Presentation

Team

Our team is **Tourists**

Team members are **Danila Khrankou & Tsimafei Kurstak**

Impementation

Language: **ProjectI (Imperative)**

Implementation language: **C#**

Tool: **gppg-based parser**

Target platform: **WASM**

Tokens

We represent tokens in **OO architecture**

```
// Lexer/Token.cs
public class Span(int line, int start, int end)
{
    public int Line { get; } = line;
    public int Start { get; } = start;
    public int End { get; } = end;

    public override string ToString() => $"{Line}:{Start}-{End}";
}

public abstract class Token(TokenType type, string lexeme, Span span)
{
    public TokenType Type { get; } = type;
    public Span Span { get; } = span;
    public string Lexeme { get; } = lexeme;

    public override string ToString() => $"{Type}: {Lexeme}, \tspan: {Span}";
}

public class IdentifierToken(string lexeme, Span span) : Token(TokenType.tkIdentifier,
{
    public string Name => Lexeme;
}

public class IntegerToken(string lexeme, long value, Span span) : Token(TokenType.tkInt
{
    public long Value { get; } = value;
}

public class RealToken(string lexeme, double value, Span span) : Token(TokenType.tkReal
{
    public double Value { get; } = value;
}

public class BooleanToken(string lexeme, bool value, Span span) : Token(TokenType.tkBoo
{
    public bool Value { get; } = value;
}

public class RecordAccessToken(string lexeme, Span span) : Token(TokenType.tkRecordAcce
{
    public string[] FieldPath { get; } = lexeme.Split('.');
```

```
    public string RecordName => FieldPath[0];
    public string FieldName => FieldPath[^1];
    public bool IsNested => FieldPath.Length > 2;
}

public class SimpleToken(TokenType type, string lexeme, Span span) : Token(type, lexeme
{
}
```

Token Definitions

Keywords and operators **are stored in a hash table**, where the **key** is the **lexeme** and the **value** is the **token code**.

```
// Lexer/TokenType.cs
public enum TokenType
{
    tkIntegerLiteral,
    tkRealLiteral,
    tkBoolLiteral,
    // String,
    tkIdentifier,

    tkVar,
    tkRealKeyword,
    tkBoolKeyword,
    ...
}
```

```
// Lexer/TokenDefenitions.cs
public static class TokenDefinitions
{
    public static readonly Dictionary<string, TokenType> Keywords = new()
    {
        {"var", TokenType.tkVar},
        {"real", TokenType.tkRealKeyword},
        {"boolean", TokenType.tkBoolKeyword},
        {"integer", TokenType.tkIntegerKeyword},
        {"type", TokenType.tkType},
        {"is", TokenType.tkIs},
        ...
    }
    public static readonly Dictionary<string, TokenType> Operators = new()
    {
        {"+", TokenType.tkPlus},
        {"-", TokenType.tkMinus},
        {"*", TokenType.tkMultiply},
        {"/", TokenType.tkDivide},
        {"%", TokenType.tkMod},
        {":=", TokenType.tkAssign},
        ...
    }
}
```

Entry point

At the moment the entry point looks like this. In the future we will add loading the file into the buffer and will accept the path to the file via command line arguments

```
string input = File.ReadAllText("./test.imperative");
var lexer = new LexerClass(input);
var tokens = lexer.NextToken();
while (tokens.Type != TokenType.tkEOF)
{
    Console.WriteLine(tokens);
    tokens = lexer.NextToken();
}
```

Lexer

```
public class LexerClass
{
    private readonly string _source;
    private int _position;
    private int _line;
    private int _column;
    private char _currentChar;
    ...
    public Token NextToken()
    {
        while (_currentChar != '\0')
        {
            if (_currentChar == '\n')
            {
                int oldColumn = _column;
                _column = 1;
                Move();
                return new SimpleToken(TokenType.tkEOL, "\\n", new Span(_line++, oldCol
            }

            if (char.IsWhiteSpace(_currentChar))
            {
                SkipWhitespace();
                continue;
            }

            if (char.IsDigit(_currentChar))
            {
                return ParseNumber(_line, _column);
            }

            if (char.IsLetter(_currentChar))
            {
                return ParseIdentifierOrKeyword(_line, _column);
            }

            if (TokenDefinitions.Operators.ContainsKey(_currentChar.ToString()))
            {
                return ReadOperator(_line, _column);
            }
```

```
            Span span = new(_line, _column, _column);
            string invalidLexeme = _currentChar.ToString();
            Move();
            return new SimpleToken(TokenType.tkInvalid, invalidLexeme, span);
        }
        return new SimpleToken(TokenType.tkEOF, "", new Span(_line, _column, _column));
    }
}
```