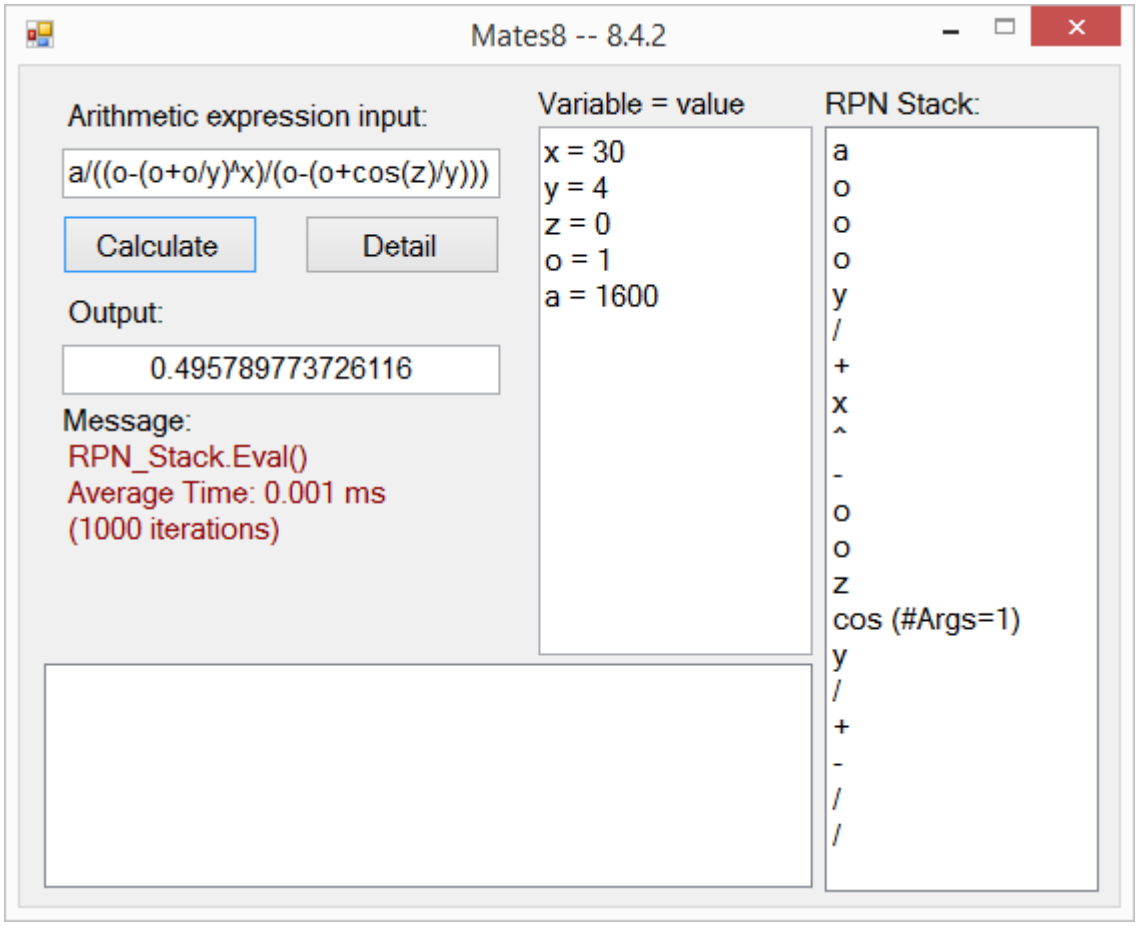
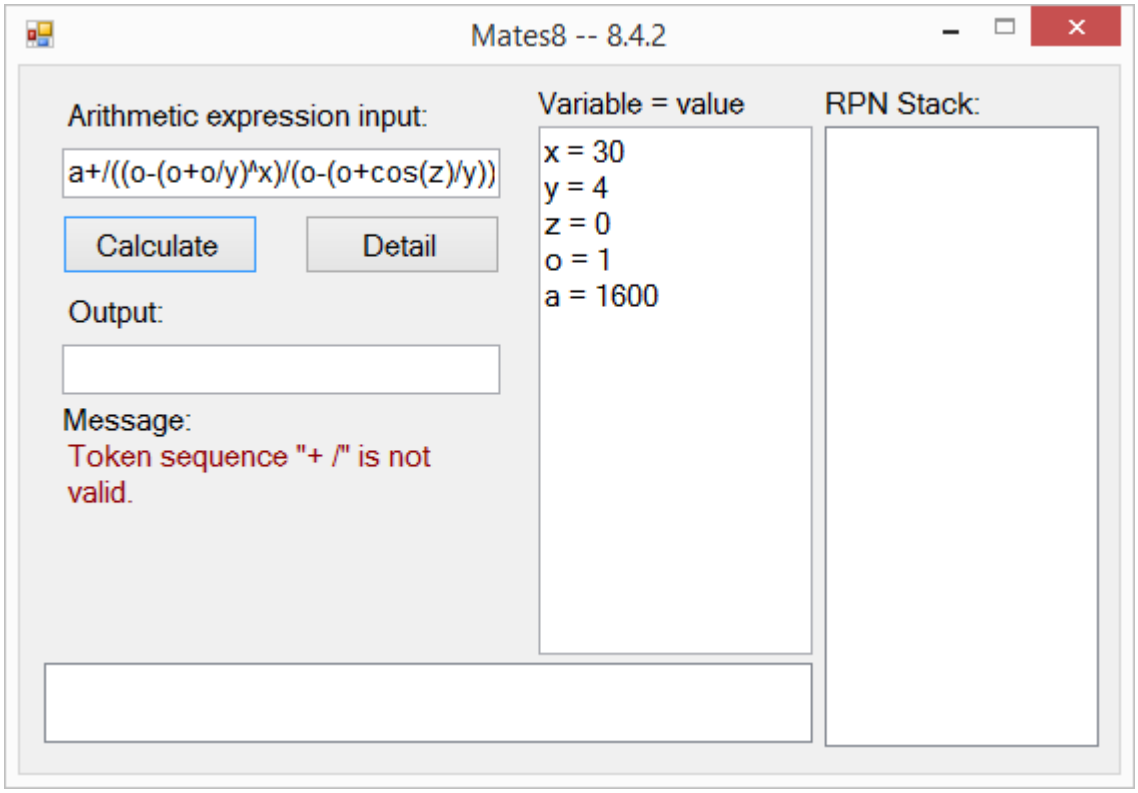
Well, again there are a lot of changes.

In version 8.4.2 there are only left four classes: exprParser, RPN_Stack, Config and msg8. Input validation and use of variables is included.





A. Client side.

```
    Private Sub Calculate()
        Dim eP As New exprParser()
        Dim sExprToParse As String = "a/((o-(o+o/y)^x)/(o-(o+cos(z)/y)))"

1:      If eP.Parse(sExprToParse) Then ' Parse the expression and if...
            ' OK, set variables' names...
            Dim sNames() As String = {"x", "y", "z", "o", "a"} ' (1 char long)
            Dim values() As Double = {30, 4, 0, 1, 1600} ' ...and values
            Dim db As Double = eP.Eval(sNames, values) '  Evaluate
            tbOutput.Text = db.ToString(Config.us) ' display
        Else
2:          lblMessage.Text = eP.retErr.ToString ' if error show message
            Exit Sub
        End If
    End Sub
```

As you may see, by now, variables must be one character long. The code is self-explaining, the only required parameters are the input string and, in case there are variables, their names and values.

To see the operations step by step, 'db' will hold the result and sDetail the steps in:

```
    Private Sub Detail()
        Dim eP As New exprParser()
        Dim sExprToParse As String = "a/((o-(o+o/y)^x)/(o-(o+cos(z)/y)))"
        If eP.Parse(sExprToParse) Then ' Parse the expression and if...
            ' OK, set variables' names...
            Dim sNames() As String = {"x", "y", "z", "o", "a"} ' (1 char long)
            Dim values() As Double = {30, 4, 0, 1, 1600} ' ...and values
            Dim db As Double
            Dim sDetail As String = _
                eP.ToStringDetail(db, sNames, values) '  Detail
            Trace.Write(sDetail)
        Else
            lblMessage.Text = eP.retErr.ToString ' if error show message
            Exit Sub
        End If
    End Sub
```

The content of sDetail is (current operation is enclosed by squared brackets):

```
1600/((1-(1+ [ 1/4 ] )^30)/(1-(1+cos(0)/4)))
1600/((1-(1+ [ 0.25 ] )^30)/(1-(1+cos(0)/4)))

1600/((1- [ 1+0.25 ] ^30)/(1-(1+cos(0)/4)))
1600/((1- [ 1.25 ] ^30)/(1-(1+cos(0)/4)))

1600/((1- [ 1.25^30 ] )/(1-(1+cos(0)/4)))
1600/((1- [ 807.793566946316 ] )/(1-(1+cos(0)/4)))

1600/( [ 1-807.793566946316 ] /(1-(1+cos(0)/4)))
1600/( [ -806.793566946316 ] /(1-(1+cos(0)/4)))

1600/(-806.793566946316/(1-(1+ [ cos(0) ] /4)))
1600/(-806.793566946316/(1-(1+ [ 1 ] /4)))

1600/(-806.793566946316/(1-(1+ [ 1/4 ] )))
1600/(-806.793566946316/(1-(1+ [ 0.25 ] )))

1600/(-806.793566946316/(1- [ 1+0.25 ] ))
1600/(-806.793566946316/(1- [ 1.25 ] ))

1600/(-806.793566946316/ [ 1-1.25 ] )
1600/(-806.793566946316/ [ -0.25 ] )

1600/( [ -806.793566946316/-0.25 ] )
1600/( [ 3227.17426778526 ] )

 [ 1600/3227.17426778526 ]
 [ 0.495789773726116 ]
```

B. The insides

1. Class 'exprParser'

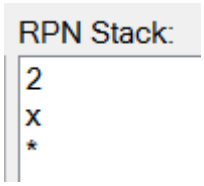
a. Overview.

The function of class 'exprParser' is to parse the input expression -a math expression- and obtain a RPN stack. (See http://en.wikipedia.org/wiki/Reverse_Polish_notation for more on Reverse Polish Notation details). Once the stack has been generated, it is possible to evaluate the expression at different variables' values.

Let's ilustrate through a simple sample. Given the input "2*x", the code snippet

```
Dim eP As New exprParser()
eP.Parse("2*x")
```

causes the following stack:



Here on, variable "x" may have any valid 'double' value and be evaluated.

Thus,

```
Dim sNames() As String = {"x"}
Dim values() As Double = {3}
Dim db As Double = eP.Eval(sNames, values) '  Evaluate
Trace.WriteLine("2*x=" + db.ToString(Config.us))
```

gives, as expected:

```
2*x=6
```

Moreover, assigning more values to "x":

```
Application.CurrentCulture = Config.us
Dim eP As New exprParser()
eP.Parse("2*x")
Dim sNames() As String = {"x"}
Dim xVal() As Double = {3, -1, 5.2, 2.3, 7} ' 5 values for x
For i As Int32 = 0 To xVal.Length - 1
    Dim xCurVal() As Double = {xVal(i)}
    Dim db As Double = eP.Eval(sNames, xCurVal) ' Evaluate each x value
    Trace.WriteLine(String.Format("2*{0} = {1}", xCurVal(0), db))
Next
```

results in the output window:

```
2*3 = 6
2*-1 = -2
2*5.2 = 10.4
2*2.3 = 4.6
2*7 = 14
```

b. How the stack is generated.

1.) Function Parse()

Function exprParse.Parse is:

```
  Public Function Parse(strToParse As String, _
                 Optional bValidate As Boolean = True) As Boolean
    Try
1:      Me.bValidate = bValidate
2:      If bValidate Then
          ' supress leading and trailing white spaces:
3:        sbExpr = New StringBuilder(Regex.Replace(strToParse, "(^(\+|\s)+)|((" + _
              Config.sCol + "|" + Config.sRow + "|\s+)$)", ""))
        Else
4:        sbExpr = New StringBuilder(strToParse)
        End If
5:      ln = sbExpr.Length
6:      If ln = 0 Then
          Dim err As New msg8(Me, 1) ' empty string
          Exit Try
        End If
7:      If Not rpn1.bInitialized Then
8:        rpn1 = New RPN_Stack(Me, ln * 3 / 2)
        Else
9:        rpn1.Clear()
        End If
10:     iRe = 0 : curNum = 0.0 : nOpnd = 0 : iToken = -1 : LP = 0 : RP = 0
11:     err = Nothing
12:     nextExpr()
13:     If err IsNot Nothing Then
          Exit Try
        End If
14:     ReDim Preserve rpn1.oStack(rpn1.iSt - 1)
15:     If bValidate AndAlso LP > RP Then
          err = New msg8(Me, 10) ' missing matching RP
          Return False
        End If
16:     Return True
    Catch ex As Exception
      err = ex
    End Try
    Return False
  End Function
```

Lines 1 through 11 are for initialization and some validation. In line #1 if bValidate is "True", later, will cause token sequence validation and inclusion of "*" or "^". For example, validating the input "2x" will transform the expression into "2*x", or "2x2" will become "2*x^2".

However, the call to function exprParser.Parse() for the input "2*/x" will return "False" indicating some error. Then, examination of member exprParser.retErr would give (see line #2 in the first code snippet):

```
?eP.retErr.ToString
"Token sequence "* /" is not valid."
```

Line #12 invokes the core of the algorithm, as we will see. In line #13 execution exits if some error has been found. Line #14 adjusts the stack to the real size, because of been initially oversized. Line #15 compares left and right parenthesis count. In case LP < RP, the error would be detected previously in the course of line #12.

2.) Sub nextExpr()

In nextExpr() procedure, first nextTerm() is invoked and, on return, in case the input expression contains one or more "+" (or "-") operators, the execution will enter into the loop in line #3. It will leave the loop, only in case that inside line #6 nextTerm(), a ")" token or end of tokens is found. The same applies for line #1. Tokenizing takes place, as we will see, exclusively in nextToken().

```
    Private Sub nextExpr()          ' - +
        Try
1:          nextTerm()
2:          Do While curOptor = 45 OrElse curOptor = 43
3:             Dim curOptor As optor = Me.curOptor
4:             Dim optoriTkn As Int32 = Me.optoriTkn
5:             Dim curPos As Int32 = Me.opndCurPos
6:             nextTerm()
7:             rpn1.Add(New StackTkn(tokenType.optor, curOptor, _
                       0, curPos, optoriTkn, Chr(curOptor))) ' operator
           Loop
        Catch ex As Exception
            err = ex
        End Try
    End Sub
```

The value curOptor = 45 corresponds to the character "+" ascii value and similarly Asc("-") = 43.

```
Friend Enum optor
    ' -  +  *  /  ^  %  !  .  :
    '45 43 42 47 94 37 33 46 58
    substract = 45
    add = 43
    multiply = 42
    divide = 47
    power = 94
    modulo = 37
    factorial = 33
End Enum
```

Line #3 saves the current operator's ascii value, this is, a 45 in case of addition or 43 for substraction. The value needs to be saved because during line #6 execution, member's "me.CurOptor" value will be overridden.

Lines #4 and #5 save, respectively, the token count for the operator and the position of the operator in the input string. If, for example, the input string is "23-45-78" and we add in between lines #5 and #6

```
    Trace.WriteLine(String.Format("operator={0}", Chr(Me.curOptor)))
    Trace.WriteLine(String.Format("Me.curOptor={0}", Me.curOptor))
    Trace.WriteLine(String.Format("Me.optoriTkn={0}", Me.optoriTkn))
    Trace.WriteLine(String.Format("Me.opndCurPos={0}", Me.opndCurPos))
    Trace.WriteLine("")
```

there will be two iterations through lines #2 and #7: one for "23-45" and a second for "(23-45)+78".

```
operator=-
Me.curOptor=substract
Me.optoriTkn=1  <-- corresponds to token "-" (token "23" is token #0)
Me.opndCurPos=0

operator=+
Me.curOptor=add
Me.optoriTkn=3 <-- corresponds to token "+" (token "45" is token #2)
Me.opndCurPos=0
```

Line #7 a new StackTkn object is added to the stack array.

3.) Sub nextTerm()

In a similar way, nextTerm() will enter the loop just for "*" or "/" operators and exit when comming back from the call to nextPow() because of finding a ")" token or the end of tokens.

```
Private Sub nextTerm() '     * /
  Try
    nextPow()
    '- + * / ^ % ! . :
    '45 43 42 47 94 37 33 46 58
    Do While curOptor = 42 OrElse curOptor = 47
      Dim curOptor As optor = Me.curOptor
      Dim optoriTkn As Int32 = Me.optoriTkn
      Dim curPos As Int32 = Me.opndCurPos
      nextPow()
      rpn1.Add(New StackTkn(tokenType.optor, curOptor, _
          0, curPos, optoriTkn, Chr(curOptor))) ' operator
    Loop
  Catch ex As Exception
    err = ex
  End Try
End Sub
```

4.) Sub nextPow()

Also, the same stands for "nextPow()" procedure, except that power operator proceeds operating from right to left, i.e. power operator has "right associativity" (http://en.wikipedia.org/wiki/Operator_associativity). Here the operators involved are "^" and "%".

```
Private Sub nextPow() '     ^ !
    Dim sgn As Int32
    Try
        Dim pos As Int32 = opndCurPos + 1
        nextToken(sgn)
        ' -  +  *  /  ^  %  !  .  :
        '45 43 42 47 94 37 33 46 58
        Do While curOptor = 94 OrElse curOptor = 37
            pos = opndCurPos + 1
            If curOptor = 94 Then ' ^ operator
            ......
            ......
            ......
            Else
                ' % operator
                nextToken(sgn)
                rpn1.Add(New StackTkn(tokenType.optor, 37, 0, pos, _
                    optoriTkn, "%")) ' operator
                sgn = 1
            End If
            nOpnd -= 1
        Loop
        If sgn = -1 Then
            rpn1.AddOptor(tokenType.chgSgn, chgSgnPos, chgSgniTkn)
        End If
    Catch ex As Exception
        err = ex
    End Try
End Sub
```

5.) Sub nextToken()

Next and last procedure in the calling chain, although there are cases including more calls, is "nextToken()", schematically:

```
1:        Dim c As Int32
2:        Dim bNotUnary As Boolean
          Try
3:            sgn = 1
4:            Do
5:                iToken += 1
6:                If iRe >= ln Then Exit Do
7:                c = AscW(sbExpr.Chars(iRe))
retry:
8:                If c = 45 OrElse c = 43 OrElse c = 42 OrElse c = 47 _
                  OrElse c = 94 OrElse c = 37 OrElse c = 33 Then  '      O P E R A T O R
                    ' -  +  *  /  ^  %  !
                    '45 43 42 47 94 37 33
8:                  ...
9:                ElseIf (48 <= c AndAlso c <= 57) OrElse c = 46 Then '       N  U  M  B  E  R
10:                 ...
11:               ElseIf c = 40 Then ' 40 = "(",  LP
12:                 ...
13:               ElseIf c = 41 Then ' 41 = ")", RP
14:                 ...
15:               ElseIf (97 <= c AndAlso c <= 122) OrElse _
                  (65 <= c AndAlso c <= 90) Then ' 97="a" 122="z" 65="A" 90="Z"  fn / Var
                    ' Is a function?
16:                 Dim m As Match = Config.reFnAndVars.Match(sbExpr.ToString, iRe)
17:                 If m.Groups("fn").Success Then
                            '       F U N C T I O N
18:                  ...
19:                 ElseIf m.Groups("const").Success Then
                            '          C O N S T A N T
20:                  ...
21:                 ElseIf m.Groups("var").Success Then
                            '       V A R I A B L E
22:                   ...
                    Else
                        ' TODO
                    End If
23:               ElseIf c = 32 Then ' SPACE
24:                   iRe += 1
25:               ElseIf c = 1055 OrElse c = 1087 Then ' constant Π (lower/upper case)
26:               ElseIf c = 58 Then ' 58=":"
                      ' transform ":" into "/"
27:                   c = 47 : GoTo retry '  47=/
28:               ElseIf c = 58 OrElse c = 247 Then ' OPERATOR
                      '  :  ÷
                      '  58 247
29:                   c = 47 ' convert ":" or "÷" into "/"
30:                   GoTo retry
31:               Else
32:                   Dim vsErr() As String = {Chr(c)}
33:                   err = New msg8(Me, 6, vsErr)   ' not allowed/unknown token
34:                   Exit Do
                  End If
35:               bNotUnary = True
36:             Loop While iRe < ln

37:             curOptor = -4 ' End Of Tokens

          Catch ex As Exception
38:             err = ex
          End Try
      End Sub
```

To be continued....