사칙연산 계산기 완성하기

연산 방식

* **중위 연산** : 일상에서 사용하는 연산 방식으로 ,
  피연산자 - 연산자 - 피연산자 순으로 표현
* **전위 연산** : 연산할 연산자를 먼저 표현하는 방식
  연산자 -피연산자1 - 피연산자2
* **후위 연산** : 연산할 연산자가 뒤에 표현되는 방식
  피연산지1 - 피연산자2 - 연산자

컴퓨팅에서는 연산의 동작이 '메모리 A 에 있는 값과 메모리 B 에 있는 값을 연산' 하는,
 후위 연산 방식으로 이루어집니다.
계산기를 구현할 때도 그런 방식으로 우선순위를 포함해 봅니다.

사람에게 입력받을테니 기본 입력은 중위 연산이라 가정하고,
먼저 이를 후위 연산으로 변환합니다

입력 : 1 + 2 + 3 * 4 + 5

Step	Token	연산자 스택	결과리스트
1	1		1
2	+	+	1
3	2	+	1 2
4			1 2 +
5	+	+	1 2 +
6	3	+	1 2 + 3
7	*	+ , *	1 2 + 3
8	4	+ , *	1 2 + 3 4
9	+	+	1 2 + 3 4 * +
10	5	+	1 2 + 3 4 * + 5
11			1 2 + 3 4 * + 5 +

변환된 후위 연산식 : 1 2 + 3 4 * + 5 +

후위 연산식을 계산

입력 : 1 2 + 3 4 * + 5 +

Step	Token	스택	동작
1	1	1	Stack.Push
2	2	1 , 2	Stack.Push
3	+	3	A = Stack.Pop B = Stack.Pop Stack.Push(A + B)
4	3	3 , 3	Stack.Push
5	4	3 , 3 , 4	Stack.Push
6	*	3 , 12	A = Stack.Pop B = Stack.Pop Stack.Push(A * B)
7	+	15	A = Stack.Pop B = Stack.Pop Stack.Push(A + B)
8	5	15 , 5	Stack.Push
9	+	20	A = Stack.Pop B = Stack.Pop Stack.Push(A + B)

그럼 기존의 계산기 (우선순위 없는 Calculator.cs) 를 고치기 위해,
두가지 과정을 거치게 됩니다.

1. **사람에게 입력받은 (Human readable) 중위 연산식을 후위 연산으로 변환하고**
2. **후위 연산식을 계산**

중위 연산 -> 후위 연산

```
126         // 후위 표현식으로 변환될 토큰 리스트
127         List<Token> postFix = new List<Token>();
128
129         // 연산자들을 담는 스택
130         Stack<Token> operators = new Stack<Token>();
131
132         // 1. 중위 표현식 → 후위 표현식으로 변환
133         for (int i = 0; i < _inputToken.Count; i++)
134         {
135             Token token = _inputToken[i];
136
137             // 숫자일 경우 결과(postfix)에 바로 추가
138             if (token.TokenType == Token.eTokenType.NUM)
139             {
140                 postFix.Add(token);
141             }
142             else
143             {
144                 // 연산자일 경우, 스택에 있는 연산자들과 우선순위를 비교
145                 while (operators.Count > 0 &&
146                     operators.Peek().OperatorPrecendence() >= token.OperatorPrecendence())
147                 {
148                     // 현재 연산자보다 우선순위가 높거나 같은 연산자들을 결과에 추가
149                     postFix.Add(operators.Pop());
150                 }
151                 // 현재 연산자를 스택에 push
152                 operators.Push(token);
153             }
154         }
155
156         // 남아있는 연산자를 모두 결과에 추가
157         while (operators.Count > 0)
158         {
159             postFix.Add(operators.Pop());
160         }
```

후위 표현식 계산

```
162         // 2. 후위 표현식 계산
163         Stack<Token> calcStack = new Stack<Token>();
164
165         for (int i = 0; i < postFix.Count; i++)
166         {
167             Token token = postFix[i];
168
169             // 숫자면 스택에 push
170             if (token.TokenType == Token.eTokenType.NUM)
171             {
172                 calcStack.Push(token);
173             }
174             else
175             {
176                 // 연산자면 두 숫자를 pop해서 계산
177                 float valueB = calcStack.Pop().Value;
178                 float valueA = calcStack.Pop().Value;
179
180                 // 계산 후 다시 스택에 push
181                 if (token.TokenType == Token.eTokenType.ADD)
182                     calcStack.Push(new Token() { Value = valueA + valueB, TokenType = Token.eTokenType.NUM });
183                 else if (token.TokenType == Token.eTokenType.SUB)
184                     calcStack.Push(new Token() { Value = valueA - valueB, TokenType = Token.eTokenType.NUM });
185                 else if (token.TokenType == Token.eTokenType.MUL)
186                     calcStack.Push(new Token() { Value = valueA * valueB, TokenType = Token.eTokenType.NUM });
187                 else if (token.TokenType == Token.eTokenType.DIV)
188                     calcStack.Push(new Token() { Value = valueA / valueB, TokenType = Token.eTokenType.NUM });
189             }
190         }
191
192         // 최종 계산 결과 반환
193         return _validStrings + " = " + calcStack.Pop().Value;
```