‘Mates8’

Calculator of Polynomials

Programming Overview Tutorial

1. Handling simple expressions

Version 8.4.0

Xavier R. Junqué

Introduction

Version 8.4.0 is an overhauling starting point of previous Mates8 versions. Yet from the user’s perspective nearly nothing changes, in the insides, the code is almost all new. While the operators’ priority still keeps inherent to the code, the code core’s main algorithm -later on explained- remains. The rest has been modified for enhancement purposes.

The tutorial assumes the reader has certain Visual Basic .Net skills and, therefore, some object oriented programming knowledge. Although, there is no need of parsing methods background, perhaps more notions may infer greater insight.

Being the first of a series of documents and v8.4.x versions, gradually the tutorial will drive the Calculator to reach more completeness. The aim is to give those interested in the matter a better understanding about how current Mates8 Calculator does its tasks.

Nevertheless, this work is not focusing in the user’s form or interface coding, but highlighting the Calculator’s traits.

Finally, hoping you can enjoy and/or take advantage of this tutorial, the source code download address is:

http://xrjunque.nom.es/precis/tutorialv8_4.aspx

Presenting Matesv8.4.0

At least as a first approach, given a valid user input, this is a simple arithmetic expression contained in a string, the process involved carries: parsing, evaluating and giving back the result. The result is brought out by a class m8Response instance.

```
Public Class m8Response
    Dim sExpr As String
    Dim retDbl As Double
    Dim cfg As Config
    Public Property sExpression As String
        Get
            Return sExpr
        End Get
        Set(value As String)
            sExpr = value
        End Set
    End Property
    Public Property retDouble As Double
        Get
            Return retDbl
        End Get
        Set(value As Double)
            retDbl = value
        End Set
    End Property
    Public Property retCfg As Config
        Get
            Return cfg
        End Get
        Set(value As Config)
            cfg = value
        End Set
    End Property
    Public Overrides Function ToString() As String
        Return retDouble.ToString
    End Function
End Class
```

Table 1. 'm8Response' Class

The point to note in Table 1 is the retDouble property, which returns a double value, result of the arithmetic operation.

As mentioned, v8.4.0 is the first approximation capable to cope simple arithmetic, but precisely the lack of extra hard complexities, classes and code, helps to grab the main strokes.

Classes' overview

Besides the fore mentioned m8Response class, a Mates8 client should be aware of class Config, responsible of configuring both: how the input data has to be analyzed and how the output data must be formatted.

'Mates8' leans much to the VB.Net's 'Regex' class, which in turn makes use of patterns. The patterns involved in the code are bunched in class MathGlobal8 and mostly are shared strings. These strings or patterns determine how the tokenizing will be.

Now, two classes conform the calculator's essence:

- currentMatch
- exprParser

At the start, the client calls the exprParser.Parse method, passing two parameters: the user's input string and, optionally, an instance of the Config class. Later, inside the 'Parse' method, the 'getCurAndNxtToken()' method of currentMatch class is invoked as many times as tokens are in the input string; in each of the calls, two retToken class instances are originated. These two objects are accessible and give information about the current and the next token being processed.

```
    Function nextExpr() As Double            ' - +
        Dim dbA As Double
        Try
            dbA = nextTerm()
            Dim sOp As String = Me.cur.nxtRTkn.sVal
            Do While InStr("+-", sOp)
                cur.getCurAndNxtToken() ' skip operator
                If retErr IsNot Nothing Then
                    Exit Try
                End If
                Dim dbB As Double = nextTerm()
                Select Case sOp
                    Case "+"
                        dbA += dbB
                    Case Else
                        dbA -= dbB
                End Select
                sOp = cur.nxtRTkn.sVal
            Loop
        Catch ex As Exception
            Throw ex
        End Try
        Return dbA
    End Function
```

Table 2. exprParser.nextExpr() method.

In fact, 'Parse' method calls nextExpr; nextExpr calls nextTerm method; nextTerm calls nextPow method; nextPow calls nextToken; and nextToken calls getCurAndNxtToken(). In each of these methods, just specific operators are considered:

Method	Evaluates operators	Calls method
Parse	(None; returns response to client)	nextExpr
nextExpr	Subtraction and addition	nextTerm
nextTerm	Multiplication and division	nextPow
nextPow	Exponentiation and factorial	nextToken
nextToken	Functions (trigonometric, …)	getCurAndNxtToken

Execution's path

In this way, the operators precedence is implicit in the code. For instance, given the input string "2+3*4" lets see how it goes:

1. The client calls exprParser.Parse("2+3*4"):
   Dim eP as exprParser = exprParser.Parse("2+3*4")
2. 'Parse' calls nextExpr
3. nextExpr calls nextTerm
4. nextTerm calls nextPow
5. nextPow calls nextToken
6. nextToken calls getCurAndNxtToken
7. getCurAndNxtToken obtains two tokens:
   a. "2", i.e., the current token
   b. "+", this is, the next token
8. Execution returns to nextToken, where, being number "2" the current token, execution jumps back to nextPow.
9. nextPow, as the next token is "+", it cannot process the addition operator, because nextPow is in charge of "^" and "!" operators. So, execution returns back to nextTerm.
10. Similarly, from nextTerm execution returns to nextExpr.
11. In nextExpr, variable dbA is assigned with the value of the current token, this is, dbA = 2.

12. Now, nextExpr recognizes in the next token a “+” operator, therefore, calls getCurAndNxtToken() once to skip the “+” token (see Table 2) and calls nextTerm in order to get the second operand (notice in 11., variable dbA holds the first operand).
13. nextTerm calls nextPow and the process is repeated all the way down to getCurAndNxtToken. Here:
    a. “3” becomes the current token
    b. “*” becomes the next token
14. Execution returns the way up until nextTerm recognizes, in the next token, the “*” operator and
    a. Assigns to a local variable the first operand (=3)
    b. Skips “*” operator calling getCurAndNxtToken()
    c. Invokes nextPow in order to get the second multiplicand
    d. The process is repeated “downwards” and at getCurAndNxtToken:
        i. “4” becomes the current token
        ii. A flag End Of Tokens is set for the next token
    e. Back again in nextTerm the second multiplicand is assigned
    f. nextTerm evaluates 3*4. (Notice here, multiplication operation preceding addition; otherwise instead of 2+3*4, mistakenly, (2+3)*4 would have been handled)
    g. End Of Tokens is examined and execution goes back to nextExpr returning the value 12 (=3*4).
15. Back in nextExpr, the returning value, 12, is assigned to the second operand dbB, (see Table 2).
16. The addition 2+12 takes place.
17. nextExpr examines End Of Tokens, in the next token, exits the loop and comes back to Parse method. The return value is 14.
18. Parse method assigns the resulting value to the client’s return variable, i.e., a m8Response instance object, and exits the method, ergo, the execution’s control is, lastly, regained by the client:

```
        If eP.retErr IsNot Nothing Then
            Console.WriteLine(eP.retErr.ToString)
        Else
            Console.WriteLine(eP.retVal.ToString)
        End If
```

Two tokens sequence validity

Next, follows a guiding table to determine the validity of current and next token sequence in currentMatch.validate method:

Token sequences' table and validity			NEXT TOKEN																	
											operators				matrix	logical operators		ampe-	End	sub-	un-
			LP		num						(-,+,*,				operators	and, or, xor		rsand	Of	sti-	kno-
CURRENT TOKEN			√, Dx	RP	constant	fn	col	row	=	variable	/,^,!)	mod	dx	∫	(-,+,*,/,^)	nand, nor	not	&	Tokens	tute	wn
examples	token(s)	value	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16-17	20	25	50
(, √, Dx, Dy	LP, √, Dx (derivate)	1	((	()	(pi	(cos	(;	(|	(=	(x	(- or (+	(mod	(dx	(∫	all OK	none	(NOT				
)	RP	2	)*(	))	)^PI	)*cos	);	)|	) =	) * x	all OK	)mod	)dx	)*∫	all OK	all OK	) NOT				
1, -5, pi	num, constant	3	pi*(| pi)	2*pi, pi^2	pi*cos	pi;	4|	3 =	2 * x	all OK	2 mod	3 dx	4*∫	all OK	all OK	3*not				
cos,sin	fn	4	sin(| sin)	sin2	cos sin	cos ;	cos |	sin =	cos x	none	cos mod	cos dx	cos ∫	none	none	cos not				
; or comma	col	5	; (| ;)	; pi	; cos	;;	;|	; =	; x	;- or ;+	; mod	; dx	; ∫	none	none	; not				
| or carriage	row	6	| (| |)	| 5	| sin	| ;	||	| =	| x	|- or |+	| mod	| dx	| ∫	all OK	none	| not				
	=	7	= (| =)	= 2	= sin	= ;	= |	= =	= x	= - or = +	= mod	= dx	= ∫	none	none	= not				
x	variable	8	x*(| x)	x^2	x * cos	x ;	x |	x =	x * x	all OK	x mod	x dx	x * ∫	all OK	all OK	x * not				
(-,+,*,/,^,!)	operators	9	-(| -)	-pi	-cos	- ;	- |	- =	all ok, but ! * x	-1-	none	none	all OK	none, but ! mtxOp.	none	not		only !		
mod	modulo	10	mod(| mod)	mod 2	mod cos	mod ;	mod |	mod =	mod x	none	none	none	mod ∫	none	none	mod not				
dx, dy	integralResp	11	dx*(| dx)	dx*pi	dx*sin	dx ;	dy |	dx, dy =	dx * x	all OK	dx mod	dx dy	dx * ∫	all OK	all OK	dx not				
∫	integral	12	∫ (| ∫)	∫ pi	∫ cos	∫ ;	∫ |	∫ =	∫ x	∫- or ∫+	∫ mod	∫ dx	∫ ∫	none	none	∫ not				
(-,+,*,/,^)	matrix optors.	13	/(| *)	*PI	*cos	* ;	* |	* =	+ x	none	none	none	all OK	-1-	none	not		only !		
3 AND 15	and,or,xor, nand,nor	14	AND (| AND)	OR &b1	xor cos	nand ;	or |	and =	xor x	-2-	and mod	and dx	or ∫	none	none	xor not				
NOT &hF10	not	15	not (| not)	not &o5	not exp	not ;	not |	and =	not x	-3-	not mod	not dx	not ∫	none	none	not not				
&b10	binary	16																			
&o17	octal	16																			
&d10	decimal	16																			
&hFA	hexadecimal	16																			
&rad	radians	17																			
&deg	degrees	17																			
&grad	gradians	17																			
	Start Of Tokens	21									- or +				- or +				#, cnt, var,&		
**	substitute	25																			
	unknown	50																			

Legend	
valid:	valid scope or sequence example
error:	invalid scope or sequence
insert '*':	)*cos, pi*cos, ...
insert '^':	x ^ 2
unary - or +:	(-, (+, ;- and so on
needs further analyzing:	sin2

-1- Only [*,/,^} followed by - allowed, or ! followed by any

-2- Only allowed a logical operator followed by unary operator - or +

-3- Only allowed a logical NOT operator followed by unary - or + operators