  public class Parser  {  private List<Token> \_tokens;  private List<ErrorEntry> \_errors;  protected int \_index;  protected string \_text;  protected int Index { get { return \_index; } set { \_index = value; } }  protected Token Token { get { return \_tokens[Index]; } }  public Parser(string text)  {  \_text = text;  \_errors = new List<ErrorEntry>() { };  \_index = 0;  }  public List<ErrorEntry> Parse()  {  /\*  string[] parts = \_text.Split(';').Select((s, i) => i < \_text.Count(c => c == ';') ? s + ";" : s).ToArray();  for (int i = 0; i < parts.Length; i++)  {  \_text = parts[i];  ParseOneFunc();  }  \*/  var scaner = new Scaner();  \_tokens = scaner.Scan(\_text);  RemoveUnknownSymbols();  List<Token> tokens = new List<Token>() { };  int i = 0;  while (i < \_tokens.Count)  {  while (i < \_tokens.Count && \_tokens[i].Code != CODE.END)  {  tokens.Add(\_tokens[i]);  i++;  }  if (i < \_tokens.Count) tokens.Add(\_tokens[i]);  ParseOneFunc(tokens);  tokens.Clear();  i++;  }    return \_errors;  }  public void ParseOneFunc(List<Token> tokens)  {  var funcHeadParser = new FuncHeadParser(tokens, 0, \_text);  try  {  funcHeadParser.Parse();  }  catch { }  finally  {  AddErrorsList(funcHeadParser.Errors);  }  var argumentsParser = new ArgumentsParser(funcHeadParser.Tokens, funcHeadParser.NextPosition, \_text);  try  {  argumentsParser.Parse();  }  catch { }  finally  {  AddErrorsList(argumentsParser.Errors);  }  var expressionParser = new ExpressionParser(argumentsParser.Tokens, argumentsParser.NextPosition, \_text);  try  {  expressionParser.Parse();  }  catch { }  finally  {  AddErrorsList(expressionParser.Errors);  }  SortErrors();  ClearRepearError();  foreach (var er in \_errors)  {  Console.WriteLine(er);  }  Console.WriteLine("\n\nТОКЕНЫ:");  foreach (var token in \_tokens)  {  Console.WriteLine(token);  }  }  private void SortErrors()  {  \_errors = \_errors.OrderBy(e => e.Line).ThenBy(e => e.Column).ToList();  }  private void ClearRepearError()  {  int i = 0;  while (i < \_errors.Count - 1)  {  if (\_errors[i].Token == \_errors[i + 1].Token)  {  if (\_errors[i].Priority > \_errors[i + 1].Priority)  {  \_errors.RemoveAt(i + 1);  }  else  {  \_errors.RemoveAt(i);  }  continue;  }  i++;  }  }  // Обработаем все невалидные токены и удалим их  private void RemoveUnknownSymbols()  {  int i = 0;  while (i < \_tokens.Count)  {  if (\_tokens[i].Code == CODE.ERROR)  {  int startIndex = i;  string unknownSequence = \_tokens[i].TokenValue;  i++;  while (i < \_tokens.Count && \_tokens[i].Code == CODE.ERROR)  {  unknownSequence += \_tokens[i].TokenValue;  i++;  }  AddError($"Неожиданная последовательность символов: '{unknownSequence}'", 0, \_tokens[startIndex]);  \_tokens.RemoveRange(startIndex, i - startIndex);  i = startIndex;  }  else  {  i++;  }  }  i = 0;  // Следующий этап - склеиваем ошибочные токены и числа. Возможно мы найдем индификатор?  while (i < \_tokens.Count - 1)  {  if (\_tokens[i].Code == CODE.IDENTIFIER && \_tokens[i + 1].Code == CODE.IDENTIFIER ||  \_tokens[i].Code == CODE.IDENTIFIER && \_tokens[i + 1].Code == CODE.UNSIGNED\_INT)  {  var token = new Token(CODE.IDENTIFIER, \_tokens[i].TokenValue + \_tokens[i + 1].TokenValue,  \_tokens[i].Line, \_tokens[i].StartColumn, \_tokens[i + 1].EndColumn, \_tokens[i].StartIndex, \_tokens[i + 1].EndIndex);  \_tokens.RemoveAt(i);  \_tokens.RemoveAt(i);  \_tokens.Insert(i, token);  continue;  }  if (\_tokens[i].Code == CODE.UNSIGNED\_INT && \_tokens[i + 1].Code == CODE.UNSIGNED\_INT)  {  var token = new Token(CODE.UNSIGNED\_INT, \_tokens[i].TokenValue + \_tokens[i + 1].TokenValue,  \_tokens[i].Line, \_tokens[i].StartColumn, \_tokens[i + 1].EndColumn, \_tokens[i].StartIndex, \_tokens[i + 1].EndIndex);  \_tokens.RemoveAt(i);  \_tokens.RemoveAt(i);  \_tokens.Insert(i, token);  continue;  }  if (isWord(\_tokens[i].Code))  {  switch (\_tokens[i].TokenValue)  {  case "func":  \_tokens[i].Code = CODE.FUNC; break;  case "int":  \_tokens[i].Code = CODE.INT; break;  case "uint":  \_tokens[i].Code = CODE.UINT; break;  case "float32":  \_tokens[i].Code = CODE.FLOAT32; break;  case "float64":  \_tokens[i].Code = CODE.FLOAT64; break;  case "return":  \_tokens[i].Code = CODE.RETURN; break;  }  }  i++;  }  }  ///<summary> Добавление ошибок в список ошибок </summary>  protected void AddError(string message, int priority = 0, Token token = null)  {  \_errors.Add(new ErrorEntry(message, token ?? Token, priority));  }  private void AddErrorsList(List<ErrorEntry> ers)  {  foreach (var er in ers)  {  \_errors.Add(er);  }  }  private bool isWord(CODE code)  {  if (code == CODE.FUNC || code == CODE.INT || code == CODE.IDENTIFIER || code == CODE.UINT ||  code == CODE.FLOAT32 || code == CODE.FLOAT64 || code == CODE.RETURN) return true;  return false;  }  }  } |