Code for My compiler

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

// Lexical Analysis: Token definition
public enum SyntaxKind
{
    NumberToken,
    PlusToken,
    MinusToken,
    StarToken,
    SlashToken,
    OpenParenthesisToken,
    CloseParenthesisToken,
    EndOfFileToken
}

public class SyntaxToken
{
    public SyntaxToken(SyntaxKind kind, string text)
    {
        Kind = kind;
        Text = text;
    }

    public SyntaxKind Kind { get; }
    public string Text { get; }
}

// Syntax Tree: AST nodes
public abstract class SyntaxNode
{
}

public class NumberExpressionSyntax: SyntaxNode
{
    public NumberExpressionSyntax(SyntaxToken numberToken)
    {
        NumberToken = numberToken;
    }
```

```
    public SyntaxToken NumberToken { get; }
}

public class BinaryExpressionSyntax: SyntaxNode
{
    public BinaryExpressionSyntax(SyntaxNode left, SyntaxToken operatorToken, SyntaxNode right)
    {
        Left = left;
        OperatorToken = operatorToken;
        Right = right;
    }

    public SyntaxNode Left { get; }
    public SyntaxToken OperatorToken { get; }
    public SyntaxNode Right { get; }
}

// Parsing: Syntax tree construction
public class Parser
{
    private readonly SyntaxToken[] _tokens;
    private int _position;

    public Parser(string text)
    {
        var lexer = new Lexer(text);
        var tokens = new List<SyntaxToken>();

        while (true)
        {
            var token = lexer.NextToken();
            if (token.Kind == SyntaxKind.EndOfFileToken)
                break;

            tokens.Add(token);
        }

        _tokens = tokens.ToArray();
    }

    private SyntaxToken Current => _tokens[_position];

    private void NextToken()
```

```
{
    _position++;
}

private SyntaxToken Peek(int offset = 0)
{
    var index = _position + offset;
    return index < _tokens.Length? _tokens[index] : _tokens.Last();
}

public SyntaxNode Parse()
{
    return ParseTerm();
}

private SyntaxNode ParseTerm()
{
    var left = ParseFactor();

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

    return left;
}

private SyntaxNode ParseFactor()
{
    var left = ParsePrimary();

    while (Current.Kind == SyntaxKind.StarToken || Current.Kind == SyntaxKind.SlashToken)
    {
        var operatorToken = Current;
        NextToken();
        var right = ParsePrimary();
        left = new BinaryExpressionSyntax(left, operatorToken, right);
    }

    return left;
}
```

```
    private SyntaxNode ParsePrimary()
    {
        if (Current.Kind == SyntaxKind.NumberToken)
        {
            var numberToken = Current;
            NextToken();
            return new NumberExpressionSyntax(numberToken);
        }
        else if (Current.Kind == SyntaxKind.OpenParenthesisToken)
        {
            NextToken();
            var expression = ParseTerm();
            if (Current.Kind == SyntaxKind.CloseParenthesisToken)
                NextToken();
            return expression;
        }
        else
        {
            // Handle other primary expressions if needed
            // For simplicity, we only handle numbers and parentheses in this example
            throw new Exception("Unexpected token");
        }
    }
}

// Lexical Analysis: Tokenization
public class Lexer
{
    private readonly string _text;
    private int _position;

    public Lexer(string text)
    {
        _text = text;
    }

    private char Current => _position < _text.Length ? _text[_position] : '\0';

    private void Next()
    {
        _position++;
    }
```

```
    public SyntaxToken NextToken()
    {
        if (_position >= _text.Length)
            return new SyntaxToken(SyntaxKind.EndOfFileToken, string.Empty);

        if (char.IsDigit(Current))
        {
            var start = _position;
            while (char.IsDigit(Current))
                Next();
            var length = _position - start;
            var text = _text.Substring(start, length);
            return new SyntaxToken(SyntaxKind.NumberToken, text);
        }

        switch (Current)
        {
            case '+':
                Next();
                return new SyntaxToken(SyntaxKind.PlusToken, "+");
            case '-':
                Next();
                return new SyntaxToken(SyntaxKind.MinusToken, "-");
            case '*':
                Next();
                return new SyntaxToken(SyntaxKind.StarToken, "*");
            case '/':
                Next();
                return new SyntaxToken(SyntaxKind.SlashToken, "/");
            case '(':
                Next();
                return new SyntaxToken(SyntaxKind.OpenParenthesisToken, "(");
            case ')':
                Next();
                return new SyntaxToken(SyntaxKind.CloseParenthesisToken, ")");
            default:
                throw new Exception($"Unexpected character: {Current}");
        }
    }
}

// Evaluation: Interpretation of the syntax tree
public class Evaluator
{
```

```
    public int Evaluate(SyntaxNode node)
    {
        if (node is NumberExpressionSyntax number)
        {
            return int.Parse(number.NumberToken.Text);
        }
        else if (node is BinaryExpressionSyntax binary)
        {
            var left = Evaluate(binary.Left);
            var right = Evaluate(binary.Right);

            switch (binary.OperatorToken.Kind)
            {
                case SyntaxKind.PlusToken: return left + right;
                case SyntaxKind.MinusToken: return left - right;
                case SyntaxKind.StarToken: return left * right;
                case SyntaxKind.SlashToken: return left / right;
                default:
                    throw new Exception($"Unexpected operator: {binary.OperatorToken.Kind}");
            }
        }

        throw new Exception($"Unexpected syntax node: {node.GetType().Name}");
    }
}

// Main program
class Program
{
    static void Main()
    {
        while (true)
        {
            Console.Write("> ");
            var line = Console.ReadLine();
            if (string.IsNullOrEmpty(line))
                break;

            var parser = new Parser(line);
            var syntaxTree = parser.Parse();

            if (syntaxTree != null)
            {
                var evaluator = new Evaluator();
```

```
            var result = evaluator.Evaluate(syntaxTree);
            Console.WriteLine(result);
        }
        else
        {
            Console.WriteLine("Invalid expression");
        }
    }
  }
}
```