以 C#製作算式解析器函式庫

palapapa/
MathExpressionParser

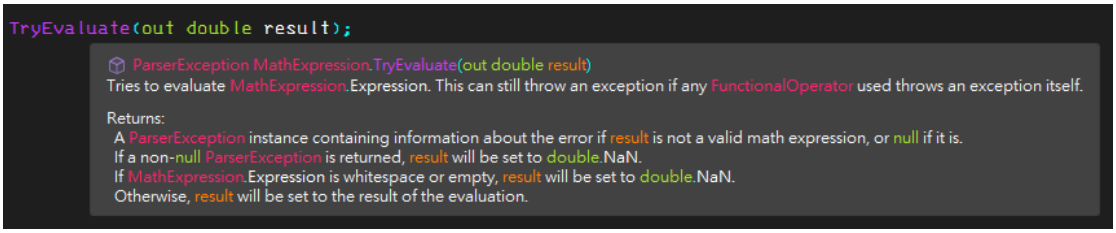
A lightweight customizable math expression parsing library that supports custom functions and variables, with a complete error handling system.

1 Contributor　1 Issue　0 Stars　0 Forks

https://github.com/palapapa/MathExpressionParser

一、作品簡介

在我這次的專案中，我利用 C#製作了一個可以解析一個以字串表示的數學算式的函式庫——MathExpressionParser——，它可以一次將一個複雜的算式的值計算出來，並且支援自訂函數與變數，使整個函式庫非常容易擴充，以符合使用者自己的需求。除此之外，它也包含了完整的錯誤訊息系統，當使用者提供的算式並不是正確的數學算式時，例如格式錯誤或者函數參數數量錯誤等，這個函式庫會以 Exception 的方式提供完整的錯誤訊息，包含錯誤位置、錯誤原因等。為了讓我的函式庫更方便使用，我也製作了完整的 documentation，每一個 class 與 method 都有完整與詳細的解說，並且因為我是利用 C#的 XML documentation 來撰寫，這些 documentation 在 Visual Studio 裡也可見，讓使用者不必不停翻看。如下圖所示：



在程式碼的部分，我充分利用了物件導向，將每一個運算子、函數與算式以物件的方式表示，讓我的程式碼變得易讀且易維護。我也第一次的利用了單元測試，我發現它是一個非常方便以及重要的工具，特別是對於我這種類型的專案，函式庫給出正確的結果是非常重要的一件事，而我透過單元測試，只要一鍵就可以驗證我的算法是否正確，他也因此讓我發現了無數個原本我完全不疑有他的程式碼，其實都暗藏了 bug。我也趁這個機會嘗試使用了 Travis CI，它會在我 push 到 GitHub 時自動執行我的單元測試，算是我第一次體驗持續整合。

下圖是一些常見的使用方法示範：

```
MathExpression expr = new("1 * (2 + 3) - (4 * 5 * (6 + 7))");
double value = expr.Evaluate(); // value is -255
```

```
MathExpression expr = new("1! * 2! * 3! * 4!");
double value = expr.Evaluate(); // value is 288
```

```
MathExpression expr = new("sin(90 torad)");
double value = expr.Evaluate(); // value is 1
```

```
MathExpression expr = new("sqrt(1 + sin(pi / 2) * 3)");
double value = expr.Evaluate(); // value is 2
```

```
MathExpression expr = new("log(log(3.2e+2, -(-1e1)), sqrt(1E-4))");
double value = expr.Evaluate(); // value is -0.19941686566
```

下表是所有支援的運算子以及函數：

Type	Name	Description
Binary	+	Addition
Binary	-	Subtraction
Prefix unary	-	Negation
Binary	*	Multiplication
Binary	/	Division
Binary	^	Power
Binary	%	Modulo
Constant	pi	π
Constant	e	Euler's number
Function(1)	sqrt	Square root
Function(1)	sin	Sine
Function(1)	asin	Arc sine
Function(1)	cos	Cosine

Function(1)	acos	Arc cosine
Function(1)	tan	Tangent
Function(1)	atan	Arc tangent
Function(1)	csc	Cosecant
Function(1)	acsc	Arc cosecant
Function(1)	sec	Secant
Function(1)	asec	Arc secant
Function(1)	cot	Cotangent
Function(1)	acot	Arc cotangent
Function(1)	sinh	Hyperbolic sine
Function(1)	asinh	Hyperbolic arc sine
Function(1)	cosh	Hyperbolic cosine
Function(1)	acosh	Hyperbolic arc cosine
Function(1)	tanh	Hyperbolic tangent
Function(1)	atanh	Hyperbolic arc tangent
Function(1)	csch	Hyperbolic cosecant
Function(1)	acsch	Hyperbolic arc cosecant
Function(1)	sech	Hyperbolic secant
Function(1)	asech	Hyperbolic arc secant
Function(1)	coth	Hyperbolic cotangent
Function(1)	acoth	Hyperbolic arc cotangent
Function(2)	P	Permutation
Function(2)	C	Combination
Function(2)	H	H(x, y) = C(x + y - 1, x - 1) = C(x + y - 1, y)
Function(2)	log	Logarithm. The second argument is the base.
Function(1)	log10	Logarithm base 10
Function(1)	log2	Logarithm base 2

Function(1)	ln	Natural logarithm
Function(1)	ceil	Least integer not less than
Function(1)	floor	Greatest integer not greater than
Function(1)	round	Round decimal places
Function(1)	abs	Absolute value
Function(any)	min	Returns the least value of the arguments. If no arguments are given, returns 0.
Function(any)	max	Returns the greatest value of the arguments. If no arguments are given, returns 0.
Postfix unary	!	Factorial
Postfix unary	torad	Converts degrees to radians
Postfix unary	todeg	Converts radians to degrees

下圖是如何加入自訂函數與變數的示範：

```
MathExpression expr = new("f(x)");
expr.CustomFunctions.Add(new FunctionalOperator("f", x => x[0] * 2, 1));
expr.CustomConstants.Add(new ConstantOperator("x", 100));
double value = expr.Evaluate(); // value is 200
```

下圖是錯誤訊息處理的示範：

```
MathExpression expr = new("sin(1,)");
Console.WriteLine(expr.Validate()?.Context);
// prints ParserExceptionContext { Position = 6, Type = UnexpectedClosingParenthesis }
```

```
MathExpression expr = new("2sin(2)");
Console.WriteLine(expr.Validate()?.Context);
// prints ParserExceptionContext { Position = 1, Type = UnexpectedFunctionalOperator }
```

```
MathExpression expr = new("cos(1, 2)");
Console.WriteLine(expr.Validate()?.Context);
// prints ParserExceptionContext { Position = 0, Type = IncorrectArgumentCount }
```

```
MathExpression expr = new("tan(pipi)");
Console.WriteLine(expr.Validate()?.Context);
// prints ParserExceptionContext { Position = 4, Type = UnknownOperator }
```

下表是所有支援的錯誤類型：

Type	Description
InvalidNumberFormat	An error where a number with an invalid format was found in a `MathExpression`.
IncorrectArgumentCount	An error where either too many or too few arguments were passed to a `FunctionalOperator`.
InvalidCustomFunctionName	An error where some of the custom functions provided have names that either start with a number, are empty, or contain characters that are not alphanumeric or are not underscores.
NullCustomFunction	An error where `MathExpression.CustomFunctions` contains a null element.
InvalidCustomConstantName	An error where some of the custom constants provided have names that either start with a number, are empty, or contain characters that are not alphanumeric or are not underscores.
NullCustomConstant	An error where `MathExpression.CustomConstants` contain a mull element.
NaNConstant	An error where `MathExpression.CustomConstants` have a `ConstantOperator.Value` of `double.NaN`.
ConflictingNames	An error where two `Operator`s in a `MathExpression` share the same name.
UnexpectedBinaryOperator	An error where a `BinaryOperator` is used incorrectly.
TooManyOpeningParentheses	An error where a opening parenthesiis is used without a corresponding opening parenthesis.
UnexpectedClosingParenthesis	An error where a closing parenthesis is used incorrectly, or where a closing parenthesis is used without a corresponding opening parenthesis.
UnexpectedComma	An error where a comma is used incorrectly.
UnexpectedConstantOperator	An error where a `ConstantOperator` is used incorrectly.
UnexpectedFunctionalOperator	An error where a `FunctionalOperator` is used incorrectly.
UnexpectedNumber	An error where a number is used incorrectly.
UnexpectedOpeningParenthesis	An error where a opening parenthesis is used incorrectly.
UnexpectedPostfixUnaryOperator	An error where a `PostfixUnaryOperator` is used incorrectly.
UnexpectedPrefixUnaryOperator	An error where a `PrefixUnaryOperator` is used incorrectly.
UnknownOperator	An error where an unknown `Operator` is used in a `MathExpression`.
UnexpectedNewline	An error where a `MathExpression` ended unexpectedly.

二、製作過程與遭遇困難

在最一開始的時候，我認為這個專案的製作不會太難，在沒有先研究清楚實作這個功能會需要什麼演算法的情況下就一股腦的做下去了，結果後來慢慢發現事情沒有我想得那麼簡單。

我一開始的打算是，將所有種類的運算子分門別類表示成物件，像是前綴與後綴一元運算子、二元運算子、函數，再把他們要對數字做的運算表示成 delegate（函數指標），以及他們的優先度和名字存在他們裡面。之後，當要做計算的時候，只要先掃描一遍字串，找出優先度最高的運算子，再根據它的型別來判斷它要如何取參數，例如二元運算子就是從它的左邊和右邊各取一個數字來當成參數傳給屬於那個運算子的 delegate，並且把 delegate 的回傳值放回原本的地方。之後再掃描一次字串，找出優先度次高的運算子，一直重複直到所有整個字串只剩下一個數字。

但是這很快就出現了問題，像是沒有辦法處理括號，例如加號的右邊是左括號的話，因為它不是數字就會無法處理。我之後想到了可以用遞迴，把每一個括號都當成一個新的算式，但是括號裡面可能又會有巢狀括號，因此程式碼越來越複雜，執行效率也不佳，而且如果使用遞迴，在算式出現錯誤時，會沒辦法輸出正確的錯誤位置，因為解析器函數只能看到它目前處於的那一層遞迴的子字串。同時我也忘記了不同運算子可能會有不同的結合律，最典型的就是次方運算子，它有著由右到左的結合律，這也是我的算法沒辦法處理的。

算法的問題在我一次上網找資料時才被解決，我發現了 Shunting Yard Algorithm，它可以用來把中綴表示法轉換成後綴表示法，也可以處理上面提到的結合律的問題，而解析後綴表示法非常簡單，只要使用一個 stack 就可以做到，比起直接計算中綴表示法快速許多。但是有一個問題，維基百科上提供的實作程式碼不支援有多個參數的函數，也不支援一元運算子，我最後解決的方法是在轉換成後綴表示法之前先透過數逗號的方式先把一個函數的參數數量記起來，之後轉換為後綴表示法後才知道要拿多少個數字當參數，這麼做的原因是因為後綴表示法沒有括號，因此如果不先做處

理會不知道哪幾個數字是屬於函數的。至於一元運算子，則只要做一個小修改就可以支援。

我還犯了一個設計上的大錯，我允許使用者增刪任何運算子，我原本以為這可以讓我的函式庫更有彈性，結果最後我不得不捨棄這個設定，原因是因為我原本以為我還是可以透過運算子的名字搜尋它是哪一個型別，但是之後我發現這樣會把減號和負號搞混（前者是二元運算子，後者是一元前綴運算子），這代表我會需要「過載」，也就是不同型別的運算子可以有同一個名字，但是這也代表我不能再用名字來判斷它是哪一個型別了，只能透過它的上下文來判斷，如下圖：

```
bool isLastTokenOperator = MathOperator.IsValidOperatorName(lastToken ?? "0"),
    isLastTokenOutOfBound = lastToken is null,
    isLastTokenOpeningParenthesis = lastToken is "(",
    isLastTokenClosingParenthesis = lastToken is ")",
    isLastTokenNumber = operators.Any(o => o.Name == lastToken && o.GetType() == typeof(ConstantMathOperator)) || (lastToken?.IsDouble() ?? false);
bool isNextTokenOperator = MathOperator.IsValidOperatorName(nextToken ?? "0"),
    isNextTokenOutOfBound = nextToken is null,
    isNextTokenOpeningParenthesis = nextToken is "(",
    isNextTokenClosingParenthesis = nextToken is ")",
    isNextTokenNumber = operators.Any(o => o.Name == nextToken && o.GetType() == typeof(ConstantMathOperator)) || (nextToken?.IsDouble() ?? false);
if ((isLastTokenNumber && isNextTokenNumber) ||
    (isLastTokenNumber && isNextTokenOpeningParenthesis) ||
    (isLastTokenClosingParenthesis && isNextTokenOperator) ||
    (isLastTokenClosingParenthesis && isNextTokenNumber) ||
    (isLastTokenClosingParenthesis && isNextTokenOpeningParenthesis))
{
    return typeof(BinaryMathOperator);
}
else if ((isLastTokenOperator && isNextTokenOpeningParenthesis) ||
    (isLastTokenOpeningParenthesis && isNextTokenOpeningParenthesis))
{
    return typeof(FunctionMathOperator);
}
else if ((isLastTokenOperator && isNextTokenNumber) ||
    (isLastTokenOpeningParenthesis && isNextTokenOperator) ||
    (isLastTokenOpeningParenthesis && isNextTokenNumber) ||
    (isLastTokenOutOfBound && isNextTokenOperator) ||
    (isLastTokenOutOfBound && isNextTokenNumber) ||
    (isLastTokenOutOfBound && isNextTokenOpeningParenthesis))
{
    return typeof(PrefixUnaryMathOperator);
}
else if ((isLastTokenNumber && isNextTokenOperator) ||
    (isLastTokenNumber && isNextTokenClosingParenthesis) ||
    (isLastTokenNumber && isNextTokenOutOfBound) ||
    (isLastTokenClosingParenthesis && isNextTokenClosingParenthesis) ||
    (isLastTokenClosingParenthesis && isNextTokenOutOfBound))
{
    return typeof(PostfixUnaryMathOperator);
}
else if ((isLastTokenOperator && isNextTokenOperator) ||
    (isLastTokenOperator && isNextTokenClosingParenthesis) ||
    (isLastTokenOperator && isNextTokenOutOfBound) ||
    (isLastTokenOpeningParenthesis && isNextTokenClosingParenthesis) ||
    (isLastTokenOutOfBound && isNextTokenOutOfBound))
{
    return typeof(ConstantMathOperator);
}
else
{
    return null;
}
```

可以看出要做到這件事需要很複雜的程式碼，因為我有五種型別的運算子，所以如果要包含到一個運算子前面和後面是什麼的所有情況，就有 25 種情況要檢查，而且就算做了這麼多，我最後還是發現只透過一個運算子的前面和後面是什麼來決定它的型別，也還是不可避免的會產生歧義，例如 1 + -(1)，，因為那個負號後面接著一對括號，而我又允許任意型別的運算子都可以有任意名字，這

導致無法判斷它到底是函數還是一元前綴運算子，最後我只好只允許使用者增加函數和變數，其他預設的都不能改動，並把減號和負號會重複的情況當成一個特例解決了。

我還犯了另一個設計上的錯，因為我一開始還不熟悉怎麼寫一個物件導向的函式庫，我把所有東西都寫成 static，使用者需要自己把要解析的字串傳進一個 static 的方法裡，而這樣做就完全失去了物件導向的優勢和意義，我最後也全部重寫，建立了 MathExpression 這個 class 來代表一個算式。

我最後還遇到一個問題，就是我發現不能盲目的套用 Shunting Yard Algorithm，因為如果使用者給的是不合理的算式，演算法就會有不可預期的行為，例如「+ 1 1」，也就是說，這個演算法並沒有內建錯誤檢查。我最後決定使用「修改過的 state machine」來檢查錯誤，一般的 state machine 只有一個 state，但是因為算式可能有括號，而括號裡面就算是一個獨立的算式，因此只使用一個 state 明顯不足，所以我把 stack 結合 state machine，每次進入一個括號時，就在 stack 上新增一個 state，而離開括號時，就拆掉一個 state。每個 state 都表示了下一個 token 可以是什麼種類，例如當目前的 token 是二元運算子時，下一個就允許出現數字、變數、函數、括號或一元前綴運算子。

除此之外，有些種類的 token 只使用 state 來判斷的話是不夠的，例如逗號，他除了需要出現在數字、變數、後綴一元運算子或右括號之後以外，還需要出現在函數的括號裡面才算正確，這時候那個 stack 就派上用場了，在進入一個括號前，如果那個括號屬於一個函數，則在 stack 上新增一個 state 前，目前的 state 會被加上一個 flag，表示目前正在函數裡面，之後遇到像逗號這種情況時，只要看目前 state 底下的那個 state 有沒有那個 flag，就可以正確判斷了。在製作錯誤判斷系統的同時，我也學到了怎麼正確的寫一個 C#的 Exception，原本我設計成每一種可能的錯誤都有一個對應的 Exception，但是最後我有 20 種不同的錯誤，我就發現這個做法不適合，於是我把所有 Exception 全部集合成 ParserException，並

把錯誤的原因改成用 enum 表示。我也學到一個正確的 Exception 需要有被序列化的功能，因為一個 Exception 有可能會需要在一個網路中傳輸，所以我也查找資料並幫我的 ParserException 實作了正確的序列化功能。

三、心得與反思

在這次的專案製作完畢後，我認為我最大的收穫就是學習到怎麼正確的使用物件導向來製作一個函式庫。我學到要把程式各個部件分成獨立的 class，像是 MathExpression, Token, Operator 這些，而不是像我一開始一樣把所有東西都寫成 static。我也學到不應該把 Exception 做為 flow control 的工具，我一開始判斷一個字串是不是 double 的方法是直接把它傳給 double.Parse，如果他有丟出 Exception 就不是，但我後來發現丟出 Exception 極其耗時，因此改採用 double.TryParse，我也進而了解到 C#的標準函式庫裡的很多常丟出 Exception 的方法，都有一個 Try 版本，原因就是為了讓使用者可以檢查參數的正確性的同時避免丟出 Exception，因此我也在 MathExpressionParser 裡提供了 TryEvaluate 的方法，並了解了 out parameter 的使用。我也學到 C#的 GetHashCode 的用途和 record 的使用，它提供了值相等和 immutability 的功能，但是維持 reference type 的性質，讓我在比較 token 時很方便。我也學到隱式轉型運算子只有在轉型時不會丟失資訊且不會丟出 Exception 時才可以定義，我就因為曾經把所有轉型運算子定義為隱式，我以為這樣會比較方便，結果之後程式出現一個我一直找不到原因的 bug，我最後才發現是因為我的隱式轉型運算子不小心被呼叫，使它意外轉型，最後導致錯誤的方法被呼叫。我也學到了在物件導向中，程式應該要盡量使用 interface 而不是 concrete implementation，我透過在我的 public API 中使用 IList 而不是 List 來遵守這個原則。

總而言之，這次經驗讓我對 C#的熟悉度更高，我期望在以後利用 C#撰寫更多的專案。