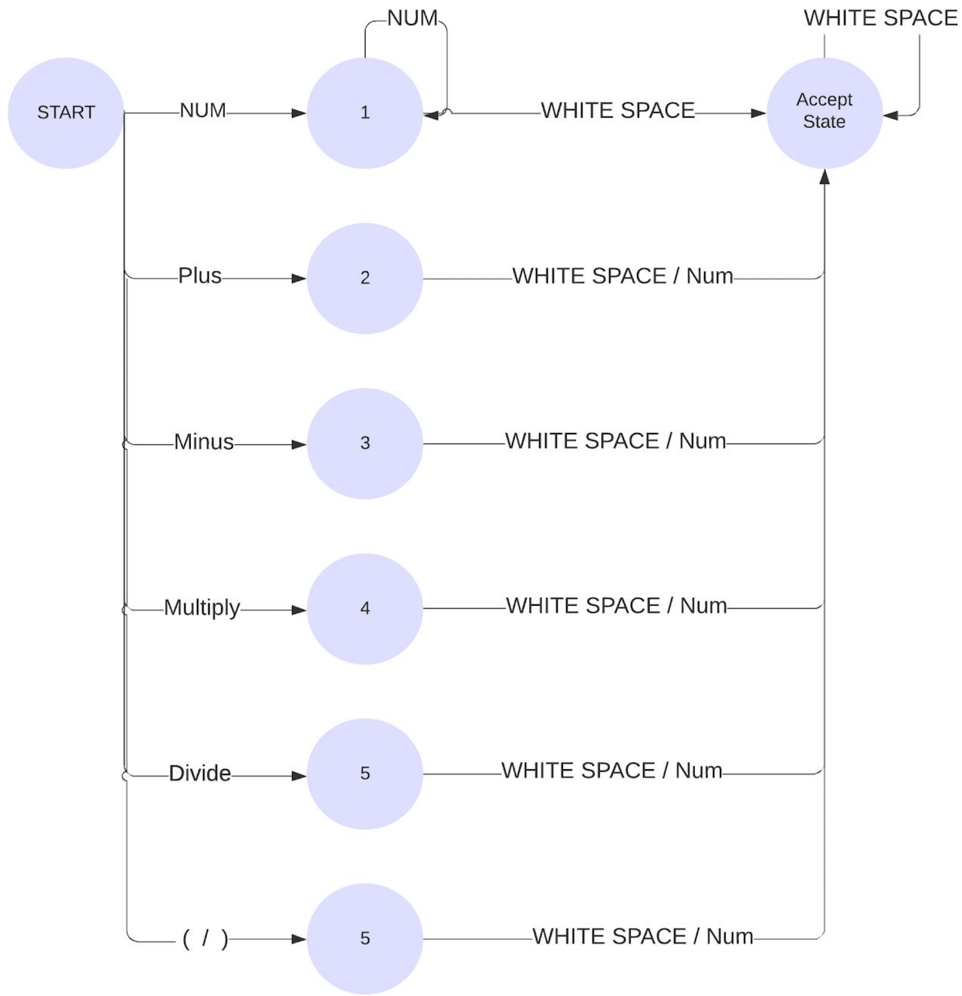
Tiny interpreter (Parser for arithmetic operations)

Lexer

The lexer is all about divide the input into tokens, each is one type of the following tokens :

- Numbers	{0-9}*
- Plus	{ + }
- Minus	{ - }
- Multiply	{ * }
- Divide	{ / }
- Open Parenthesis	{ (}
- Close Parenthesis	{) }
- End Of File	{ \0 }
- White Space	{ }*
- Bad Token	{ $ / @ / ! / % / ^ …. etc}*

The following DFA represents the lexer, which accepts only these tokens :



Lexer Implementation

Any token will be represented as one kind that we mentioned before in the following enum :

```
enum SyntaxKind{
        2 references
        NumberToken,
        2 references
        WhitespaceToken,
        3 references
        PlusToken,
        3 references
        MinusToken,
        3 references
        MultiplyToken,
        3 references
        DivideToken,
        2 references
        OpenParenthesisToken,
        2 references
        CloseParenthesisToken,
        3 references
        BadToken,
        4 references
        EndOfFileToken,
        1 reference
        NumberExpression,
        1 reference
        BinaryExpression,
        1 reference
        ParenthesizedExpression
}
```

Each token that the lexer passes from the input text, will be a "SyntaxToken object" with its properties.

```
31 references | You, 4 days ago | 1 author (You)
class SyntaxToken : SyntaxNode{

    12 references
    public SyntaxToken(SyntaxKind kind, int position, string text, object value){
        Kind = kind;
        Position = position;
        Text = text;
        Value = value;
    }
```

```
2 references
class Lexer {
    7 references
    private readonly string _text;
    16 references
    private int _position;
    3 references
    private List<string> _diagnostics = new List<string>();
    2 references
    public Lexer(String text){
        _text = text;
    }

```

The lexer takes the text input, then tokenizes it with the suitable kind using "NextToken()" function.

```
2 references
public SyntaxToken NextToken(){
    // numbers
    // + - * / ()
    // <whitespace>
    if(Current == '\0'){
        return new SyntaxToken(SyntaxKind.EndOfFileToken, _position, "\0", null);
    }
```

If the current token is "\0", then the lexer will return an "End Of File" token.

```
        if(char.IsDigit(Current)){

            var start = _position;
            while(char.IsDigit(Current)){
                Next();
            }
```

While the current is number, it will call for “Next()” function recursively until it finds something else that is not a digit (0 - 9), then it will return a number token with the corresponding attributes of a syntax token.

```
        if(char.IsWhiteSpace(Current)){

            var start = _position;
            while(char.IsWhiteSpace(Current)){
                Next();
            }
```

While the current is white space, it will call for “Next()” function recursively until it finds something else that is not a white space, then it will return a number token with the corresponding attributes of a syntax token.

```
        if(Current == '+'){
            //We can replace the coming two lines with ++ as the following if statements
            var start = _position;
            Next();
            return new SyntaxToken(SyntaxKind.PlusToken, start, "+", null);
        }

        if(Current == '-')
            return new SyntaxToken(SyntaxKind.MinusToken, _position++, "-", null);
        else if(Current == '*')
            return new SyntaxToken(SyntaxKind.MultiplyToken, _position++, "*", null);
        else if(Current == '/')
            return new SyntaxToken(SyntaxKind.DivideToken, _position++, "/", null);
        else if(Current == '(')
            return new SyntaxToken(SyntaxKind.OpenParenthesisToken, _position++, "(", null);
        else if(Current == ')')
            return new SyntaxToken(SyntaxKind.CloseParenthesisToken, _position++, ")", null);

```

If it is not a number or a white space, it will check if ia is an operator.

```
_diagnostics.Add($"Error: bad chracter input: '{Current}'.");
return new SyntaxToken(SyntaxKind.BadToken, _position++, _text.Substring(_position - 1, 1), null);
```

Else, it will return a bad token.

Parser

We used the following rules to derive our expressions

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid num$$

After converting these rules to be LL(1) the following are our new rules

$$E \rightarrow T\ E'$$
$$E' \rightarrow +T\ E' \mid \varepsilon$$
$$T \rightarrow F\ T'$$
$$T' \rightarrow *F\ T' \mid \varepsilon$$
$$F \rightarrow (E) \mid num$$

Computing the first and follow :

	FIRST	FOLLOW
E –> TE'	{ id, (}	{ $,) }
E' –> +TE'/e	{ +, e }	{ $,) }
T –> FT'	{ id, (}	{ +, $,) }
T' –> *FT'/e	{ *, e }	{ +, $,) }
F –> id/(E)	{ id, (}	{ *, +, $,) }

Calculating parsing table :

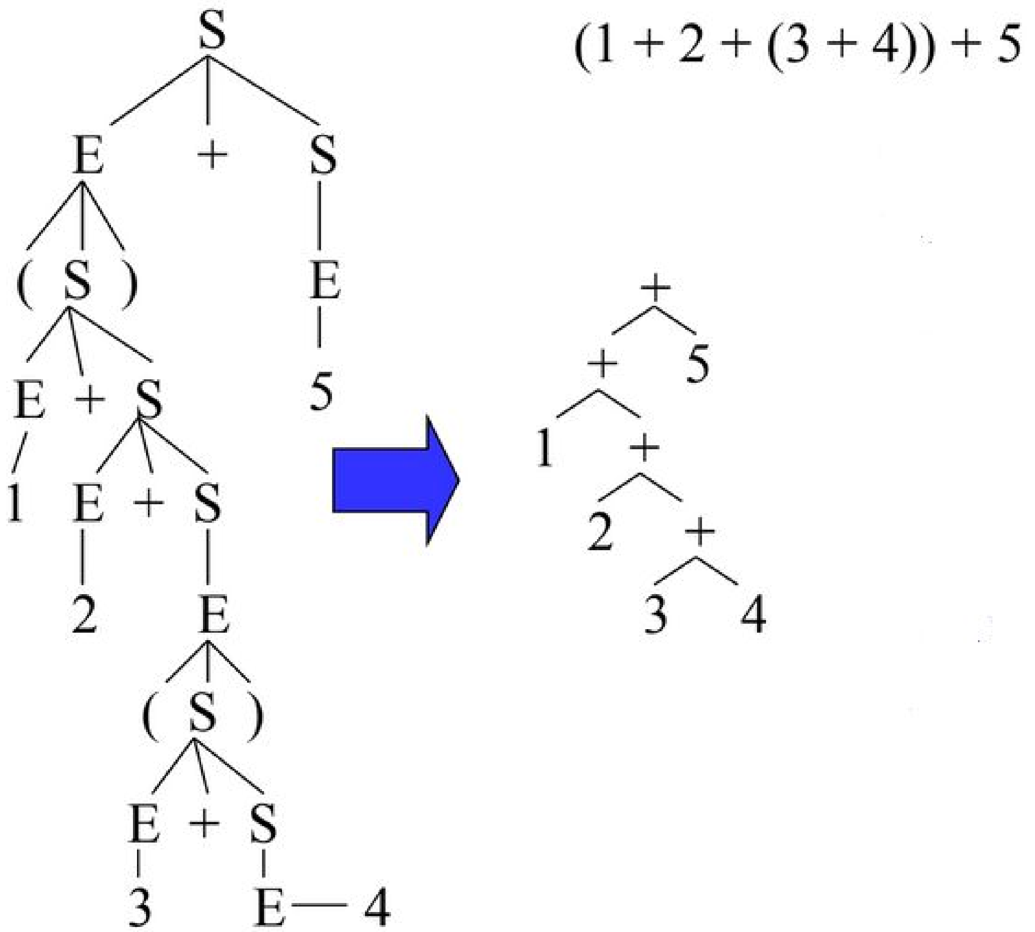
	ID	+	*	(	)	$
E	E –> TE'			E –> TE'		
E'		E' –> +TE'			E' –> e	E' –> e
T	T –> FT'			T –> FT'		
T'		T' –> e	T' –> *FT		T' –> e	T' –> e
F	F –> id			F –> (E)		

```
0 references | You, 12 hours ago | 1 author (You)
class Parser {

    4 references
    private readonly SyntaxToken[] _tokens;
    3 references
    private int _position;
    4 references
    private List<string> _diagnostics => new List<string>();
    0 references
    public Parser(string text){
        var tokens = new List<SyntaxToken>();
        var lexer = new Lexer(text);
        SyntaxToken token;

        do {
            token = lexer.NextToken();

            if(token.Kind != SyntaxKind.WhitespaceToken &&
                token.Kind != SyntaxKind.BadToken)
            {
                tokens.Add(token);
            }

        } while(token.Kind != SyntaxKind.EndOfFileToken);

        _tokens = tokens.ToArray();
        _diagnostics.AddRange(lexer.Diagnostics);
    }        You, 12 hours ago • Divide the code into multiple files.
```

The parser class is something like the above. It has an IEnumerable of tokens, an int to hold the current position in the tokens list, and a list of strings representing the bad tokens if any is inserted.

```
You, a few seconds ago | 1 author (You)
namespace Parser.CodeAnalysis{
    2 references | You, 13 hours ago | 1 author (You)
    sealed class SyntaxTree{
        1 reference
        public SyntaxTree(IEnumerable<string> diagnostocs, ExpressionSyntax root, SyntaxToken endOfFileToken){
            Diagnostics = diagnostocs.ToArray();
            Root = root;
            EndOfFileToken = endOfFileToken;
        }
        1 reference
        public IReadOnlyList<string> Diagnostics {get;}
        1 reference
        public ExpressionSyntax Root {get;}
        1 reference
        public SyntaxToken EndOfFileToken {get;}
}
}       You, a few seconds ago • Uncommitted changes
```

This “SyntaxTree” class represents the tree that the parser will return. It has a root, a list of diagnostics for errors, and an end of file.
It will be something like this :



This function returns a tree representing the operation.
It's all about how to get the root of the tree so that when we evaluate the operation we make sure that the result is correct.
We are using recursive descent to implement the tree. And wrote the code depending on the parsing table as the following :

```
2 references
public ExpressionSyntax ParseTerm(){
    //      +
    //     / \
    //    +   3
    //   / \
    //  1   2
    var left = ParseFactor();

    while (Current.Kind == SyntaxKind.PlusToken ||
           Current.Kind == SyntaxKind.MinusToken)
    {
        var operatorToken = NextToken();
        var right = ParseFactor();
        left = new BinaryExpressionSyntax(left, operatorToken, right);
    }

    return left;
}
```

Which calls for ParseFactor function :

```
2 references
public ExpressionSyntax ParseFactor(){
    var left = ParsePrimaryExpression();
    while (Current.Kind == SyntaxKind.MultiplyToken ||
           Current.Kind == SyntaxKind.DivideToken)
    {
        var operatorToken = NextToken();
        var right = ParsePrimaryExpression();
        left = new BinaryExpressionSyntax(left, operatorToken, right);
    }
    return left;
}       You, 14 hours ago • Divide the code into multiple files.
```

Which calls for ParsePrimaryExpression function :

```
2 references
private ExpressionSyntax ParsePrimaryExpression(){
    if(Current.Kind == SyntaxKind.OpenParenthesisToken){
        var left = NextToken();
        var expression = ParseTerm();
        var right = Match(SyntaxKind.CloseParenthesisToken);
        return new ParenthesizedExpressionSyntax(left, expression, right);
    }

    var numberToken = Match(SyntaxKind.NumberToken);
    return new NumberExpressionSyntax(numberToken);
}      You, 14 hours ago • Divide the code into multiple files.
```

We do this recursively till we get to the root of the tree. Then parse function will return a Syntax tree with the root, list of diagnostics, and end of file.

```
1 reference | You, 32 minutes ago | 1 author (You)
class Evaluator {

    2 references
    private readonly ExpressionSyntax _root;
    1 reference
    public Evaluator(ExpressionSyntax root){
        this._root = root;
    }

```

The evaluator is where we calculate the result starting from the tree. It takes the root that the parser passes.

```
    4 references
    private float EvaluateExpression(ExpressionSyntax node){
        // We have (until now):
        // BinaryExpresion, NumberExpression, Parentheses        You, 32

```

This function checks whether the root is a number, a binary expression, or a parentheses.

If the root is a number, it returns it as it's.

```
if(node is BinaryExpressionSyntax b){
    var left = EvaluateExpression(b.Left);
    var right = EvaluateExpression(b.Right);

    if(b.OperatorToken.Kind == SyntaxKind.PlusToken){
        return left + right;
    }
    else if(b.OperatorToken.Kind == SyntaxKind.MinusToken){
        return left - right;
    }
    else if(b.OperatorToken.Kind == SyntaxKind.MultiplyToken){
        return left * right;
    }
    else if(b.OperatorToken.Kind == SyntaxKind.DivideToken){
        return left / right;
    }
    else
        throw new Exception($"Unexpected binary operator: {b.OperatorToken.Kind}");
}       You, a day ago • Divide the code into multiple files.
```

The second if statement, is checking whether the root is a binary expression or not. If it is, it will recursively check on its children. And if they are numbers, it will return the result. Else, it will continue on checking the children kind.

If it isn’t a binary expression from these (+, -, *, /), it will throw an exception.

```
    if(node is ParenthesizedExpressionSyntax p){
        return EvaluateExpression(p.Expression);
    }
    throw new Exception($"Unexpected node: {node.Kind}");
}
```

The last if statement is the parentheses.

If any node does not satisfy any of the if statements, it will throw an exception that the node is an unexpected node.

The console will look like :

```
C:\Program Files\dotnet\dotnet.exe
> (1 + 2) * 4 - 5
BinaryExpression
    BinaryExpression
        ParenthesizedExpression
            OpenParenthesisToken
            BinaryExpression
                NumberExpression
                    NumberToken 1
                PlusToken
                NumberExpression
                    NumberToken 2
            CloseParenthesisToken
        MultiplyToken
        NumberExpression
            NumberToken 4
    MinusToken
    NumberExpression
        NumberToken 5
7
```

```
              -
           /     \
           *      5
        /     \
        +      4
      /    \
     1      2
```

The input is : (1 + 2) * 4 - 5. The tree is implemented first. Then, the result is displayed.

If there is a bad token in the input, the console will look like :

```
> $#$#!5
NumberExpression
    NumberToken 5
5
BadToken: '$'
BadToken: '#'
BadToken: '$'
BadToken: '#'
BadToken: '!'
NumberToken: '5'  5
```