FOXES TEAM

Reference for clsMathParser

- clsMathParser -

A Class for Math Expressions Evaluation in Visual Basic

REFERENCE FOR

- clsMathParser

v. 4.2. Oct. 2006

Leonardo Volpi

Index

Summary

MathParser - clsMathParser 4. - is a parser-evaluator for mathematical and physical string expressions.
This software, based on the original MathParser 2 developed by Leonardo Volpi , has been modified with the collaboration of Michael Ruder to extend the computation also to the physical variables like "3.5s" for 3.5 seconds and so on. In advance, a special routine has been developed to find out in which order the variables appear in a formula string. Thomas Zeutschler has kindly revised the code improving general efficiency by more than 200 % . Lieven Dossche has finally encapsulated the core in a nice, efficient and elegant class. In addition, starting from the v.3 of MathParser, Arnoud has created a sophisticated class- clsMathParserC - for complex numbers adding a large, good collection of complex functions and operators in a separate, reusable module.

clsMathParser 4

This document describes the clsMathParser 4.x class evaluating real math expressions.
In some instances, you might want to allow users to type in their own numeric expression in ASCII text that contains a series of numeric expressions and then produce a numeric result of this expression.
This class does just that task. It accepts as input any string representing an arithmetic or algebraic expression and a list of the variable's values and returns a double-precision numeric value. clsMathParser can be used to calculate formulas (expressions) given at runtime, for example, for plotting and tabulating functions, making numerical computations and many other. It works in VB 6, and VBA.
Generally speaking clsMathParser is a numeric evaluator - not compiled - optimized for fast loop evaluations.
The set of expression strings, physical expressions, constants and international units of measure is very wide.
Typical mixed math-physical expressions are:

1+(2-5)*3+8/(5+3)^2	sqr(2)
(a+b)*(a-b)	x^2+3*x+1
300 km + 123000 m	(3000000 km/s)/144 Mhz
256.33*Exp(-t/12 us)	(1+(2-5)*3+8/(5+3)^2)/sqr(5^2+3^2)
2+3x+2x^2	0.25x + 3.5y + 1
0.1uF*6.8kohm	sqr(4^2+3^2)
(12.3 mm)/(856.6 us)	(-1)^(2n+1)*x^n/n!
And((x<2),(x<=5))	sin(2*pi*x)+cos(2*pi*x)

Variables can be any alphanumeric string and must start with a letter

```
x  y  a1  a2  time  alpha   beta
```

Also the symbol "_" is accepted for writing variables in "programming style".

```
time_1  alpha_b1   rise_time
```

Implicit multiplication is not supported because of its intrinsic ambiguity. So "xy" stands for a variable named "xy" and not for x*y. The multiplication symbol "*" cannot generally be omitted.
It can be omitted only for coefficients of the classic math variables ***x***, ***y***, ***z***. This means that strings like “2x” and “2*x” are equivalent

```
2x   3.141y   338z^2   ⇔   2*x    3.141*y    338*z^2
```

On the contrary, the following expressions are illegal.

```
2a  3(x+1)  334omega  2pi
```

Constant numbers can be integers, decimal, or exponential

```
2 -3234  1.3333  -0.00025   1.2345E-12
```

From version 4.2, MathParser accepts both decimal symbols "." or ",". See international setting

Physical numbers are numbers followed by a unit of measure

`"1s"` for 1 second `"200m"` for 200 meters `"25kg"` for 25 kilograms

For better reading they may contain blanks

```
"1 s"   "200 m"   "25 kg"   "150 MHz"   "0.15 uF"   "3600 kohm"
```

They may also contain the following **multiplying factors**:

```
T=10^12  G=10^9  M=10^6  k=10^3  m=10^-3  u=10^-6  n=10^-9  p=10^-12
```

Functions are called by their function-name followed by parentheses. Function arguments can be: numbers, variables, expressions, or even other functions

```
sin(x)   log(x)   cos(2*pi*t+phi)   atan(4*sin(x))
```

For functions which have more than one argument, the successive arguments are separated by commas (default)

```
max(a,b)     root(x,y)    BesselJ(x,n)    HypGeom(x,a,b,c)
```

Note. From version 4.2 , the argument separator depends on the MathParser decimal separator setting. If decimal symbol is point "." (i.e. 3.14) , the argument separator is ",". If it is comma "," (i.e. 3,14) , the argument separator is ";".

Logical expressions are supported

```
x<1   x+2y >= 4    x^2+5x-1>0   t<>0   (0<x<1)
```

Logical expressions return always 1 (True) or 0 (False). Compact expressions, like “0<x<1” , are supported. We can enter `(0<x<1) or (0<x)*(x<1)` as well.

Numerical range can be inserted using logical symbols and boolean functions. For example:

For `2<x<5`	insert	`(2<x)*(x<5)`	or also `(2<x<5)`	
For `x<2 , x>=10`	insert	`OR(x<2, x>=10)`	or also `(x<2)+(x>=10)`	
For `-1<x<1`	insert	`(x>-1)*(x<1)` ,	or `(-1<x<1)` ,	or also `|x|<1`

Piecewise Functions. Logical expressions can also be useful for defining piecewise functions, such as:

$$f(x) = \begin{cases} 2x\text{-}1\text{-}\ln(2) & x \le 0.5 \\ \ln(x) & 0.5 < x < 2 \\ x/2\text{-}1\text{+}\ln(2) & x \ge 2 \end{cases}$$

The above function can be written as:

```
f(x) = (x<=0.5)*(2*x-1-ln(2))+(0.5<x<2)*ln(x)+(x>=2)*(x/2-1+ln(2))
```

Starting from v3.4, the parser adopts a new algorithm for evaluating math expressions depending on logical expressions, which are evaluated only when the correspondent logical condition is true (Conditioned-Branch algorithm). Thus, the above piecewise expression can be evaluated for any real value x without any domain error. Note that without this features the formula could be evaluated only for x>0. Another way to compute piecewise functions is splitting it into several formulas (see example 6)

Percentage. it simply returns the argument divided by 100

`3%` `=>` returns the number 3/100 = 0.03

Math Constants supported are: Pi Greek (π), Euler-Mascheroni (γ) , Euler-Napier’s (e), Goldean mean (φ). Constant numbers must be suffixed with # symbol (except pi-greek that can written also without a suffix for compatibility with previous versions)

```
pi = 3.14159265358979    or  pi# = 3.14159265358979
```

```
pi2# = 1.5707963267949  (π /2),  pi4# = 0.785398163397448  (π /4))
eu# = 0.577215664901533
e# = 2.71828182845905
phi# = 1.61803398874989
```

Note: pi-greek constant can be indicated with “pi” or “PI” as well. All other constants are case sensitive.

Angle expressions
This version supports angles in radians, sexagesimal degrees and centesimal degrees. The right angle unit can be set by the property AngleUnit ("RAD" is the default unit). This affects all angle computation of the parser.

For example if you set the unit "DEG", all angles will be read and converted in degree

```
sin(120)  =>  0.86602540378444
asin(0.86602540378444)  =>  60
rad(pi/2)  =>  90         grad(400)  =>  360       deg(360)  =>  360
```

Angles can also be written in **ddmmss** format like for example 45d 12m 13s

```
sin(29d 59m 60s)  =>  0.5            29d 59s 60m  =>  30
```

Note This format is only for sexagesimal degree. It’s independent from the unit set
Note The old format 45° 12' 13" is no more supported

Physical Constants supported are:

Planck constant	**h#**	6.6260755e-34 J s
Boltzmann constant	**K#**	1.380658e-23 J/K
Elementary charge	**q#**	1.60217733e-19 C
Avogadro number	**A#**	6.0221367e23 particles/mol
Speed of light	**c#**	2.99792458e8 m/s
Permeability of vacuum (m)	**mu#**	12.566370614e-7 T^2m^3/J
Permittivity of vacuum (e)	**eps#**	8.854187817e-12 C^2/Jm
Electron rest mass	**me#**	9.1093897e-31 kg
Proton rest mass	**mp#**	1.6726231e-27 kg
Neutron rest mass	**mn#**	n 1.6749286e-27 kg
Gas constant	**R#**	8.31451 m^2kg/s^2k mol
Gravitational constant	**G#**	6.672e-11 $m^3/kg\ s^2$
Acceleration due to gravity	**g#**	9.80665 m/s^2

Physical constants can be used like any other symbolic math constant.
Just remember that they have their own dimension units listed in the above table.

Example of physical formulas are:

```
 m*c# ^2         1/(4*pi*eps#)*q#/r^2       eps# * S/d
sqr(m*h*g#)      s0+v*t+0.5*g#*t^2
```

Multivariable functions. Starting from 4.0, clsMathParser also recognizes and calculates functions with more than 2 variables. Usually they are functions used in applied math, physics, engineering, etc. Arguments are separated by commas (default)

```
 HypGeom(a,b,c,x)    Clip(x,a,b)    betaI(x,a,b)   DNorm(x,μ,σ)  etc.
```

Functions with variable number of arguments. clsMathParser accepts and calculates functions with variable number of arguments (max 20). Usually they are functions used in statistic and number theory.

```
min(x1,x2,...)    max(x1,x2,...)    mean(x1,x2,...)    gcd(x1,x2,...)
```

The max argument limit is set by the global constant HiARG

Time functions. These functions return the current date, time and timestamp of the system. They have no argument and are recognized as intrinsic constants

`Date#` = current system date
`Time#` = current system time
`Now#` = current system timestamp (date + time)

International setting. Form 4.2, clsMathParser can accept both decimal separators symbols "." or ",". Setting the global constant DP_SET = False, the programmer can force the parser to follow the international setting of the machine. This means that possible decimal numbers in the input string must follow the local international setting.

Example: the expression $0.0125(1+0.8x-1.5x^2)$

must be written as `0.0125*(1+0.8x-1.5x^2)` if you system is set for decimal point "." or, on the contrary, must be written `0,0125*(1+0,8x-1,5x^2)` for system working with decimal comma ",".

Setting DP_SET = True, the parser, as in the previous releases, ignores the international setting of the system. In this case the only valid decimal separator is "." (point). This means that the math input strings are always valid independently from the platform local setting.

Of course, the argument separator symbol changes consequently to the decimal separator in order to avoid conflict. The parser automatically adopts the argument separator "," if the decimal separator is point "." or adopts ";" if the decimal separator is "," comma.
The following table shows all possible combinations.

System setting	**Parser setting DP_SET**	**Decimal separator**	**Arguments separator**
Decimal is point "."	False	`3.141`	`COMB(35,12)`
Decimal is comma ","	False	`3,141`	`COMB(35;12)`
Any	True	`3.141`	`COMB(35,12)`

Symbols and operators.

This version recognizes more than 150 functions and operators

Function	Description	Note
+	addition	
-	subtraction	
*****	multiplication	
/	division	35/4 = 8.75
%	percentage	35% = 0.35
****	integer division	35\4 = 8
^	raise to power	3^1.8 = 7.22467405584208 (°)
| |	absolute value	|-5|=5 (the same as abs)
!	factorial	5!=120 (the same as fact)
abs(x)	absolute value	abs(-5)= 5
atn(x), atan(x)	inverse tangent	atn(pi/4) = 1
cos(x)	cosine	argument in radiant
sin(x)	sin	argument in radiant
exp(x)	exponential	exp(1) = 2.71828182845905
fix(x)	integer part	fix(-3.8) = 3
int(x)	integer part	int(-3.8) = −4
dec(x)	decimal part	dec(-3.8) = -0.8
ln(x), log(x)	logarithm natural	argument x>0
logN(x,n)	N-base logarithm	logN(16,2) = 4
rnd(x)	random	returns a random number between x and 0
sgn(x)	sign	returns 1 if x >0 , 0 if x=0, -1 if x<0
sqr(x)	square root	sqr(2) =1.4142135623731, also 2^(1/2)
cbr(x)	cube root	∀x, example cbr(2) = 1.2599, cbr(-2) = -1.2599
tan(x)	tangent	argument (in radian) x≠ k*π/2 with k = ± 1, ± 2…
acos(x)	inverse cosine	argument -1 ≤ x ≤ 1
asin(x)	Inverse sine	argument -1 ≤ x ≤ 1
cosh(x)	hyperbolic cosine	∀ x
sinh(x)	hyperbolic sine	∀ x
tanh(x)	hyperbolic tangent	∀ x
acosh(x)	Inverse hyperbolic cosine	argument x ≥ 1
asinh(x)	Inverse hyperbolic sine	∀ x
atanh(x)	Inverse hyperbolic tangent	-1 < x < 1
root(x,n)	n-th root (the same as x^(1/n)	argument n ≠ 0 , x ≥ 0 if n even , ∀x if n odd
mod(a,b)	modulus	mod(29,6) = 5 mod(-13,4) = 3
fact(x)	factorial	argument 0 ≤ x ≤ 170
comb(n,k)	combinations	comb(6,3) = 20 , comb(6,6) = 1
perm(n,k)	permutations	perm(8,4) = 1680 ,
min(a,b,...)	minimum	min(13,24) = 13
max(a,b,...)	maximum	max(13,24) = 24
mcd(a,b,...)	maximum common divisor	mcd(4346,174) = 2
mcm(a,b,...)	minimum common multiple	mcm(1440,378,1560,72,1650) = 21621600
gcd(a,b,...)	greatest common divisor	The same as mcd
lcm(a,b,...)	lowest common multiple	The same as mcm
csc(x)	cosecant	argument (in radiant) x≠ k*π with k = 0, ± 1, ± 2…
sec(x)	secant	argument (in radiant) x≠ k*π/2 with k = ± 1, ± 2…
cot(x)	cotangent	argument (in radiant) x≠ k*π with k = 0, ± 1, ± 2…
acsc(x)	inverse cosecant	
asec(x)	inverse secant	
acot(x)	inverse cotangent	
csch(x)	hyperbolic cosecant	argument x>0

Function	Description	Note
sech(x)	hyperbolic secant	argument x>1
coth(x)	hyperbolic cotangent	argument x>2
acsch(x)	inverse hyperbolic cosecant	
asech(x)	inverse hyperbolic secant	argument $0 \le x \le 1$
acoth(x)	inverse hyperbolic cotangent	argument x<-1 or x>1
rad(x)	radiant conversion	converts radiant into current unit of angle
deg(x)	degree sess. conversion	converst sess. degree into current unit of angle
grad(x)	degree cent. conversion	converts cent. degree into current unit of angle
round(x,d)	round a number with d decimal	round(1.35712, 2) = 1.36
>	greater than	return 1 (true) 0 (false)
>=	equal or greater than	return 1 (true) 0 (false)
<	less than	return 1 (true) 0 (false)
<=	equal or less than	return 1 (true) 0 (false)
=	equal	return 1 (true) 0 (false)
<>	not equal	return 1 (true) 0 (false)
and	logic and	and(a,b) = return 0 (false) if a=0 or b=0
or	logic or	or(a,b) = return 0 (false) only if a=0 and b=0
not	logic not	not(a) = return 0 (false) if $a \neq 0$, else 1
xor	logic exclusive-or	xor(a,b) = return 1 (true) only if $a \neq b$
nand	logic nand	nand(a,b) = return 1 (true) if a=1 or b=1
nor	logic nor	nor(a,b) = return 1 (true) only if a=0 and b=0
nxor	logic exclusive-nor	nxor(a,b) = return 1 (true) only if a=b
Psi(x)	Function psi	
DNorm(x,μ,σ)	Normal density function	$\forall x$, $\mu > 0$, $\sigma > 0$
CNorm(x,m,d)	Normal cumulative function	$\forall x$, $\mu > 0$, $\sigma > 1$
DPoisson(x,k)	Poisson density function	x >0, k = 1, 2, 3 ...
CPoisson(x,k)	Poisson cumulative function k = 1, 2,3 ...	x >0, k = 1, 2, 3 ...
DBinom(k,n,x)	Binomial density for k successes for n trials	k , n = 1, 2, 3…, k < n , $x \le 1$
CBinom(k,n,x)	Binomial cumulative for k successes for n trials	k , n = 1, 2, 3…, k < n , $x \le 1$
Si(x)	Sine integral	$\forall x$
Ci(x)	Cosine integral	x >0
FresnelS(x)	Fresnel's sine integral	$\forall x$
FresnelC(x)	Fresnel's cosine integral	$\forall x$
J0(x)	Bessel's function of 1st kind	$x \ge 0$
Y0(x)	Bessel's function of 2st kind	$x \ge 0$
I0(x)	Bessel's function of 1st kind, modified	x >0
K0(x)	Bessel's function of 2st kind, modified	x >0
BesselJ(x,n)	Bessel's function of 1st kind, nth order	$x \ge 0$, n = 0, 1, 2, 3…
BesselY(x,n)	Bessel's function of 2st kind, nth order	$x \ge 0$, n = 0,1, 2, 3…
BesselI(x,n)	Bessel's function of 1st kind, nth order, modified	x >0 , n = 0,1, 2, 3…
BesselK(x,n)	Bessel's function of 2st kind, nth order, modified	x >0 , n = 0,1, 2, 3…
HypGeom(a,b,c,x)	Hypergeometric function	-1 < x <1 a,b >0 $c \neq 0, -1, -2…$
PolyCh(x,n)	Chebycev's polynomials	$\forall x$, orthog. for $\mathbf{-1 \le x \le 1}$
PolyLe(x,n)	Legendre's polynomials	$\forall x$, orthog. for $\mathbf{-1 \le x \le 1}$
PolyLa(x,n)	Laguerre's polynomials	$\forall x$, orthog. for $\mathbf{0 \le x \le 1}$
PolyHe(x,n)	Hermite's polynomials	$\forall x$, orthog. for $\mathbf{-\infty \le x \le +\infty}$
AiryA(x)	Airy function Ai(x)	$\forall x$
AiryB(x)	Airy function Bi(x)	$\forall x$
Elli1(x)	Elliptic integral of 1st kind	$\forall \phi$, 0 < k < 1
Elli2(x)	Elliptic integral of 2st kind	$\forall \phi$, 0 < k < 1
Erf(x)	Error Gauss's function	x >0
gamma(x)	Gamma function	$\forall x$, $x \neq 0, -1, -2, -3…$ (x > 172 overflow error)
gammaln(x)	Logarithm Gamma function	x >0
gammai(a,x)	Gamma Incomplete function	$\forall x$ a > 0

Function	Description	Note
digamma(x) psi(x)	Digamma function	x ≠ 0, −1, −2, −3…
beta(a,b)	Beta function	a >0 b >0
betaI(x,a,b)	Beta Incomplete function	x >0 , a >0 , b >0
Ei(x)	Exponential integral	x ≠0
Ein(x,n)	Exponential integral of n order	x >0 , n = 1, 2, 3…
zeta(x)	zeta Riemman's function	x <-1 or x >1
Clip(x,a,b)	Clipping function	return a if x<a , return b if x>b, otherwise return x.
WTRI(t,p)	Triangular wave	t = time, p = period
WSQR(t,p)	Square wave	t = time, p = period
WRECT(t,p,d)	Rectangular wave	t = time, p = period, d= duty-cycle
WTRAPEZ(t,p,d)	Trapez. wave	t = time, p = period, d= duty-cycle
WSAW(t,p)	Saw wave	t = time, p = period
WRAISE(t,p)	Rampa wave	t = time, p = period
WLIN(t,p,d)	Linear wave	t = time, p = period, d= duty-cycle
WPULSE(t,p,d)	Rectangular pulse wave	t = time, p = period, d= duty-cycle
WSTEPS(t,p,n)	Steps wave	t = time, p = period, n = steps number
WEXP(t,p,a)	Exponential pulse wave	t = time, p = period, a= dumping factor
WEXPB(t,p,a)	Exponential bipolar pulse wave	t = time, p = period, a= dumping factor
WPULSEF(t,p,a)	Filtered pulse wave	t = time, p = period, a= dumping factor
WRING(t,p,a,fm)	Ringing wave	t = time, p = period, a= dumping factor, fm = frequency
WPARAB(t,p)	Parabolic pulse wave	t = time, p = period
WRIPPLE(t,p,a)	Ripple wave	t = time, p = period, a= dumping factor
WAM(t,fo,fm,m)	Amplitude modulation	t = time, p = period, fo = carrier freq., fm = modulation freq., m = modulation factor
WFM(t,fo,fm,m)	Frequecy modulation	t = time, p = period, fo = carrier freq., fm = modulation freq., m = modulation factor
Year(d)	year	d = dateserial
Month(d)	month	d = dateserial
Day(d)	day	d = dateserial
Hour(d)	hour	d = dateserial
Minute(d)	minute	d = dateserial
Second(d)	second	d = dateserial
DateSerial(a,m,d)	Dateserial from date	a = year, m = month, d = day
TimeSerial(h,m,s)	Timeserial from time	h = hour, m = minute, s = second
time#	system time	
date#	system date	
now#	system timestamp	
Sum(a,b,...)	Sum	sum(8,9,12,9,7,10) = 55
Mean(a,b,...)	Arithmetic mean	mean(8,9,12,9,7,10) = 9.16666666666667
Meanq(a,b,...)	Quadratic mean	meanq(8,9,12,9,7,10) = 9.30053761886914
Meang(a,b,...)	Arithmetic mean	meang(8,9,12,9,7,10) = 9.03598945281812
Var(a,b,...)	Variance	var(1,2,3,4,5,6,7) = 4.66666666666667
Varp(a,b,...)	Variance pop.	varp(1,2,3,4,5,6,7) = 4
Stdev(a,b,...)	Standard deviation	Stdev(1,2,3,4,5,6,7) = 2.16024689946929
Stdevp(a,b,...)	Standard deviation pop.	Stdevp(1,2,3,4,5,6,7) = 2
Step(x,a)	Haveside's step function	Returns 1 if x ≥ a , 0 otherwise

(°) the operation 0^0 (0 raises to 0) is not allowed here.
Symbol "!" is the same as "Fact"; symbol "%" is percentage; symbol "\" is integer division; symbol "| . |" is the same as Abs.
Logical functions and operators return 1 (true) or 0 (false)
Functions with (a,b,...) accept up to 20 arguments

Limits of 4.2:

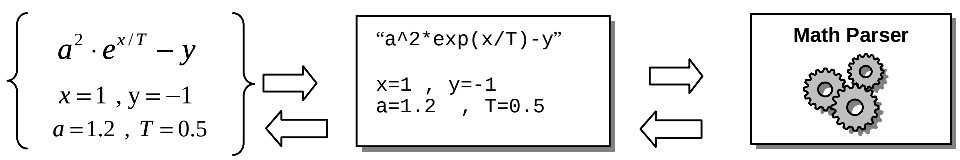
max operations/functions = 200; max variables = 100; max function types = 140; max arguments for function = 20; max nested functions = 20; max expressions stored at the same time = undefined. Increasing these limits is easy. For example, if you want to parse long strings up to 500 operations or functions with 60 max arguments, and max 250 variables, simply set the variable

```
Const HiVT   As Long = 250
Const HiET   As Long = 500
Const HiARG  As Long = 60
```

The ET table can now contain up to 400 rows, each of one is a math operator or function

Main Applications

In some instances, you might want to allow users to type their own numeric expression in a TextBox and then evaluate it. The user may also type symbolic variables “x”, “y”, “a”, “b”, etc. in the expression and then calculate it for several different numeric values assigned to the variables.
The expression is created at runtime and will be interpreted and calculated by your VB program. This Parser is the black-box that just performs this task



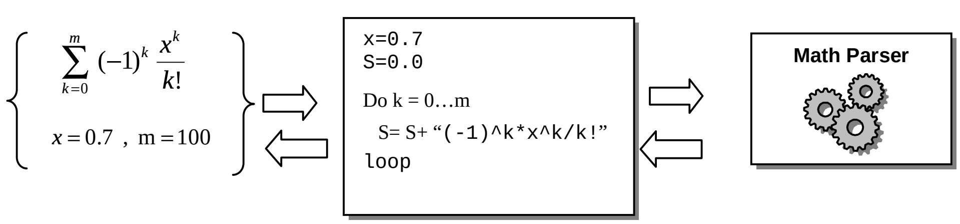
a math calculation in math language…

…is translated into plain ascii text…

…and finally interpreted and calculated by the parser

The final response that is retrieved to the user can be either a numeric result or an error message

Another case: the user wants to approximate a series. Usually the input of these operations is a function defined by mathematical expressions. This parser is suitable for performing many computation of the same expression with different variable values (loop). The interface program implements the algorithm performing the series and demanding the evaluation of the function to the MathParser. For programmers, the complication of the function calculation is totally transparent.



a math calculation in math language…

…is translated into plain ascii text, repeating the evaluation of f(x, k) for k =0, 1, 2,…m. Each partial result is accumulated and the total is retrieved to the user

…interpreted (at the first calling) and calculated for different values of k, and for the same value x

The final response that is retrieved to the user can be either a numeric result or an error message

The isolation between the main algorithm and the function calculation is the main goal of the MathParser.

Class

This software is structured following the object programming rules, consisting of a set of methods and properties

Class		
clsMathParser	**Methods**	
	StoreExpression(f)	Stores, parses, and checks syntax errors of the formula "f"
	Eval	Evaluates expression
	Eval1(x)	Evaluates mono-variable expression f(x)
	EvalMulti(x(), id)	Evaluates expression for a vector of values
	ET_Dump	Dumps the internal ET table (debug only)
	Properties	
	Expression	Gets the current expression stored (R)
	VarTop	Gets the top of the var array (R)
	VarName(i)	Gets name of variable of index=i (R)
	Variable(x)	Sets/gets the value of variable passes by index/name (R/W)
	AngleUnit	Sets/gets the angle unit of measure (R/W)
	ErrorDescription	Gets error description (R)
	ErrorID	Error message number
	ErrorPos	Error position
	OpAssignExplicit	Option: Sets/gets the explicit variables assignment (R/W)
	OpUnitConv	Enable/disable unit conversion
	OpDMSConv	Enable/disable DMS angle conversion

Note. The old properties *VarValue* and *VarSymb* are obsolete, being substituted by the *Variable* property. However, they are still supported for compatibility.
R = read only, R/W = read/write

Methods

StoreExpression Stores, parses, and checks syntax errors. Returns True if no errors are detected; otherwise it returns False

Sintax

object. **StoreExpression**(ByVal *strExpr* As String) As Boolean

Where:

Parts	**Description**
Object	Obligatory. It is always the clsMathParser object.
strExpr	Math expression to evaluate.

This method can be invoked after a new instance of the class has been created. It activates the parse routine.
Example

```
Dim OK as Boolean
Dim Fun As New clsMathParser
......
  OK = Fun.StoreExpression(txtFormula)
```

Notes
Typical mixed math-physical expressions are

```
1+(2-5)*3+8/(5+3)^2   (a+b)*(a-b)    (-1)^(2n+1)*x^n/n!    x^2+3*x+1
256.33*Exp(-t/12 us)    (3000000 km/s)/144 Mhz    0.25x+3.5y+1   space/time
```

Variables can be any alphanumeric string and must start with a letter:

```
 x  y  a1  a2  time  alpha   beta
```

Implicit multiplication is not supported because of its intrinsic ambiguity. So "xy" stand for a variable named "xy" and not for x*y. The multiplication symbol "*" cannot generally be omitted.
It can be omitted only for coefficients of the classic math variables x, y, z. It means that strings like 2x and 2*x are equivalent:

```
   2x  3.141y  338z^2   ⇔   2*x  3.141*y  338*z^2
```

On the contrary, the following expressions are illegal: 2a, 3(x+1), 334omega

Constant numbers can be integers, decimal, or exponential:

```
2  -3234  1.3333  -0.00025   1.2345E-12
```

Physical numbers are alphanumeric strings that must begin with a number:

```
 "1s" for 1 second   "200m" for 200 meters  "25kg" for 25 kilograms.
```

For better reading they may contain a blank:

```
"1 s"   "200 m"   "25 kg"   "150 MHz"   "0.15 uF"   "3600 kOhm"
```

They may also contain the following **multiplying factor**:

```
T=10^12  G=10^9  M=10^6  k=10^3  m=10^-3  u=10^-6  n=10^-9  p=10^-12
```

Eval	Evaluates the previously stored expression and returns its value.

Sintax

object. **Eval**() As Double

Where:

Parts	**Description**
object	Obligatory. It is always the clsMathParser object.

This method substitutes the variables values previously stored with the **Variable** property and performs numeric evaluation.

Notes
The error detected by this method is any computational domain error. This happens when variable values are out of the domain boundary. Typical domain errors are:

Log(-x) Log(0) $\sqrt{-x}$ x/0   arcsin(2)

They cannot be intercepted by the parser routine because they are not syntax errors but depend only on the wrong numeric substitution of the variable. When this kind of error is intercepted, this method raises an error and its text is copied into the *ErrDescription* property of the clsMathParser object and also into the same property of the global Err object.

Eval1 Evaluates a monovariable function and returns its value

Sintax

*object***. Eval1**(ByVal x As Double) As Double

Where:

Parts	**Description**
object	Obligatory. It is always the clsMathParser object.
x	Variable value

Notes
This method is a simplified version of the general **Eval** method
It is adapted for monovariable function f(x)

```
Dim OK As Boolean, x As Double, f As Double
Dim Funct As New clsMathParser

On Error Resume Next

OK = Funct.StoreExpression("(x^2+1)/(x^2-1)")  'parse function

If Not OK Then
    Debug.Print Funct.ErrorDescription
Else
    x = 3
    f = Funct.Eval1(x)    'evaluate function value

    If Err = 0 Then
        Debug.Print "x="; x, "f(x)="; f
    Else
        Debug.Print "x="; x, "f(x)="; Funct.ErrorDescription
    End If
End If
```

Note in this case how the code is simple and compact. This method is also about 11% faster than the general **Eval** method.

EvalMulti Substitutes and evaluates a vector of values

Sintax

*object***.EvalMulti**(ByRef VarValue() As Double, Optional ByVal VarName)

Where:

Parts	**Description**
Object	Obligatory. It is always the clsMathParser object.
VarValue()	Vector of variable values
VarName	Name or index of the variable to be substituted

Notes
This property is very useful to assign a vector to a variable obtaining of a vector values in a very easy way. It is also an efficient method, saving more than 25% of elaboration time.
The formula expression can have several variables, but only one of them can receive the array.
Let’s see this example in which we will compute 10000 values of the same expression in a flash.
Evaluate with the following

```
f(x,y,T,a) = (a^2*exp(x/T)-y)
for: x = 0...1 (1000 values),  y = 0.123 , T = 0.4 , a = 100
```

```
Sub test_array()
Dim x() As Double, F, Formula, h
Dim Loops&, i&
Dim Funct As New clsMathParser
'-----------------------------------------------
```

```
Formula = "(a^2*exp(x/T)-y)"
Loops = 10000
x0 = 0
x1 = 1
h = (x1 - x0) / Loops
'load x-samples
ReDim x(Loops)
For i = 1 To Loops
    x(i) = i * h + x0
Next i

If Funct.StoreExpression(Formula) Then
    On Error GoTo Error_Handler
    T0 = Timer

    Funct.Variable("y") = 0.123
    Funct.Variable("T") = 0.4
    Funct.Variable("a") = 100

    F = Funct.EvalMulti(x, "x")   'evaluate in one shot 10000 values

    T1 = Timer - T0
    T2 = T1 / Loops
    Debug.Print T1, T2

Else
    Debug.Print Funct.ErrorDescription
End If
Exit Sub
Error_Handler:
    Debug.Print Funct.ErrorDescription
End Sub
```

ET_Dump	Return the ET internal table (only for debugging)

Sintax

*object***. ET_Dump**(ByRef Etable as Variant)

Where:

Parts	**Description**
object	Obligatory. It is always the clsMathParser object.
ETable	Array containing the ET table

Notes
This method copies the internal ET Table into an array (n x m). The first row (0) contains the column headers. The array must be declares as a Variant undefined array.

```
Dim Etable()
```

For details see example 8.

Properties

Expression	Returns the current stored expression

Sintax

*object***. Expression**() As String

Where:

Parts	**Description**
object	Obligatory. It is always the clsMathParser object.

This property is useful to check if a formula is already stored. In the example below, the main routine skips the parsing step if the function is already stored, saving a good deal of time.

```
...
...
    If Funct.Expression <> Formula Then
        OK = Funct.StoreExpression(Formula)  'parse function
    End If
....
...
```

VarTop get the top of the variable array

Sintax

object. **VarTop**() As Long

Where:

Parts	**Description**
object	Obligatory. It is always the clsMathParser object.

Notes
It returns the total number of different symbolic variables contained in an expression.
This is useful in order to dynamically allocate the variables array.
Example, for the following expression:

```
"(x^2+1)/(y^2-y-2)"
```

We get

2 = object.VarTop

VarName returns the name of the i-th variable.

Sintax

object. **VarName**(ByVal Index As Long) As String

Where:

Parts	**Description**
object	Obligatory. It is always the clsMathParser object.
Index	Pointer of variable array. It must be within 1 and VarTop

Example
This code prints all variable names contained in an expression, with their current values

```
Dim Funct As New clsMathParser

Funct.StoreExpression "(x^2+1)/(y^2-y+2)"

For i = 1 To Funct.VarTop
    Debug.Print Funct.VarName(i); " = "; Funct.Variable(i)
Next
```

Variable	sets or gets the value of a variable by its symbol or its index.

Sintax

*object***. Variable**(ByVal Name) As Double

Where:

Parts	**Description**
object	Obligatory. It is always the clsMathParser object.
Name	symbolic name or index of the variable.

Example
Assign the value 2.3333 to the variable of the given index. The parser builds this index reading the formula from left to right; the first variable encountered has index = 1; the second one has index = 2 and so on.

```
Dim f As Double
Dim Funct As New clsMathParser
Funct.StoreExpression "(x^2+1)/(y^2-y+2)" 'parse function

Funct.Variable(1) = 2.3333
Funct.Variable(2) = -0.5

f = Funct.Eval
Debug.Print "f(x,y)=" + Str(f)
```

But we could write as well

```
Funct.Variable("x") = 2.3333
Funct.Variable("y") = -0.5
```

Using index is the fastest way to assign a variable value. Normally, it is about 2 times faster.
But, on the other hand, the symbolic name assignment is easier and more flexible.

AngleUnit	Sets/gets the angle unit of measure

Sintax

*object***. AngleUnit** As String

Where:

Parts	**Description**
object	Obligatory. It is always the clsMathParser object.
AngleUnit	Sets/gets the current angle unit (read/write)

The default unit of measure of angles is “RAD” (Radian), but we can also set “DEG” (sexagesimal degree) and “GRAD” (centesimal degree)

Example

```
Dim f As Double
Dim Funct As New clsMathParser

Funct.StoreExpression "sin(x)"    'parse function
Funct.AngleUnit= "DEG"

f = Funct.Eval1(45)    ' we get f= 0.707106781186547
f = Funct.Eval1(90)    ' we get f= 1
```

The angles returned by the functions are also converted.

```
Funct.StoreExpression "asin(x)"    'parse function
Funct.AngleUnit= "DEG"

f = Funct.Eval1(1)    ' we get f= 90
f = Funct.Eval1(0.5)  ' we get f= 30
```

Note: angle setting only affects the following trigonometric functions:
sin(x) , cos(x) , tan(x), atn(x) , acos(x) , asin(x) , csc(x) , sec(x) , cot(x), acsc(x) , asec(x) , acot(x) , rad(x) , deg(x) , grad(x)

ErrDescription Returns any error detected.

Sintax

object. **ErrorDescription**() As String

Where:

Parts	**Description**
object	Obligatory. It is always the clsMathParser object.
ErrorDescription	Contains the error message string

Note
This property contains the error description detected by internal routines. Errors are divided into two groups: syntax errors and evaluation errors (or domain errors).
If a domain error is intercepted, an error is generated, so you can also check the global Err object.
For a complete list of the error descriptions see “Error Messages”.

ErrorID Returns the error message number.

Sintax

object. **ErrorID** () As Long

Where:

Parts	**Description**
object	Obligatory. It is always the clsMathParser object.
ErrorID	Returns the error message number

This property contains the error number detected by internal routines. It is the index of the internal error table ErrorTbl()
For a complete list of the error numbers see “Error Messages”.

ErrorPos Returns the error position.

Sintax

object. **ErrorPos** () As Long

Where:

Parts	**Description**
object	Obligatory. It is always the clsMathParser object.
ErrorPos	Returns the error position

This property contains the error position detected by internal routines.
Note that the parser cannot always detect the exact position of the error
For example

```
root(-5,2)  returns "Evaluation error < root(-5, 2) > at pos: 5"
```

```
comb(50,2.2)  returns " Evaluation error < comb(50, 2.2) > at pos: 5"
acos(0.5)+acos(2)  returns  " Evaluation error < acos(2) > at pos: 15"
```

As we can see, the ErrorPos always points to the wrong function no matter which the wrong argument is.

OpAssignExplicit Enable/disable the explicit variable assignment.

Sintax

*object***. OpAssignExplicit**() As Boolean

Where:

Parts	**Description**
object	Obligatory. It is always the clsMathParser object.
OpAssignExplicit	True/False. If True, forces the explicit variables assignement.

All variables contained in a math expression are initialized to zero by default. The evaluation methods usually use these values in the substitution step without checking if the variable values have been initialized by the user or not. If you want to force explicit initialization, set this property to “true”. In that case, the evaluation methods will perform the check: If one variable has never been assigned, the MathParser will raise an error. The default is “false”.

Example. Compute f(x,y) = x^2+y , assigning only the variable x = −3

```
Sub test_assignement1()
Dim x As Double, F, Formula
Dim Funct As New clsMathParser
    Formula = "x^2+y"
    If Not Funct.StoreExpression(Formula) Then GoTo Error_Handler
    Funct. OpAssignExplicit = False
    Funct.Variable("x") = -3
    F = Funct.Eval
    If Err <> 0 Then GoTo Error_Handler
    Debug.Print "F(x,y)= "; Funct.Expression
    Debug.Print "x= "; Funct.Variable("x")
    Debug.Print "y= "; Funct.Variable("y")
    Debug.Print "F(x,y)= "; F
Exit Sub
Error_Handler:
    Debug.Print Funct.ErrorDescription
End Sub
```

If *OpAssignExplicit = False* then the response will be

```
F(x,y)= x^2+y
x= -3
y=  0
F(x,y)=  9
```

As we can see, variable y, never assigned by the main program, has the default value = 0, and the eval method returns 9.
On the contrary, if *OpAssignExplicit = True*, then the response will be the error:

“Variable <y> not assigned”

OpUnitConv	Enable/disable unit conversion.

Sintax

*object***. OpUnitConv** () As Boolean

Where:

Parts	**Description**
object	Obligatory. It is always the clsMathParser object.
OpUnitConv	True/False. If True, enables the unit conversion.

This option switches off the unit conversion if not needed. The default is “True”.

OpDMSConv	Enable/disable angle DMS conversion.

Sintax

*object***. OpUnitConv** () As Boolean

Where:

Parts	**Description**
object	Obligatory. It is always the clsMathParser object.
OpDMSConv	True/False. If True, enables the angle dms conversion.

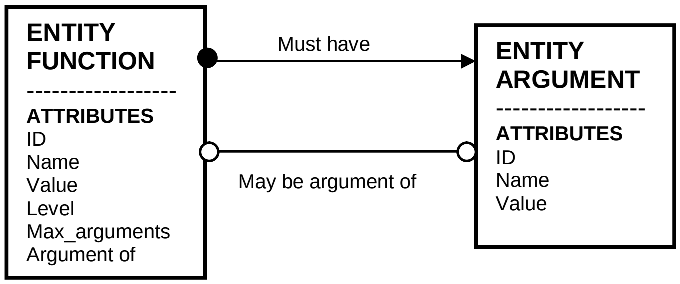
This option switches off the angle DMS conversion if not needed. The default is “True”.

Parser

It is the heart of the algorithm. The Parser algorithm reads the input math expression string and translates the conventional symbols into a conceptual structured database.
This database, designed in the ET table, contains all the operational information to perform the calculation.

The Conceptual Model Method

In the language of structured data modeling we can say that this algorithm translates any arithmetic or algebraic input expression to the follow conceptual data model:



This model expresses the following sentences:

1. Each arithmetic or algebraic expression is composed by functions. Common binary operators like +, -, *, / are considered as functions with two variables. For example: ADD(a,b) instead of a+b , DIV(a,b) instead of a/b and so on.
2. Each function must have one value and one or more arguments.
3. Each argument has one value.
4. A function may be the argument of another function

The algorithm assigns a priority level to each function, based on an internal level-table, respecting the usual order of the mathematical operators:

Level=	1	2	3	10	+10	-10
Functions=	+ -	* /	^	Any functions	([{	)] }

This level is increased by 10 every time the parser finds a left bracket (no matter which type) and, on the contrary, is decreased by 10 with a right bracket.

Example
The following example explains how this algorithm works.

Suppose you evaluate the expression: `(a+b)*(a-b)` , with: `a = 3 ,` and: `b = 5`

We call the `parse()` routine with the following arguments:

ExprString = "(a+b)*(a-b)"
Variable(1) = 3 'value of variable a
Variable(2) = 5 'value of variable b

The Variable Table and Expression Table

The parser builds the two following tables: Variable Table (**VT**) and Expression Table (**ET**)
The parser has recognized and indexed 2 variables. It assigns 1 to the first variable recognized from left to right; 2 to the second one, and so on. Note that the variable name is stored only for better debugging. It is not really necessary for the algorithm process.

Variables table

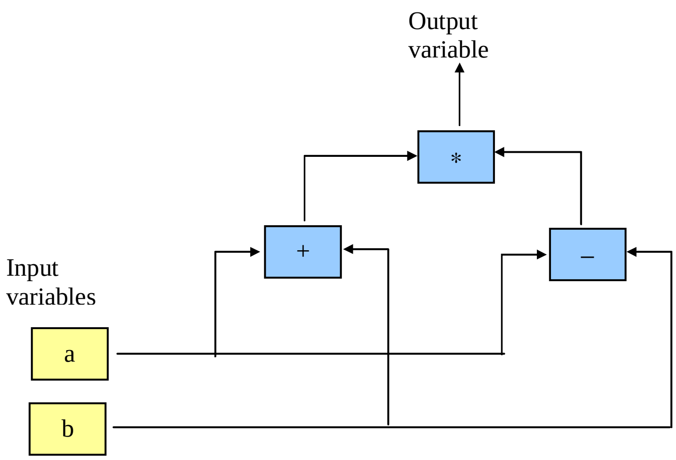
Id	Variable
1	a
2	b

Formula Structure Table   ET

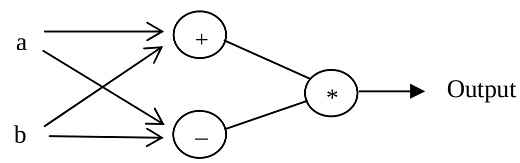
Id	Fun	Max Arg	Arg(1). Id	Arg(1). Name	Arg(1). Value	Arg(2). Id	Arg(2). Name	Arg(2). Value	ArgOf	IndArg	Value	Level	Offset	Seq
1	+	2	1	a		2	b		2	1		11	3	1
2	*	2	0			0			0	0		2	6	3
3	-	2	1	a		2	b		2	2		11	9	2

The Parser has recognized 3 functions (operators: +, *, -) and assigned the priority level of 11, 2, 11 respectively. The operators + and - have normally a level of L =1 (the lowest). Because of brackets, their level becomes L = 10+1 = 11.
The function "+", ID =1, has two arguments: "a" and "b", respectively as shown in columns "Arg name". The result of function "+" is the first argument of function "*" (ID=2), as indicated in the columns ArgOf = 2 and IndArg = 1.
The function "-", ID =3, has two arguments: "a" and "b". The result of function "-" is the second argument of function "*" (ID=2), as indicated in the columns ArgOf = 2 and IndArg = 2.
Finally, the function "*" is not the argument of any other function, as shown by the corresponding ArgOf = 0; so its result is the result of the given expression.
The sequence (Seq) column is the key for speed-up computing. Practically, it is the secondary index of the table; reading it from top to bottom, we have the exact sequence of functions that we must execute. In the example we must execute first the function with ID =1, "+", (in row 1), then we must execute the function with ID =3, "-", (in row 3); and finally we execute the function with ID =2, "*", (in row 2).
The column Seq is generated from the level of each function stored in the Level column. The score=1 is for the highest level; if there are more than one value, (in the example we have two level 11 in rows 1 and 3) the higher row is the winner.
The same rule is applied to the remaining rows, and so on. The last Seq is for the lowest level function.
Note that, in this step, no value cells are filled. This task is performed by the `Eval()` subroutine.
The Parser reads the string from left to right and substitutes the first variable found with the first value of VarVector(), the second variable with the second value and so on. All the functions recognized by this Parser have two variables at most.

A graphical representation of ET table is shown below



Or, more synthetically, by the following binary tree



Eval

This routine performs two simple actions:
1. substitutes all symbolic variables (if present) with their numeric values;
2. performs the computation of all functions according to their sequence (Seq column).

In the above example, this routine inserts values 3 and 5 into the "Arg value" columns for the corresponding variables "a" and "b".

Id	Fun	Max Arg	Arg(1). Id	Arg(1). Name	Arg(1). Value	Arg(2). Id	Arg(2). Name	Arg(2). Value	ArgOf	IndArg	Value	Level	Offset	Seq
1	+	2	1	a	3	2	b	5	2	1		11	3	1
2	*	2	0			0			0	0		2	6	3
3	-	2	1	a	3	2	b	5	2	2		11	9	2

After this substitution, the algorithm begins the computation, one function at a time, from top to bottom. In this case the order of computation is 1, 3, 2, as indicated in the last column "Seq".
The operations performed are (in sequence): row 1, 3, 2
row=1, 3+5= 8 ,
is assigned to the function indexed by the corresponding field ArgOf=2 and by the argument IndArg=1.
row=3; 3-5= -2 ,
is assigned to the function indexed by the corresponding field ArgOf=2 and by the argument IndArg=2
row=2; 8*(-2)= -16 ,
because ArgOf=0 the routine returns this value as the result of the evaluation of the expression.

Id	Funz	Max Arg	Arg(1). Id	Arg(1). Name	Arg(1). Value	Arg(2). Id	Arg(2). Name	Arg(2). Value	ArgOf	IndArg	Value	Level	Offset	Seq
1	+	2	1	a	3	2	b	5	2	1	8	11	3	1
2	*	2	0		8	0		-2	0	0	-16	2	6	3
3	-	2	1	a	3	2	b	5	2	2	-2	11	9	2

Result

See Appendix for other examples:

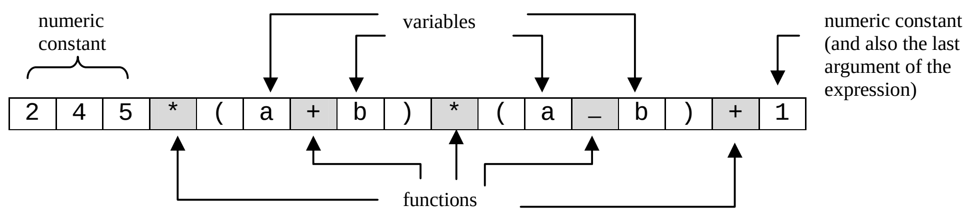
Parser Algorithm step-by-step

Example1: Suppose we compute the following mathematical expression: `245*(a+b)*(a–b)+1`
The parser begins to examine each character, from left to right, applying the following rules.
Rules:
Arithmetic functions are the following one-char symbols: `+ – * / ^`
Elementary functions are any substring ending with a left parenthesis (matching in the given function list)
Elementary functions are also the following symbols ! (factorial) and | (absolute)
Functions can have 1, 2, or more arguments
Functions have a specific priority level
Parentheses are `{ [ ( ) ] }` without any difference. Each open parenthesis increments by 10 the priority level; each closed decrements it by 10.
Any substring that does not contain the above character must be a symbolic variable, a numeric constant or a literal constant
Any symbolic variable, numeric constant or literal constant must be the argument of a function (by default, the left argument of the following function - except the last argument, which must be the right one)

Parser Steps

By this rules, the parser recognizes 5 functions and begins to build the structured ET tables

1° Parser Step



ID	Fun	MaxArg	Arg(1).	Arg(2).	funct level	parenthesis level	Level
1	*	2	245		2	0	2
2	+	2	a		1	10	11
3	*	2	b		2	0	2
4	–	2	a		1	10	11
5	+	2	b	1	1	0	1

Level=Funct Level + Parenthesis Level

Variables or constants are assigned to the left arguments of the next function, except for the last variable or constant, which must be assigned to the second argument of the last function.
If a function has only one argument, then only the left argument is filled.

2° Parser step.

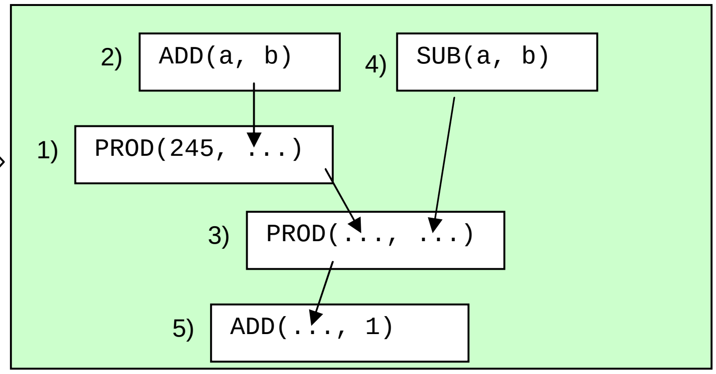
Fill the second argument

ID	Fun	MaxArg	Arg(1).	Arg(2).	funct level	parenthesis level	Level
1	*	2	245	a	2	0	2
2	+	2	a	b	1	10	11
3	*	2	b	a	2	0	2
4	–	2	a	b	1	10	11
5	+	2	b	1	1	0	1

At the 2° step, the parser fills the second argument (for functions that need 2 arguments). The rules are simple.
Each left argument of a given function is also a right argument of the next function, except the first and the last.
Starting from bottom to top, the parser simply copies each Arg(1) into the Arg(2) of the previous function.

3° Parser step.
Here, the relationships between functions are built.
Rule: Each function must be an argument of its adjacent function with the lowest priority level. It must be the left argument for the right adjacent function and vice versa.

`245*(a–b)*(a–b)+1`



Relationships:

Function 2 ADD (a,b) ⇒ Argument 2° of Function 1 PROD(245, ...)
Function 1 PROD(245, ...) ⇒ Argument 1° of Function 3 PROD(...., ...)
Function 4 SUB (a,b) ⇒ Argument 2° of Function 3 PROD(...., ...)
Function 3 PROD(...,...) ⇒ Argument 1° of Function 5 ADD(...., 1)

ID	Fun	MaxArg	Arg(1).	Arg(2).	ArgOf	Arg ID	Level
1	*	2	245	a	3	1	2
2	+	2	a	b	1	2	11
3	*	2	b	a	5	1	2
4	–	2	a	b	3	2	11
5	+	2	b	1	0	0	1

Function 5 is argument of no other function. So its result is the final result of the expression and the process ends.
Rules for building relations
Starting from the higher level function "k" (if many rows have the same level, choose one of them, for example the upper), search for the adjacent row "j", upper, lower or both, having ArgOf = 0. If two rows exist, choose the one with the higher level. If they have the same level, choose one of them (for example, the upper one).
Then the ArgOf(k) = j
If j > k then ArgId(k) = 1 (left argument); elseif j < k then ArgId(k) = MaxArg (right argument for function having 2 arguments; left function for function having one argument)
The following example explains these rules better.

ID	Fun	MaxArg	ArgOf	Arg ID	Level
1	*	2	0	0	2
2	+	2	1	2	11
3	*	2	0	0	2
4	–	2	0	0	11
5	+	2	0	0	1

Start from this row, k=2
we select the upper and lower adjacent rows, having ArgOf = 0.
Both rows have Level=2; then we choose the upper j=1.
So we get ArgOf =1 and ArgID = MaxArg =2.

(Note that the row k=2, from now on, will not be selected, because of its ArgOf >0)

ID	Funz	MaxArg	ArgOf	Arg ID	Level
1	*	2	0	0	2
2	+	2	1	2	11
3	*	2	0	0	2
4	–	2	3	2	11
5	+	2	0	0	1

Continue selecting the row with the highest Level with ArgOf = 0. Now we choose k = 4
we select the upper and lower adjacent rows, having ArgOf = 0,
we choose the upper j=3, because it has a higher level.
This gives ArgOf = 3 and ArgID = 2.

ID	Funz	MaxArg	ArgOf	Arg ID	Level
1	*	2	3	1	2
2	+	2	1	2	11
3	*	2	0	0	2
4	–	2	3	2	11
5	+	2	0	0	1

Continue selecting the row with the highest Level, with ArgOf =0. Both row 1 and 3 have the same Level =2. We choose k =1
We select the lower adjacent rows, having ArgOf = 0; we choose j =3.
(note that row=2 cannot be selected because it has ArgOf >0)
This gives ArgOf =3 and ArgID =1.

ID	Funz	MaxArg	ArgOf	Arg ID	Level
1	*	2	3	1	2
2	+	2	1	2	11
3	*	2	5	1	2
4	–	2	3	2	11
5	+	2	0	0	1

Continue selecting the row with the highest Level, with ArgOf =0. . We choose k =3
We select the lower adjacent rows, having ArgOf = 0; we choose j =5.
(note that row =1,2,4 cannot be selected because it has ArgOf >0)
This gives ArgOf =5 and ArgID =1.

ID	Funz	MaxArg	ArgOf	Arg ID	Level
1	*	2	3	1	2
2	+	2	1	2	11
3	*	2	5	1	2
4	–	2	3	2	11
5	+	2	0	0	1

Now we select the only row k =5 with ArgOf =0
Because there are no other rows with the ArgOf =0, the process ends.

This means that the row ID =5 is not an argument of any other function. So its results is the final result of the expression

4° Parser step

This builds the sequence index; resulting in a column giving the exact sequence of operations that must be performed. This can be easily obtained from the level information. But, in order to save time during the evaluation, we add this redundant information.
Rule: the first function (operation) that must be executed is the highest-level function. If there exist more than one row, we choose the upper one. Then we repeat this process till the lowest-level row.

ID	Fun	Level
1	*	2
2	+	11
3	*	2
4	–	11
5	+	1

Sequence	Fun	Index Fun
1	+	2
2	–	4
3	*	1
4	*	3
5	+	5

In order to save space the Sequence Table is attached to the ET. It can be easily built adding the index column to the first table

ID	Fun	Level	Index
1	*	2	2
2	+	11	4
3	*	2	1
4	–	11	3
5	+	1	5

5° Parser Step
This finds primary arguments. This step is useful for saving computation time. During the evaluation performed by the Eval() subroutine, all symbolic variables "a" and "b" will be substituted by their numeric values. In the ET table of the example below we should perform 8 substitutions

ID	Funz	MaxArg	Arg(1).	Arg(2).	ArgOf	Arg ID	Level	Index
1	*	2	245	a	3	1	2	2
2	+	2	a	b	1	2	11	4
3	*	2	b	a	5	1	2	1
4	–	2	a	b	3	2	11	3
5	+	2	b	1	0	0	1	5

We note that each argument which comes from another function will be replaced during computation. We say that it is a "non-primary argument".
Rule: Clear all arguments indicated by ArgID of functions indicated by ArgOf
In the table above, the step will delete the argument 1 of the row 3 ; the argument 2 of the row 1, the argument 1 of the row 5, and the argument 2 of the row 3. Arguments that remain are "primary".

ID	Funz	MaxArg	Arg(1).	Arg(2).	ArgOf	Arg ID	Level	Index
1	*	2	245		3	1	2	2
2	+	2	a	b	1	2	11	4
3	*	2			5	1	2	1
4	–	2	a	b	3	2	11	3
5	+	2		1	0	0	1	5

We see that only 4 substitutions are really necessary. The time saved is 50%

This ET table contains all the information to perform the computing of the original expression at the highest speed

```
245*(a+b)*(a–b)+1
```

See Appendix for several other examples

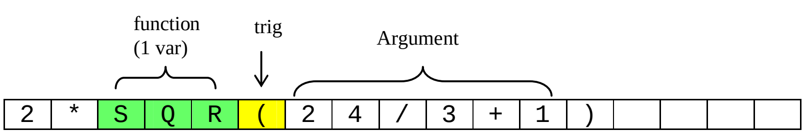
How to parse functions

We have to distinguish among monovarible, bivariable and multivariable functions.

Monovariable functions.

This group comprises many functions like: sin(x), cos(x), sqr(x) , etc.
Rule. The parser recognizes a function when it finds the character "(". Then, it extracts the sub-string before the parenthesis and checks if the name is in the monovariable function list. If so, it immediately adds a new record in the ET table for the recognized function. Example:

```
2*SQR(24/3+1)
```



The parser recognizes the “(“ chararcter at the 6th position. Then it extracts all the alphanumeric characters before this position. The “SQR” sub-string, is searched on the monovariable function list. The name is recognized as a monovariable function and added to the ET table. We note that functions are recognized before finding its argument (enclosed between brackets), which will be caught in the next step.

The Structured Table is:

ID	Fun	MaxArg	Arg(1).	Arg(2).	ArgOf	Arg ID	Level	Index
1	*	2	2	0	0	0	2	3
2	sqr	1	0	0	1	2	10	4
3	/	2	24	3	4	1	12	2
4	+	2	3	1	2	1	11	1

Exceptions: functions x! and |x|

The Parser can also recognize the symbols “!” (exclamation mark) and “|” (pipe) respectively for factorial and absolute value. Of course you can use the *Abs()* and *Fact()* function as well but, because these symbols are largely adopted, we have taught the parser to detect them, even though they are exceptions to the general rule above. Let’s see how they work.

Symbol “!”
In this case, the argument comes before the function symbols, contrary to the usual rule where the argument follows the function name

`FACT(x)` ⇔ `x!`

To manage this case, the parser saves the argument x until the symbol “!” is intercepted, and then, registers the Fact() function as usual. In the next step, the argument x is loaded into the ET table.

Symbol “|”
In this case, the argument-position rule is still valid

`ABS(x)` ⇔ `|x|`

Argument follows both the function name and the function symbol. But, unlike brackets, the pipe symbols are not oriented at all. So we cannot say if a symbol “|” is opening or closing the function.
For example:

`Abs(x-3*Abs(x-1)+1) ,` we count two Abs() functions and two left brackets

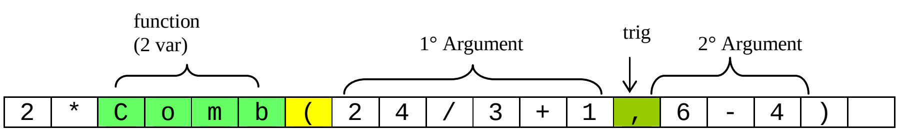
`|x-3*|x-1|+1|` we count two Abs() functions and four pipe symbols “|”. But which are open and which are closed?

To solve this ambiguity, the parser applies the following rule:
Rule: if the pipe “|” is preceded by any arithmetic operator (or nothing), then it is equivalent to the open bracket “(”. On the contrary, if the pipe “|” is preceded by a numeric or alphanumeric string, then it is equivalent to the closed bracket “)”

Bivariable functions.

This group comprises a few functions like : comb(a,b), max(a,b), min(a,b) , mod(a,b), etc.
When the search on the monovariable function list is negative, the Parser looks into the bivariable function list. If the function is found, the name is recognized as a bivariable function and saved in a buffer. Examples

`2*Comb(24/3+1 , 6-4)`



Bivariable functions can be treated as the arithmetic operators `+ * /` etc. The only difference is that its arguments come after the function identification string. In the case of arithmetic operators, the first argument comes before and the second one comes after. That is:

```
Add(a,b))    <==>  a + b
Comb(a,b)    <==>  a Comb b
```

The above example can be translated into the following expression

```
Comb(24/3+1 , 6-4)   <==>  (24/3+1) Comb (6-4)
```

Where the symbols "," are substituted by the "Comb" name
The symbols "," triggers the registration of the Comb function
The rules are:
1) Reading from left to right, the parser detects a recognizes the Comb function
2) It gets from the internal function list how many arguments the function needs (1 or 2)
3) Because the Comb function needs 2 arguments, it saves the function name but does not add any new record
4) When the Parser detects the "," symbol, it adds a new record in the ET table, assigning the function name which had been previously saved.

So the parsing of a bi-variable function takes place in two steps: 1°) recognize and save the function name. 2°) add the function record when the argument separator symbol "," is detected.

Following the above rules, the Structured Table is:

ID	Fun	MaxArg	Arg(1).	Arg(2).	ArgOf	Arg ID	Level	Index
1	*	2	2	24	0	0	2	2
2	/	2	24	3	3	1	12	3
3	+	2	3	1	4	1	11	5
4	Comb	2	1	6	1	2	10	4
5	−	2	6	4	4	2	11	1

As we can see, the 2nd argument of Comb(a,b) function comes from the "+" and "−" operations

Multivariable functions

As we known, multi-variables functions can be "naturally" described by a binary tree and implemented directly by a descendent parse algorithm. Trees are natural structures for representing certain kinds of hierarchical data. On the contrary, they do not easy adapt themselves to the plain table algorithm of the MathParser.
Arnaud de Grammont, for the first time intuited that functions with more than 2 variables could also be parsed in a plain table in the same way as other bivariable functions. He demonstrated this making the complex version of MathParser, one sophisticated class able to manage up to 6-variable functions.
The clever trick for parsing this kind of function is to guess a multivariable function as a bivariable operator (like + * / etc.), repeating the insertion of the same operator for each argument of the function itself, except the last one. Hard to understand? Never mind. Let's see this example

foo(a, b, c)	is translated into ⇒	a **foo**$_1$ b **foo**$_2$ c
foo(x^2+1, x/2, c+2)	is translated into ⇒	(x^2+1) **foo**$_1$ (x/2) **foo**$_2$ (c+2)

As we can see, the function name is repeated for each “,” argument delimiter encountered along the string. Example, for a function of 6 arguments we have

foo(a1, a2, a3, a4, a5, a6) ⇒ a1 **foo**$_1$ a2 **foo**$_2$ a3 **foo**$_3$ a4 **foo**$_4$ a5 **foo**$_5$ a6

After that, the parser applies the following rules to these functions:

1. Only the first function **foo**$_1$ is truly evaluated.
2. The 2nd , 3rd and other successive functions are never evaluated. They simply assign their right arguments to the corresponding argument of the first function
3. The first function has the lowest priority.

The rule in point 2 means that for i > 1 `ai` **`foo`**$_i$ `a(i+1)` $\Rightarrow$ `a(i+1)` assigned the (i+1)th argument of **`foo`**$_1$

`foo`$_i$ `is` the (i+1)th argument of **`foo`**$_1$

Example. Parse and evaluate the hypergeometric function with the following parameters

```
hypgeom(0.25, -2.5, 3, 7)
```

First of all, the parser translates the above formula into the plain form

```
0.25 hypgeom -2.5 hypgeom 3 hypgeom 7
```

building the following structured table.

N	Fun	ArgTop	Arg1 Value	Arg2 Value	ArgOf	ArgIdx	Value	PriIdx
1	hypgeom	4	0.25	-2.5	0	0	0.75666172	3
2	hypgeom	2	-2.5	3	1	3	3	2
3	hypgeom	2	3	7	1	4	7	1

Each row represents a function.
The PriIdx (priority index) shows that the evaluation starts with the 3rd row, then the 2nd one and, at the last, the 1st one.
The 3rd function assigns the value 7 to the argument 4 of the 1st function
The 2nd function assigns the value 3 to the argument 3 of the 1st function
The 1st function will be evaluated with the following arguments: hypgeom(0.25, -2.5, 3, 7) = 0.756…

Note. The assignment operation is performed in the Parser routine by the internal function *@Right*. So, all "*hypgeom*" functions below the first one will be substituted with this *@Right* function.

Of course the function arguments can have more complicated subexpressions containing other functions as well.

Example. Parse and evaluate the normal density function having: $x = 2$, $\mu = 2/3$, $\sigma = (2/3)^2$

```
Dnorm(2, 2/3, (2/3)^2)
```

The structured table will be

N	Fun	ArgTop	Arg1 Value	Arg2 Value	ArgOf	ArgIdx	Value	PriIdx
1	Dnorm	3	2	0.666666667	0	1	0.009971659	4
2	/	2	2	3	1	2	0.666666667	5
3	@Right	2	3	0.444444444	1	3	0.444444444	2
4	/	2	2	3	5	1	0.666666667	3
5	^	2	0.666666667	2	3	2	0.444444444	1

As we can see, the result of the expression `(2/3)^2` (row 5) is assigned to the 2nd argument of the *@Right* function (row 3), which passes this value to the 3rd argument of the function *Dnorm* (row 1). This row will be the last function to evaluate, as shown in the PriIdx column.

For more examples see the Appendix

Physical numbers.

From version 2.1, the parser recognizes and computes physical units of measure like

```
3.2 kg + 123 g        10/144 MHz          20.3km+8km          (0.1uF)*(6.8 kohm)
```

Note: parentheses in the last example are not necessary. They only improve the reading

First of all, we must fix some rules.

Rules and definitions:

Physical expressions are formed by two parts: a numerical and a literal part. Both parts must be inserted. The literal part can be divided into two parts: a multiplying factor and a unit of measure symbol. The multiplying factor can be omitted whereas the unit of measure must always be inserted. There may be a blank between the numeric and literal part.
Numbers without a unit of measure (UM) are called "pure", which is the case of math numbers.
For example:

"`20km`" can be transformed into a numerical value in this simple way.

20km = 20 * k * m =20 * 1000 * 1 = 20000 (definition of m=1 in the MKS system)

These are not correct physical expressions

30*Km ==> "Km" is a variable, not the unit of measure

3*V ==> "V" is a variable, not the unit of measure

MHz ==> “MHz”, without a number before, is a variable, not the unit of measure

Minus Sign.

The parser recognizes the minus sign of constants, variables and functions.

```
x^-n       10^-2      x*-sin(a)      -5*-2
```

The parser applies the following rule.
Rule: if the minus symbol is preceded by an operator, then it is treated as a sign; else it is treated as the subtraction operator.

This feature simplifies expressions writing. Without this rule, the above expression should be written with parentheses, that is:

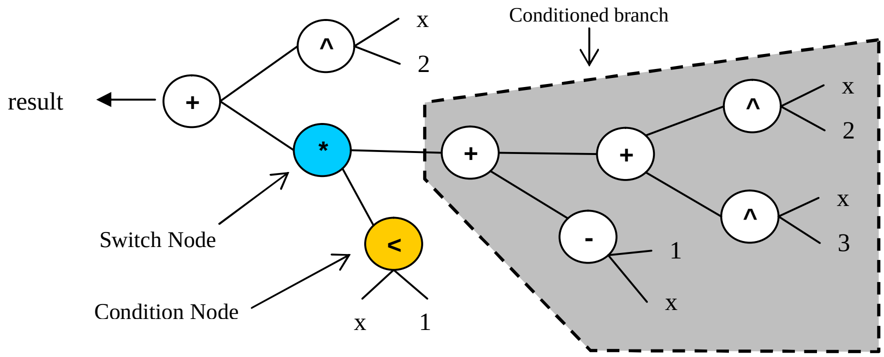
```
x^(-n)     10^(-2)    x*(-sin(a))    -5*(-2)
```

Conditioned Branch.

From version 3.4 the parser can decide when calculating a part of a formula expression subjected on a logical condition. Let's see how it works.
Assume the following formula expression: `x^2+(x<1)*(1-x+x^2+x^3)`
It's clear that if the logical condition (x<1) is 0 (false), then it is useless to compute the dependent subexpression `(1-x+x^2+x^3)`, because the result of the second terms will be always 0. In many cases, it is also necessary to switch off the evaluation of the dependent subexpression to avoid domain evaluation errors, like for example in the following expression: `(x<=0)*x+(x>0)*ln(x)`

In that case, adopting the following name convention, we have:

`(x<1)`	Condition expression	It must always be evaluated with the highest priority
`(1-x+x^2+x^3)`	Conditioned branch	It must be evaluated if, and only if, the cond. expression is true
`*`	Switch Node	
`<`	Condition node	

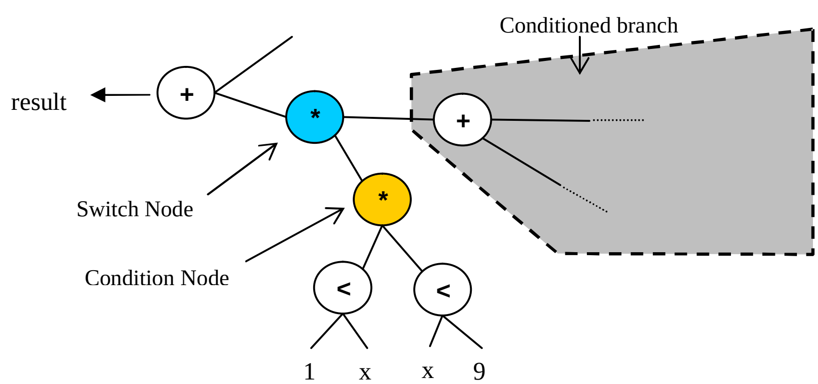


We see that the *switch node* is the product operation "*" . Any <u>child</u> node of the switch node, except the conditon node one, constitutes the set called *conditional branch*. Any node of the conditioned branch set will be calculated only if the condition node gives 1 (true) as result. We have to point out that the switch node must always be a product "*", <u>father</u> of the condition node, which is usually a logical operator <, >, = ; but can also be another operator. This happens for multiple logical conditions. Let's see.

Example …`+(1<x<9)*(1-x+x^2+x^3)+`…

that is formally equivalent to …`+(1<x)*(x<9)*(1-x+x^2+x^3)+`…

And can be represented by the following graph-tree



As we can see, in the case of multiple logic conditions, the condition node is another product “*” operator, the result of two logic operators. But the *Father-Child* rule is still true: each switch node is the father of its condition node; each condition node is the child of its corresponding switch node.
The nodes of the conditioned branch can be visited by the binary-tree descendent algorithm: starting from the switch node the parser searches for all the child paths, except the condition node one. Each node of the conditioned branch will be linked to its corresponding conditioned node. This relation is built by adding an index column, called “Cond” in the ET table

For example the ET table of the above formula `x^2+(x<1)*(1-x+x^2+x^3)` will be

ID	Fun	…	PriLvl	PriIdx	Cond
1	^	…	4	3	0
2	+	…	2	7	0
3	<	…	111	9	0
4	*	…	3	5	0
5	-	…	12	6	3
6	+	…	12	8	3
7	^	…	14	1	3
8	+	…	12	4	3
9	^	…	14	2	3

As we can see, nodes from 5 to 9 are conditioned by node 3. It means that the evaluation of the functions 5, 6, …9 will be performed only if the result of the function 3 is 1 (true). On the contrary, the evaluation will be skipped.
Of course the evaluation of the node 3 must be performed before any other functions. Within this scope, the parser assigns the highest priority (PriLvl) to the functions related to the condition node

If there exist two or more logical conditions in a formula, the algorithm is repeated for any logical condition. This is particularly important for the definition of piecewise functions, such as:

$$f(x) = \begin{cases} x^2 & x \le 0 \\ \ln(x+1) & 0 < x \le 1 \\ \sqrt{x - \ln(2)} & x > 1 \end{cases}$$

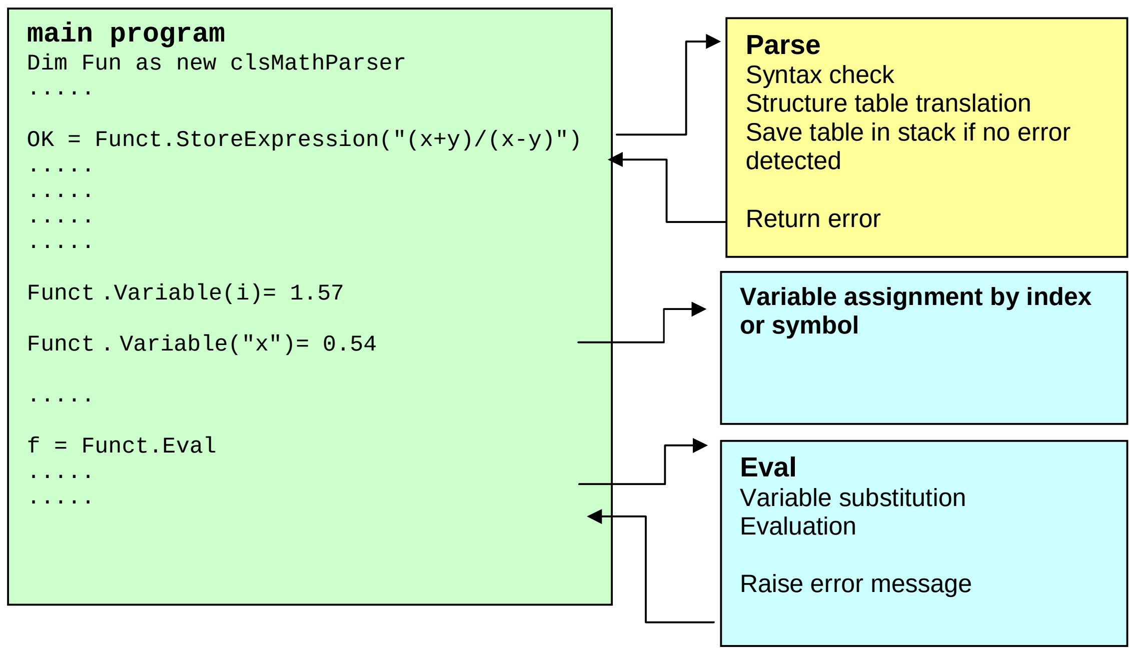
The above function can be translated into the following string

```
“(x<=0)* x^2 + (0<x<=1)* Ln(x+1) + (x>1)* Sqr(x-Ln(2))”
```

The corresponding table ET, evaluated for x = −2 , is shown in the appendix

How to improve speed in loop computation

When we have to calculate the same expression `(a+b)*(a-b)` with different numerical values we could only repeat the second step of the evaluation because the structure of the expression is unchanged. Because the first parsing step takes a relevant part of the total time, we could save a lot of time avoiding to repeat the parse.



Note that:
The **Eval** subroutine is about 50 times faster than **Parse.**
Indexed assignment is about 2 times faster than **symbolic** one.

Error messages

ErrDescription Property

The StoreExpression method returns the status of parsing. It returns TRUE or FALSE, and the property ErrDescription contains the error message. An empty string means no error. If an error happens, the formula is rejected and the property contains one of the messages below.
In addition, the methods *Eval* and *Eval1* set the ErrMessage property.

<table>
<tr><th>Error Intercepted</th><th>Description</th><th>ID</th></tr>
<tr><td>Function <name > unknown at pos: i^{th}</td><td>String detected at position i^{th} is not recognized. example “x + foo(x)”</td><td>6</td></tr>
<tr><td>Syntax error at pos: i^{th}</td><td>Syntax error at position i^{th} . “a(b+c)”,</td><td>5</td></tr>
<tr><td>Too many closing brackets at pos: i^{th}</td><td>Parentheses mismatch: “sqr(a+b+c))”</td><td>7</td></tr>
<tr><td>missing argument</td><td>Missing argument for operator or function.”3+” , “sin()”</td><td>8</td></tr>
<tr><td>variable <name> not assigned</td><td>Variable “name” has never been assigned. This error rises only if the property AssignExplicit = True</td><td>18</td></tr>
<tr><td>too many variables</td><td>Symbolic variables exceed the set limit. Internal constant HiVT =100 sets the limit</td><td>1</td></tr>
<tr><td>Too many arguments at pos: i^{th}</td><td>Too many arguments passed to the function</td><td>9</td></tr>
<tr><td>Not enough closing brackets</td><td>Parentheses mismatch: “1-exp(-(x^2+y^2)”</td><td>11</td></tr>
<tr><td>abs symbols |.| mismatch</td><td>Pipe mismatch: “|x+2|-|x-1”</td><td>4</td></tr>
<tr><td>Evaluation error <5 / 0> at pos: 6
Evaluation error <asin(-2)> at pos: 10
Evaluation error <log(-3)> at pos: 6</td><td>The only error returned by the Eval method. It is caused by any mathematical error: 1/0, log(0), etc.</td><td>14,15,
16, 17</td></tr>
<tr><td>constant unknown: <name ></td><td>a constant not recognized by the parser</td><td>19</td></tr>
<tr><td>Wrong DMS format</td><td>an angle expressed in a wrong DMS format (eg: 2° 34' 67")</td><td>21</td></tr>
<tr><td>Too many operations</td><td>The expression string contains more than 200 operations/functions</td><td>20</td></tr>
<tr><td>Function <id_function> missing?</td><td>Help for developers. Returned by the Eval_ internal routine. If we see this message we probably have forgotten the relative case statement in eval subroutine!</td><td>13</td></tr>
<tr><td>Variable not found</td><td>a variable passed to the parser not found</td><td>2</td></tr>
</table>

In order to facilitate the modification and/or translation, all messages are collected into an internal table
See appendix "Error Message Table" for details

Error Raise

Methods Eval, Eval1, EvalMulti and also raise an exception when any evaluation error occurs.
This is useful to activate an error-handling routine. The global property Err.Description contains the same error message as the ErrDescription property

Computation Time Test

The clsMathParser class has been tested with several formulas in different environments. The response time depends on the number and type of operations performed and, of course, by the computer speed. The following table shows synthetically the performance obtained. (Excel 2000, Pentium 3, 1.2 Ghz,)

Results

Clock => 1200 MHz	18/11/2002			
Expression	**operations**	**time (ms)**	**time/op. (ms)**	**Eval. / sec.**
average	**4.1**	**0.009**	**2.6**	**187,820**
1+(2-5)*3+8/(5+3)^2	7	0.01	1.43	100,000
sqr(2)	1	0.003	3.0	333,333
sqr(x)	1	0.004	4.0	250,000
sqr(4^2+3^2)	4	0.007	1.75	142,857
x^2+3*x+1	4	0.009	2.25	111,111
x^3-x^2+3*x+1	6	0.014	2.33	71,429
x^4-3*x^3+2*x^2-9*x+10	10	0.022	2.2	45,455
(x+1)/(x^2+1)+4/(x^2-1)	8	0.018	2.25	55,249
(1+(2-5)*3+8/(5+3)^2)/sqr(4^2+3^2)	12	0.017	1.42	58,824
sin(1)	1	0.003	3.0	333,333
asin(0.5)	1	0.004	4.0	250,000
fact(10)	1	0.005	5.0	200,000
sin(pi/2)	2	0.004	2.0	250,000
x^n/n!	3	0.012	4.0	83,333
1-exp(-(x^2+y^2))	5	0.016	3.2	62,500
20.3Km+8Km	2	0.002	1.0	500,000
0.1uF*6.8KOhm	2	0.003	1.5	333,333
(1.434E3+1000)*2/3.235E-5	3	0.005	1.67	200,000

As we can see, the computation time depends strongly on the number of symbolic variables contained in an expression. Numeric expressions are the fastest.

Time Test Code

The table above was obtained using the following simple straight code.

```
Sub TestCalc()
  Dim OK     As Boolean
  Dim retval As Double
  Dim t0     As Double
  Dim i      As Long
  Dim Loops  As Long
  Dim txtFormula As String
  Dim Fun As New clsMathParser

  '----- Set test values -----------------
  txtFormula = "x^3-x^2+10*x-3"
  Loops = 100000
  x = 3
  '--------------------------------------
  OK = Fun.StoreExpression(txtFormula)
  '
  If Not OK Then GoTo Error_Handler

  Debug.Print "------- Evaluation Test begin ---"; String(40, "-")

  For i = 1 To Fun.VarTop
      Fun.Variable(i) = 2.5 * i    '(any values)
  Next

  t0 = Timer
  If Fun.VarTop = 1 Then
```

```
    For i = 1 To Loops
        retval = Fun.Eval1(x)
        If Err Then GoTo Error_Handler
    Next
  Else
    For i = 1 To Loops
        retval = Fun.Eval
        If Err Then GoTo Error_Handler
    Next
  End If
  t0 = Timer - t0
  Debug.Print "Formula= "; txtFormula
  Debug.Print "Elapsed time="; t0
  Debug.Print "Loops="; Loops
  Debug.Print "Eval/sec="; Int(Loops / t0)
  Set Fun = Nothing
Exit Sub
Error_Handler:
Debug.Print Err.Description
End Sub
```

For each formula, the output will be:

```
------- Evaluation Test begin ------------------------------------------
Formula= x^3-x^2+10*x-3
Elapsed time= 1.3719999999986
Loops= 100000
Eval/sec= 72886
Result= 31.375
```

Conclusions

clsMathParser can be used to calculate any formulas (expressions) given at runtime, for example to plot and tabulate functions, to make numerical computations and much more. The clean and very simple interface is suited for engineers, mathematicians and physicians and for those people that are not professional programmers. Technical aspects, such as parsing, internal formula translation, memory sizing, etc. are resolved by an internal routine and hidden to the end programmer. This also helps to keep the final code cleaner.

This parser has several advantages:

- good speed for loop calculation
- good syntax check
- pretty staightforward and easy interface
- robust and good portability.
- good adaptability.
- simple, complete documentation
- easy debugging
- original evaluation method.

And - last but not least - freeware and open-source

This parser has obviously some disadvantages:

- speed is slower than compiled math parsers
- parsing method is not as flexible as other classical methods

This MathParser was originally developed as an internal sub-module for add-in packages in order to fast compute formulas created at runtime. Typical applications are: math integration, differential equation solving, math series evaluation, etc.

Credits

clsMathParser was ideated by

Leonardo Volpi

and developed thanks to the collaboration of

Lieven Dossche, Michael Ruder, Thomas Zeutschler, Arnaud De Grammont.

Many thanks also for their help in debugging, improving and setting up to:

Rodrigo Farinha, Shaun Walker, Iván Vega Rivera, Javie Martin Montalban, Simon de Pressinger,
Jakub Zalewski, Sebastián Naccas, RC Brewer, PJ Weng, Mariano Felice, Ricardo Martínez Camacho.
Berend Engelbrecht, André Hendriks, Michael Richter, Mirko Sartori

Special thanks for the documentation revision to

Mariano Felice

High precision Special functions have been translated into VB from

LIBRARY FOR COMPUTATION of SPECIAL FUNCTIONS in FORTRAN-77 (*) '
by Shanjie Zhang and Jianming Jin.

(*) All the subroutines of the library are subject to copyright. However, authors give kindly permission to incorporate any of these routines into other programs providing that the copyright is acknowledged.

Licence

clsMathParser is freeware open software. We are happy if you use and promote it. You are granted a free license to use the enclosed software and any associated documentation for personal or commercial purposes, except to sell the original. If you wish to incorporate or modify parts of clsMathParser please give them a different name to avoid confusion. Despite the effort that went into building, there's no warranty, that it is free of bugs. You are allowed to use it at your own risk. Even though it is free, this software and its documentation remain proprietary products. It will be correct (and fine) if you put a reference about the authors in your documentation.

History Review

MATH PARSER For Visual Basic, VBA

ClsMathParser.cls - Ver. 4.2.0 Update of Oct. 2006

The parser can manage functions with multi-variable number of arguments.
(Thanks to Mirko Sartori)
changed: max(x1,x2,...)   max of n numbers
changed: min(x1,x2,...)   min of n numbers

Added also the following functions with multi-variable arguments :
Added: lcm(x1,x2,...)   least common multiple
Added: gcd(x1,x2,...)   greatest common divisor
Added: mean(x1,x2,...)   arithmetic mean
Added: meanq(x1,x2,...)   quadratic mean
Added: meang(x1,x2,...)   geometric mean
Added: var(x1,x2,...)   variance
Added: varp(x1,x2,....)   variance pop.
Added: stdev(x1,x2,...)   standard deviation
Added: stdevp(x1,x2,..)   standard deviation pop.
Added: sum(x1,x2,...)   sum.

Added: step(x,a) Haveside's step function
Added: International   setting for decimal separators: "." ","

ClsMathParser.cls - Ver. 4.1.2 Update of March. 2006

Fix 04.05.2005. Bug in function for checking international system setting format
(Thanks to André Hendriks and Ricardo Martínez C.)
Fix 23.03.2006. Modify incomplete gamma function

ClsMathParser.cls - Ver. 4.1.0 Update of March. 2005

The following new time functions has been added
(thank to Berend Engelbrecht)

Intrinsic constants: Date# Time# Now#
Date to number: Year, Month, Day, Hour, Minute, Second,
Encoding functions: DateSerial, TimeSerial

The following new properties has been added
OpUnitConv   Option: Enable/disable unit conversion
OpDMSConv   Option: Enable/disable DMS angle conversion
ErrorID   Error message number
ErrorPos   Error position

A new Error Message Table has been implemented for making easier the error message translation

Bug 2004.12.16. For same variable name conv. DMS routine returns Wrong DMS format
(Thanks to André Hendriks)
Bug 2004.12.21. Error when using both uppercase/lowercase variable name (e.g. Alpha vs. alpha or Z versus z).
(Thanks to André Hendriks and Michael Richter)
Bug 2004.10.16. When the expression overcomes 100 operations no error was detected
(Thanks to Michael Ruder)

ClsMathParser.cls - Ver. 4.0.0 Update of Sept. 2004

This release can manage multi-variables functions
Added: Psi(x)   Alias for digamma function
Added: DNorm(x,m,d)   Normal density function
Added: CNorm(x,m,d)   Normal cumulative function
Added: DPoisson(x,k)   Poisson density function k = 1, 2, 3 ...

Added: CPoisson(x,k)	Poisson cumulative function k = 1, 2,3 ...
Added: DBinom(k,n,x)	Binomial density for k successes for n trials k < n
Added: CBinom(k,n,x)	Binomial cumulative for k successes for n trials k < n
added: Ein(x, n)	Nth Exponential integral
added: Si(x)	Sine integral
added: Ci(x)	Cosine integral
added: FresnelS(x)	Fresnel's sine integral
added: FresnelS(x)	Fresnel's cosine integral
added: J0(x)	Bessel's function of 1st kind
added: Y0(x)	Bessel's function of 2st kind
added: I0(x)	Bessel's function of 1st kind, modified
added: K0(x)	Bessel's function of 2st kind, modified
added: BesselJ(x,n)	Bessel's function of 1st kind, nth order
added: BesselY(x,n)	Bessel's function of 2st kind, nth order
added: BesselI(x,n)	Bessel's function of 1st kind, nth order, modified
added: BesselK(x,n)	Bessel's function of 2st kind, nth order, modified
added: HypGeom(x,a,b,c)	Hypergeometric function
added: GammaI(a,x)	Gamma incomplete
added: BetaI(x,a,b)	Beta incomplete
added: PolyCh(x,n)	Chebycev's polynomials
added: PolyLe(x,n)	Legendre's polynomials
added: PolyLa(x,n)	Laguerre's polynomials
added: PolyHe(x,n)	Hermite's polynomials
added: AiryA(x)	Airy function Ai(x)
added: AiryB(x)	Airy function Bi(x)
added: Elli1(x)	Elliptic integral of 1st kind
added: Elli2(x)	Elliptic integral of 2st kind
added: Clip(x,a,b)	Clipping function
added: WTRI(t,p)	Triangular wave
added: WSQR(t,p)	Square wave
added: WRECT(t,p,d)	Rectangular wave
added: WTRAPEZ(t,p,d)	Trapez. wave
added: WSAW(t,p)	Saw wave
added: WRAISE(t,p)	Rampa wave
added: WLIN(t,p,d)	Linear wave
added: WPULSE(t,p,d)	Rectangular pulse wave
added: WSTEPS(t,p,n)	Steps wave
added: WEXP(t,p,a)	Exponential pulse wave
added: WEXPB(t,p,a)	Exponential bipolar pulse wave
added: WPULSEF(t,p,a)	Filtered pulse wave
added: WRING(t,p,a,fm)	Ringing wave
added: WPARAB(t,p)	Parabolic pulse wave
added: WRIPPLE(t,p,a)	Ripple wave
added: WAM(t,fo,fm,m)	Amplitude modulation
added: WFM(t,fo,fm,m)	Frequecy modulation

New Property AssignExplicit forces explicit variable assignment
New Function argument separator is comma ","
Bug 2004.06.24 : incorrect result for absolute |3|2 input (thank to PJ Weng)
Bug 2004.08.13 : incorrect result for "x = = y" and "2<<3" input (thanks to Ricardo Martínez C.)
Document PDF revision “

ClsMathParser.cls - Ver. 3.4.5 Update of July 2004

The following structural changes have been implemented (thanks to Ricardo Martínez C.)
GoSub statement has been removed.
Lower bound of ALL arrays was changed to 0.
ByVal and ByRef are always specified.
Variable types are always specified.
Variant type has been removed.

Bug 2004.06.05 : Error raising for 0^0 indeterminate form. (thanks to Mariano Felice)

Bug 2004.06.24 : incorrect result for wrong parenthesis input (thanks to PJ Weng)
Bug 2004.07.10 : raising error for wrong decimal point 3..456 "
Bug 2004.07.30 : incorrect sign for bi-functions es: -max(4;6) "
Bug 2004.06.26 : Avogadro constant and neutron rest mass missing (thanks to Ricardo Martínez C.)
Bug 2004.07.23 : incorrect response of ErrorDescription property "
Bug 2004.07.19 : Variable case-sensitivity "
Bug 2004.07.23 : ErrorDescription property "

ClsMathParser.cls - Ver. 3.4.4 Update of June 2004

Bug 2004.31.5 : incorrect angle sign in DMS format (thank to PJ Weng)
Improved DMS angle format detection
Error raising for non-integer arguments of mcm and mcd

ClsMathParser.cls - Ver. 3.4.3 Update of May 2004

- dec(x) function improved
- LogN(x;n) n-base logarithm

ClsMathParser.cls - Ver. 3.4.2 Update of April. 2004

The following bug fixed (thank to PJ Weng)
Bug 2004.03.30 : compilation error in internal function convEGU() for VBA Chinese edition
Bug 2004.04.02 : incorrect detection for "++" "--"
Bug 2004.04.16 : Mod(a;b) function: wrong result for negative numbers

- documentation PDF revised (thank to PJ Weng)
- new function added: atan(x) beside of atn(x)
- new function added: perm(n, k) permutations of n objects in k classes

ClsMathParser.cls - Ver. 3.4.1 Update of March. 2004

Bug 2004.2.29 : incorrect error detection happens when it is used a minus sign outside of absolute "|.|" symbols. Es: -|x-1| raised an error. (Thank to Rodrigo Farinha)

ClsMathParser.cls - Ver. 3.4 Update of Feb. 2004

New property VarSymb for direct variable value assignment
This means that you can now assign a variable value by passing its symbolic name:

```
Object.VarSymb("x")= 2.5465
Object.VarSymb("time")= 0.5465
etc.
```

Of course the old index assignment is still supported.
(This property was added thank to R.C. Brewer and Sebastián Naccas)

Improved algorithm for variable sign detection. (thank to Joe Kycek)
Now it is possible to evaluate a string like the following without parentheses:

x^-n	the same as	x^(-n)
x*-sin(a)	the same as	x*(-sin(a))
-5*-2	the same as	-5*(-2)
-5^2	the same as	(-5)^2
etc.		

Interval definition (a<x)*(x<b) is now also allowed in compact form (a<x<b).

Conditioned branch evaluation by logical expressions = < > <= => <>.
If a logic expression is 0 (false), then the dependent sub-expression is not evaluated.
This feature is useful for piecewise function definitions (thanks to Rodrigo Farinha)

Example: f(x):= (x<=0)*x+(0<x<=1)*log(x)+(x>1)*sqr(x^3-x)

ClsMathParser.cls - Ver. 3.3.1 Update of Oct. 2003

Bug 2003.10.9.1 : function acosh(x) returned wrong values
Bug 2003.10.9.2 : fact(x) error raise for x<0 ; it returned wrong result 1
Bug 2003.10.9.3 : "Missing argument" error is raised for string "3+", "sin(), etc."
Bug 2003.10.16.2 : wrong result for nested bi-var functions

Foxes Team

Algebraic extension of cbr(x) and root(x, n) for x<0
(Thanks to Rodrigo Farinha)

Math constant supported are now:
pi = pi-greek, e# = Euler-Napier's constant, eu# = Euler-Mascheroni constant
Physical constants are now supported:
Planck constant, Boltzmann constant, Elementary charge, Avogadro number
Speed of light, Permeability of vacuum, Permittivity of vacuum, Electron rest mass
Proton rest mass, Gas constant, Gravitational constant, Acceleration due to gravity

ClsMathParser.cls - Ver. 3.2.3 Update of May. 2003

Bug 2003.30.5.1 : wrong result for minus sign in front of function. Es: -sqr(2), -log(2)
Bug 2003.31.5.1 : Private function zeta_ declaration
Bug 2003.01.6.1 : fix of fix of Bug 2003.02.14.1 (:-)

ClsMathParser.cls - Ver. 3.2.2 Update of March. 2003

Bug 2003.02.14.1 : wrong result for expressions like: 10^-x , 3^-2, etc.
Bug 2003.02.14.2 : incorrect detection of unit of measure "Hz"
example:50 Hz produces an error: 'variable name must start with a letter'
(Thanks to Javier M. Montalban)

Bug 2003.02.26.1 : the parser will not recognize decimal numbers starting with "." instead of "0." when the system setting is comma for decimal point (european setting). Example ".54" instead of "0.54".
Bug 2003.02.26.2 : incorrect detection of variable name "abc2xyz"
(Thanks to Michael Ruder)

ClsMathParser.cls - Ver. 3.2.1 Update of Feb. 2003

The following functions are added:
csc cosecant
sec secant
cot cotangent
acsc inverse cosecant
asec inverse secant
acot inverse cotangent
csch hyperbolic cosecant
sech hyperbolic secant
coth hyperbolic cotangent
acsch inverse hyperbolic cosecant
asech inverse hyperbolic secant
acoth inverse hyperbolic cotangent
cbr cube root
root n-th root
dec decimal part
rad convert radiant into current angle unit
deg convert degree (sess.) into current angle unit
grad convert degree (cent.) into current angle unit
round decimal round
LCM least common multiple
GCD greatest common divisor
% percentage (was the module operator in older version)
EU Euler's gamma constant eu=0.577...
nand logic nand
nor logic nor
nxor logic xnor

Enhancement of domain error reports
Property AngleUnit added to set the angle unit of measure: RAD, DEG, GRAD
Symbol "_" into variable name is now accepted (programming style variables like var_1, var_2, etc.)
degree in ddmmss format now supported (44°33'53")

Bugs found and fixed by Michael Ruder

- error in parsing the exponential factor. The problem occurs if you add or subtract two variables where the first variable ends with the letter
"E"... This lets the parser believe, that ...E+ means there will be an exponential factor.

- error in parsing physical units.
With the factorial character missing, it resulted in a rejection of a number like "5s"
(which then leads to the error "variable name must start with a letter")

- wrong calculation of XOR-Function

(Thanks Michael ;-)

ClsMathParser.cls - Ver. 3.1 Update of 2-Jan. 2003

substituted all implicit declarations for compatibility with other languages as AppForge.

ClsMathParser.cls - Ver. 3.0 Update of 9-Dic. 2002

bug: error in cos(pi) thanks to Arnaud de Grammont

ClsMathParser.cls - Ver. 3.2 Update of Jan. 2003

The following functions are added:
- csc cosecant
- sec secant
- cot cotangent
- acsc inverse cosecant
- asec inverse secant
- acot inverse cotangent
- csch hyperbolic cosecant
- sech hyperbolic secant
- coth hyperbolic cotangent
- acsch inverse hyperbolic cosecant
- asech inverse hyperbolic secant
- acoth inverse hyperbolic cotangent
- cbr cube root
- root n-th root
- dec decimal part
- rad convert into radiant
- deg convert into degree
- round decimal round
- lcm least common multiple
- gcd greatest common divisor

ClsMathParser.cls - Ver. 3.0 Update of Dic. 2002

MathParser was encapsulated in a nice class by Lieven Dossche.
In addition, the following new features are added:
Logical operators < > =
Logic functions AND, NOR, NOT
Error detection was improved

(many thanks to Lieven)

MathParser23.bas - Ver. 2.3 Update of Nov. 2002

Computation efficiency is improved thank to Thomas Zeutschler
This version is about 200% faster than the previous one.
- Parse() subroutine added in substitution of ParseExprOnly()
- Trap computation errors added (1/0, log(0), etc..)
- Exponential number detection added
- absolute symbol |.| detection added
- "ohm", "Rad", "RAD" units detection added

(many thanks to Thomas)

MathParser21MR.bas - Ver. 2.1 Update of June 2002

Born from the collaboration of Michael Ruder (ruder@gmx.de), this version includes several new features:
- Detection and computation of physical numbers with units of measure like Km Kg, V, MHz...
- Bug in parenthesis check. Test on LP negative to avoid wrongly set brackets
- New *ParseExprOnly* routine to get the variable list and to check syntax
- New operator symbols: \ (integer division), % (modulus division)
- General speed improved by about 20%

(many thanks to Michael)

MathParser21.bas - Ver. 2.1 Update of May 2002

This version includes the following upgrade from the collaboration of Iván Vega Rivera:
- Detection and computation of bivariable functions
- New functions *Comb(n,k) Max(a,b) Min(a,b) Mcm(a,b) Mcd(a,b) Mod(a,b)* added:

(greetings to Iván Vega Rivera)

MathParser11.bas - Ver. 1.1 Update of April 2002

This version includes the following upgrade:
- Dynamical manage of parsing - The EvalExpr() function stores up to 9 string formulas in a special stack in order to avoid the parsing process for the last 9 formulas passed. Useful for computing vector of functions, like gradient, hessian, etc.
- Abs() function operator is added
- General speed is increased by about 25%

MathParser10.bas - Ver. 1.0 - Update of May 2000

First release

This version 1.0 handles conventional math operators and a few main built-in functions, as shown on the following list:

*+ - * / ^ ! atn cos sin exp fix int ln log rnd sgn sqr tan acos asin cosh sinh tanh acosh asinh atanh fact pi*

APPENDIX

Source code examples in VB

To explain the use of this routine, a few simple examples in VB are shown.
Time performance has been obtained with Pentium 3, 1.2GHz, and Excel 2000

Example 1:

This example shows how to evaluate a mathematical polynomial p(x) = `x^3-x^2+3*x+6` for 1000 values of the variable "x" between x_min = -2 , x_max = +2, with step of $\Delta x = 0.005$

```
Sub Poly_Sample()
Dim x(1 To 1000) As Double, y(1 To 1000) As Double, OK As Boolean
Dim txtFormula As String
Dim Fun As New clsMathParser
'------------------------------------------------------------------------
txtFormula = "x^3-x^2+3*x+6"   'f(x) to sample.
'-----------------------------------------------------------------------
'Define expression, perform syntax check and get its handle

OK = Fun.StoreExpression(txtFormula)
If Not OK Then GoTo Error_Handler

'load input values vector.
For i = 1 To 1000
    x(i) = -2 + 0.005 * (i - 1)
Next

t0 = Timer
For i = 1 To 1000
    y(i) = Fun.Eval1(x(i))
    If Err Then GoTo Error_Handler
Next

Debug.Print Timer - t0  'about 0.015  ms for a CPU with 1200 Mhz

Exit Sub
Error_Handler:
    Debug.Print Fun.ErrorDescription
End Sub
```

We note that the function evaluation – the focal point of this task - is performed in 5 principal statements:

1) Function declaration, storage, and parsing
2) Syntax error check
3) Load variable value
4) Evaluate (eval)
5) Activate error trap for checking domain errors

Just clean and straightforward. No other statement is necessary. This takes an overall advantage in complicated math routines, when the main focus must be concentrated on the mathematical algorithm, without any other technical distraction and extra subroutines calls. Note also that declaration is needed only once at a time.

Of course that computation speed is also important and must be put in evidence.
For the example above, with a Pentium III° at 1.2 GHz, we have got 1000 points of the cubic polynomial in less than 15 ms (1.4E-2), which is very good performance for this kind of parser (70.000 points / sec)

Note also that the variable name it is not important; the example also works fine with other strings, such as: "`t^3-t^2+3*t+6`" , "`a^3-a^2+3*a+6`", etc.
The parser, simply substitutes the first variables found with the passed value.

Example2

This example shows how to build an Excel user function for evaluating math expression.

```
Function Compute(Formula As String, ParamArray Var())
Dim retval As Double, ParMax As Long
Dim Fun As New clsMathParser, OK As Boolean
On Error GoTo Error_Handler
'---------------------------------
OK = Fun.StoreExpression(Formula)
'---------------------------------
If Not OK Then Err.Raise 1001, , Fun.ErrorDescription
ParMax = UBound(Var) + 1 'number of parameters
If ParMax < Fun.VarTop Then Err.Raise 1002, , "missing parameter"
ReDim Values(1 To ParMax)
'load parameters values
For i = 1 To ParMax
    Fun.Variable(i) = Var(i - 1)
Next
Compute = Fun.Eval()
If Err <> 0 Then GoTo Error_Handler
Exit Function
Error_Handler:
Compute = Err.Description
End Function
```

This function can be used in Excel to calculate formulas passing only the string expression and parameter values (if present). Note how compact and clean the code is.

	A	B	C	D	E	F
1	x	y	z	f(x,y,z)	result	
2	5	-2	-1	3x+3y-z	10	=Compute(D3;A3;B3;C3)
3	3.5			x^4-x^2+10x^2-26x-34	135.3125	=Compute(D4;A4)
4	0.5			sin(2*pi*x)+cos(2*pi*x)	-1	=Compute(D5;A5)

You can also insert the formula string directly into the Compute() function.
In this case, do not forget the quote “”.

Example 3

This example computes Mc Lauren's series up to 16° order for exp(x) with x=0.5
(exp(0.5) ≅ 1.64872127070013)

```
Sub McLaurin_Serie()
Dim txtFormula As String
Dim n As Long, N_MAX As Integer, y As Double
Dim Fun As New clsMathParser
'-------------------------------------------------------------------------
txtFormula = "x^n / n!"   'Expression to evaluate. Has two variable
x0 = 0.5    'set value of Taylor's series expansion
N_MAX = 16  'set max series expansion
'------------------------------------------------------------------------
'Define expression, perform syntax check and get its handle
OK = Fun.StoreExpression(txtFormula)
If Not OK Then GoTo Error_Handler

'begin formula evaluation -------------------------
Fun.Variable(1) = x0 'load value x
For n = 0 To N_MAX
    Fun.Variable(2) = n 'increments the n variables
    If Err Then GoTo Error_Handler
    y = y + Fun.Eval    'accumulates partial i-term
Next
Debug.Print y
Exit Sub
Error_Handler:
Debug.Print Fun.ErrorDescription
End Sub
```

```
1.64872127070013
```

Also note the clean code in this case.

Example 4

This example shows how to capture all the variables in a formula in the right sequence (sequence is from left to right).

```
Dim Expr As String
Dim OK As Boolean
Dim Fun As New clsMathParser

Expr = "(a-b)*(a-b)+30*time/y"
-----------------------------------------------------------------------
  'Define expression, perform syntax check and detect all variables
   OK = Fun.StoreExpression(Expr)
-----------------------------------------------------------------------
If Not OK Then
    Debug.Print Fun.ErrorDescription  'syntax error detected
Else
    For i = 1 To Fun.VarTop
        Debug.Print Fun.VarName(i), Str(i) + "° variable"
    Next i
End If
```

Output will be:

```
a              1° variable
b              2° variable
time           3° variable
y              4° variable
```

Example 5

This is an example to show how to build an Excel function to calculate a definite integral.
The following is complete code for integration with Romberg method

```
'===============================================================================
' Integral - Romberg Integration of f(x).
' ---------------------------------------
' Funct  = function f(x) to integrate
' a      = lower intgration limit
' b      = upper intgration limit
' Rank   = (optional default=16) Sets the max samples = 2^Rank
' ErrMax = (optional default=10^-15) Sets the max error allowed
' Algorithm exits when one of the two above limits is reached
'================================================================================
Function Integral(Funct$, a, b, Optional Rank, Optional ErrMax)
Dim R() As Double, y() As Double, ErrLoop#, u(2)
Dim i&, Nodes&, n%
Dim Fun As New clsMathParser
'-----------------------------------------------
If VarType(Funct) <> vbString Then Err.Raise 1001, , "string missing"
If IsMissing(Rank) Then Rank = 16
If IsMissing(ErrMax) Then ErrMax = 10 ^ -15
'
'Define expression, perform syntax check and get its handle
OK = Fun.StoreExpression(Funct)
If Not OK Then Err.Raise 1001, , Fun.ErrorDescription
'
n = 0
Nodes = 1
ReDim R(Rank, Rank), y(Nodes)
y(0) = Fun.Eval1(a): If Err Then GoTo Error_Handler
y(1) = Fun.Eval1(b): If Err Then GoTo Error_Handler
h = b - a
R(n, n) = h * (y(0) + y(1)) / 2
'start loop ---------------------------------
Do
    n = n + 1
    Nodes = 2 * Nodes
    h = h / 2
    'compute e reorganize the vector of function values
    ReDim Preserve y(Nodes)
    For i = Nodes To 1 Step -1
        If i Mod 2 = 0 Then
            y(i) = y(i / 2)
        Else
            x = a + i * h
            y(i) = Fun.Eval1(x)
            If Err Then GoTo Error_Handler
        End If
    Next i
    'now begin with Romberg method
    s = 0
    For i = 1 To Nodes
        s = s + y(i) + y(i - 1) 'trapezoidal formula
    Next
    R(n, 0) = h * s / 2
    For j = 1 To n
        y1 = R(n - 1, j - 1)
        y2 = R(n, j - 1)
        R(n, j) = y2 + (y2 - y1) / (4 ^ j - 1) 'Richardson's extrapolation
    Next j    'check error
    ErrLoop = Abs(R(n, n) - R(n, n - 1))
    If Abs(R(n, n)) > 10 Then
        ErrLoop = ErrLoop / Abs(R(n, n))
    End If

Loop Until ErrLoop < ErrMax Or n >= Rank
u(0) = R(n, n)
u(1) = 2 ^ n
u(2) = ErrLoop
```

```
Integral = u
Exit Function

Error_Handler:
Integral = Err.Description
End Function
```

Using this function is very simple. For example, try

=Integral(“sin(2*pi*x)+cos(2*pi*x)”, 0, 0.5)

It returns the value:

0.318309886183791 better approximate of 1E-16

You can also use this function in an Excel sheet
Note that the function can also return the max samples used and the last iteration error. To see this value select three adjacent cells, insert the function and give the CTRL+SHIFT+ENTER sequence key

A4 = {=Integral(A2;B2;C2;D2;E2)}

	A	B	C	D	E
1	f(x)	a	b	Rank	Accuracy
2	sin(2*pi*x)+cos(2*pi*x)	0	0.5	7	1.00E-14
3	Integral of f(x)	2^r	Est. Error		
4	0.318309886183791	64	5.551E-17		
5					

Example 6

This example shows how to evaluate a function defined by pieces of sub-functions
Each sub-function must be evaluated only when x belongs to its definition domain

$$f(x) = \begin{cases} x^2 & , \quad x \le 0 \\ \log(x+1) & , \; 0 < x \le 1 \\ \sqrt{x - \log 2} & , \; x \ge 2 \end{cases}$$

Note that you get an error if you try to calculate the 2th and 3td sub-functions in points which are out of their domain ranges.

For example, we get an error for x = –1 in 2th and 3td sub-functions, because we have

Log(0) = "?" and SQR(-1-log2) =”?”

So, it is indispensable to calculate each function only when it is needed.
Here is an example showing how to evaluate a segmented function with the domain constraints explained before.
There are a few comments, but the code is very clean and straight.
The constant `MAXFUN` sets the max pieces of function definition

```
Sub Eval_Pieces_Function()
Const MAXFUN = 5
Dim Domain$(1 To MAXFUN)
Dim Funct$(1 To MAXFUN)
Dim n&, i&, j&, Value_x#, Value_D#, Value_F#
Dim xmin#, xmax#, step#, Samples&, OK As Boolean
Dim Fun(1 To MAXFUN) As New clsMathParser
Dim Dom(1 To MAXFUN) As New clsMathParser
'--- Piecewise function definition -------------
'   x^2         for x<=0
'   log(x+1)    for 0<x<=1
'   sqr(x-log2) for x>1
'-----------------------------------------------
xmin = -2: xmax = 2: Samples = 10
n = 3  'max pieces
Funct(1) = "x^2":               Domain(1) = "x<=0"
Funct(2) = "log(x+1)":          Domain(2) = "(0<x)*(x<=1)"
Funct(3) = "sqr(x-log(2))":     Domain(3) = "x>1"
On Error GoTo Error_Handler
For i = 1 To n
    OK = Fun(i).StoreExpression(Funct(i))
    If Not OK Then Err.Raise 513, "Function" + Str(i), Fun(i).ErrorDescription
    OK = Dom(i).StoreExpression(Domain(i))
    If Not OK Then Err.Raise 514, "Domain" + Str(i), Fun(i).ErrorDescription
Next

step = (xmax - xmin) / (Samples - 1)
For j = 1 To Samples
    Value_x = xmin + (j - 1) * step 'value x
    For i = 1 To n
        Value_D = Dom(i).Eval1(Value_x)
        If Err Then GoTo Error_Handler
        If Value_D = 1 Then
            Value_F = Fun(i).Eval1(Value_x)
            If Err Then GoTo Error_Handler
            Exit For
        End If
    Next i
    Debug.Print "x=" + Str(Value_x); Tab(25); "f(x)=" + Str(Value_F)
Next j
Exit Sub
Error_Handler:
Debug.Print Err.Source, Err.Description
End Sub
```

The output, unless of errors, will be:

```
x=-2                    f(x)= 4
x=-1.55555555555556     f(x)= 2.41975308641975
x=-1.11111111111111     f(x)= 1.23456790123457
x=-.666666666666667     f(x)= .444444444444445
x=-.222222222222222     f(x)= 4.93827160493828E-02
x= .222222222222222     f(x)= 8.71501757189001E-02
x= .666666666666667     f(x)= .221848749616356
x= 1.11111111111111     f(x)= .900045063009142
x= 1.55555555555556     f(x)= 1.12005605212042
x= 2                    f(x)= 1.30344543588752
```

As from release 3.4 – thanks to the Conditioned Branch algorithm – it is also possible to evaluate a piecewise expression directly and in a very short way. Look how compact the code is in this case.

```
Sub Eval_Pieces_Function()
Dim j&, Value_x#, Value_F#
Dim xmin#, xmax#, step#, Samples&
Dim Fun As New clsMathParser
'--- Piecewise function definition -------------
'   x^2         for x<=0
'   log(x+1)    for 0<x<=1
'   sqr(x-log2) for x>1
'-----------------------------------------------
    f = "(x<=0)* x^2 + (0<x<=1)* Log(x+1) + (x>1)* Sqr(x-Log(2))"
'-----------------------------------------------
xmin = -2: xmax = 2: Samples = 10

If Not Fun.StoreExpression(f) Then GoTo Error_Handler

Samples = 10
step = (xmax - xmin) / (Samples - 1)
On Error GoTo Error_Handler
For j = 1 To Samples
    Value_x = xmin + (j - 1) * step 'value x
    Fun.Variable("x") = Value_x
    Value_F = Fun.Eval
    Debug.Print "x=" + str(Value_x); Tab(25); "f(x)=" + str(Value_F)
Next j
Exit Sub
Error_Handler:
Debug.Print Err.Source, Err.Description
End Sub
```

Example 7

This example shows how to evaluate a multivariable expression passing the variables values directly by their symbolic names using the *Variable* property (3.3.2 version or higher)
It evaluates the formula:

$$f(x,y,T,a) = (a^2 \cdot \exp(y/T)-x) \qquad \text{for} \quad x = 1.5 \quad , \quad y = 0.123 \quad , \quad T = 0.4 \quad , \quad a = 100$$

```
Sub test()
Dim ok As Boolean, f As Double, Formula As String
Dim Funct As New clsMathParser

Formula = "(a^2*exp(y/T)-x)"

ok = Funct.StoreExpression(Formula)  'parse function

If Not ok Then GoTo Error_Handler

On Error GoTo Error_Handler

'assign value passing the symbolic variable name
Funct.Variable("x") = 1.5
Funct.Variable("y") = 0.123
Funct.Variable("T") = 0.4
Funct.Variable("a") = 100

f = Funct.Eval  'evaluate function

Debug.Print "evaluation = "; f

Exit Sub
Error_Handler:
    Debug.Print Funct.ErrorDescription
End sub
```

Output will be:

```
evaluation =  13598.7080850196
```

Note how plain and compact the code is when using the string-assignment. It is only twice slower that the index-assignment but it looks much simpler.

Example 8

This example shows how to get the table ET (only for parser analysis or debugging). From the rel. 3.4 the ET table is returned as an array

```
Sub test_dump_1()
Dim strFun As String
Dim ETab()
Dim f As Double, OK As Boolean
Dim Funct As New clsMathParser

strFun = "2+x-x^2+x^3"

Debug.Print "Formula := "; strFun
OK = Funct.StoreExpression(strFun) 'parse function

If Not OK Then
    Debug.Print Funct.ErrorDescription
    Exit Sub
Else
    On Error Resume Next
    Funct.Variable("x") = 2
    f = Funct.Eval
    If Err = 0 Then
        Debug.Print "result = "; f
    Else
        Debug.Print Funct.ErrorDescription
        Exit Sub
    End If
End If

Funct.ET_Dump Etab   ‘<<< array table returned
For i = LBound(ETab, 1) To UBound(ETab, 1)
    If i > 0 Then Debug.Print i, Else Debug.Print "Id",
For j = 1 To UBound(ETab, 2)
    Debug.Print ETab(i, j),
Next j
    Debug.Print ""
Next i
End Sub
```

Examples of formula parsing

This paragraph shows a collection of structured tables for several functions. They are reported in the state following the final evaluation step

`f(x,n)= x^n/n!          for x=3, n=6`

Id	Fun	Max Arg	Arg(1).Id	Arg(1). Name	Arg(1). Value	Arg(2).Id	Arg(2). Name	Arg(2). Value	ArgOf	IndArg	Value	Level	Offset	Seq
1	^	2	1	x	3	2	n	6	2	1	729	3	2	3
2	/	2	0		729	0		720	0	0	1.0125	2	4	1
3	!	1	2	n	6	0		0	2	2	720	9	6	2

`f(x)= 1+(x-5)*3+8/(x+3)^2     for x=1`

Id	Fun	Max Arg	Arg(1).Id	Arg(1). Name	Arg(1). Value	Arg(2).Id	Arg(2). Name	Arg(2). Value	ArgOf	IndArg	Value	Level	Offset	Seq
1	+	2	0		1	0		-12	4	1	-11	1	2	2
2	-	2	1	x	1	0		5	3	1	-4	11	5	6
3	*	2	0		-4	0		3	1	2	-12	2	8	7
4	+	2	0		-11	0		0.5	0	0	-10.5	1	10	3
5	/	2	0		8	0		16	4	2	0.5	2	12	5
6	+	2	1	x	1	0		3	7	1	4	11	15	1
7	^	2	0		4	0		2	5	2	16	3	18	4

`f(x1,x2)= x1/(x1^2+x2^2)      for x1=1, x2=3`

Id	Fun	Max Arg	Arg(1).Id	Arg(1). Name	Arg(1). Value	Arg(2).Id	Arg(2). Name	Arg(2). Value	ArgOf	IndArg	Value	Level	Offset	Seq
1	/	2	1	x1	1	0		10	0	0	0.1	2	3	2
2	^	2	1	x1	1	0		2	3	1	1	13	7	4
3	+	2	0		1	0		9	1	2	10	11	9	3
4	^	2	2	x2	3	0		2	3	2	9	13	12	1

`f(x,y)= 1-exp(-(x^2+y^2))     for x=0.5, y=0.1`

Id	Fun	Max Arg	Arg(1).Id	Arg(1). Name	Arg(1). Value	Arg(2). Id	Arg(2). Name	Arg(2). Value	ArgOf	IndArg	Value	Level	Offset	Seq
1	-	2	0		1	0		0.77105	0	0	0.2289484	1	2	4
2	Exp	1	0		-0.26	0		0	1	2	0.7710516	10	6	6
3	-	2	0		0	0		0.26	2	1	-0.26	11	7	5
4	^	2	1	x	0.5	0		2	5	1	0.25	23	10	3
5	+	2	0		0.25	0		0.01	3	2	0.26	21	12	2
6	^	2	2	y	0.1	0		2	5	2	0.01	23	14	1

`f()= (4+sqr(3))*(4-sqr(3))         (no variables)`

Id	Fun	Max Arg	Arg(1).Id	Arg(1). Name	Arg(1). Value	Arg(2). Id	Arg(2). Name	Arg(2). Value	ArgOf	IndArg	Value	Level	Offset	Seq
1	+	2	0		4	0		1.73205	3	1	5.7320508	11	3	2
2	sqr	1	0		3	0		0	1	2	1.7320508	20	7	5
3	*	2	0		5.73205	0		2.26795	0	0	13	2	11	1
4	-	2	0		4	0		1.73205	3	2	2.2679492	11	14	4
5	sqr	1	0		3	0		0	4	2	1.7320508	20	18	3

```
f(x)= x^4-3*x^3+2*x^2-9*x+10        for x=3
```

Id	Fun	Max Arg	Arg(1). Id	Arg(1). Name	Arg(1). Value	Arg(2). Id	Arg(2). Name	Arg(2). Value	ArgOf	IndArg	Value	Level	Offset	Seq
1	^	2	1	x	3	0		4	2	1	81	3	2	1
2	-	2	0		81	0		81	5	1	0	1	4	4
3	*	2	0		3	0		27	2	2	81	2	6	7
4	^	2	1	x	3	0		3	3	2	27	3	8	3
5	+	2	0		0	0		18	8	1	18	1	10	6
6	*	2	0		2	0		9	5	2	18	2	12	9
7	^	2	1	x	3	0		2	6	2	9	3	14	2
8	-	2	0		18	0		27	10	1	-9	1	16	5
9	*	2	0		9	1	x	3	8	2	27	2	18	8
10	+	2	0		-9	0		10	0	0	1	1	20	10

```
(x<=0)* x^2 + (0<x<=1)* Log(x+1) + (x>1)* Sqr(x-Log(2))   for x = -2
```

Id	Fun	ArgTop	A1 Idx	Arg1 Name	Arg1 Value	A2 Idx	Arg2 Name	Arg2 Value	ArgOf	ArgIdx	Value	PriLvl	PosInExpr	PriIdx	Cond
1	<=	2	1	x	-2	0		0	2	1	1	311	4	1	0
2	*	2	0		1	0		4	4	1	4	3	7	5	0
3	^	2	1	x	-2	0		2	2	2	4	4	10	7	1
4	+	2	0		4	0		0	11	1	4	2	13	12	0
5	<	2	0		0	1	x	-2	6	1	0	311	17	6	0
6	*	2	0		0	0		1	8	1	0	303	19	16	0
7	<=	2	1	x	-2	0		1	6	2	1	311	20	10	0
8	*	2	0		0	0		0	4	2	0	3	23	15	0
9	Log	1	0		0	0		0	8	2	0	10	28	9	6
10	+	2	1	x	-2	0		1	9	1	0	12	30	14	6
11	+	2	0		4	0		0	0	0	4	2	34	3	0
12	>	2	1	x	-2	0		1	13	1	0	311	38	2	0
13	*	2	0		0	0		0	11	2	0	3	41	8	0
14	Sqr	1	0		0	0		0	13	2	0	10	46	13	12
15	-	2	1	x	-2	0		0	14	1	0	12	48	4	12
16	Log	1	0		2	0		0	15	2	0	20	52	11	12

```
2*(Dnorm(x, min(4,8), max(1,0.1))-1)
```

Id	Fun	ArgTop	A1 Idx	Arg1 Name	Arg1 Value	A2 Idx	Arg2 Name	Arg2 Value	ArgOf	ArgIdx	Value	PriLvl	PosIn Expr	PriIdx
1	*	2	0		2	0		-0.99965	0	0	-1.999	3	2	3
2	Dnorm	3	1	x	0.25	0		4	6	1	0.0004	20	11	5
3	min	2	0		4	0		8	2	2	4	30	17	4
4	@Right	2	0		8	0		1	2	3	1	20	20	2
5	max	2	0		1	0		0.1	4	2	1	30	26	6
6	-	2	0		0.00035	0		1	1	2	-1	12	32	1

List of Multiplying Factors

```
'extract multiply factor um_fact
Case "p": um_fact = -12
Case "n": um_fact = -9
Case "u": um_fact = -6
Case "µ": um_fact = -6        (not for oriental VBA version)
Case "m": um_fact = -3
Case "c": um_fact = -2
Case "k": um_fact = 3
Case "M": um_fact = 6
Case "G": um_fact = 9
Case "T": um_fact = 12
```

List of Unit of Measure Symbols

```
Select Case um_symb
Case "s":   v = 1 'second
Case "Hz":  v = 1 'frequency
Case "m":   v = 1 'meter
Case "g":   v = 0.001 'gramme
Case "rad", "Rad", "RAD": v = 1 'radiant
Case "S":   v = 1 'siemens
Case "V":   v = 1 'volt
Case "A":   v = 1 'ampere
Case "w":   v = 1 'watt
Case "F":   v = 1 'farad
Case "Ohm", "ohm": v = 1 'ohm
```

Special Functions

This is the list of all special functions calculated by clsMathParser, along with their parameters and definitions

`DNorm(x,μ,σ)`	Normal density function ∀ x, μ > 0 , σ > 0	$\frac{1}{s\sqrt{2p}}e^{-(x-m)^2/\left(2s^2\right)}$
`CNorm(x, μ,σ)`	Normal cumulative function ∀ x, μ > 0 , σ > 1	$\frac{1}{2}\left[1+\operatorname{erf}\left(\frac{x-m}{s\sqrt{2}}\right)\right]$
`DPoisson(x,n)`	Poisson density function v >0, n = 1, 2, 3 ...	$\frac{x^n e^{-x}}{n!}$
`CPoisson(x,k)`	Poisson cumulative function x >0, k = 1, 2, 3 ...	$\sum_{k=0}^{n}\frac{x^k e^{-x}}{k!}$
`DBinom(k,n,x)`	Binomial density for k successes for n trials k , n = 1, 2, 3…, k < n , x ≤1	$\binom{n}{k}x^n(1-x)^{n-k}$
`CBinom(k,n,x)`	Binomial cumulative for k successes for n k , n = 1, 2, 3…, k < n , x ≤1	$\sum_{i=0}^{k}\binom{n}{i}x^n(1-x)^{n-i}$
`Si(x)`	Sine integral ∀ x	$\int_0^x \frac{\sin t}{t}\,dt$
`Ci(x)`	Cosine integral x > 0	$-\int_x^\infty \frac{\cos t}{t}\,dt$
`FresnelS(x)`	Fresnel's sine integral ∀ x	$\sqrt{\frac{2}{p}}\int_0^x \sin t^2\,dt$
`FresnelC(x)`	Fresnel's cosine integral ∀ x	$\sqrt{\frac{2}{p}}\int_0^x \cos t^2\,dt$
`J0(x)`	Bessel's function of 1st kind x ≥ 0	$\sum_{k=0}^{\infty}(-1)^k\frac{x^{2k}}{4^k(k!)^2}$

Y0(x)	Bessel's function of 2nd kind x > 0	$-\frac{2}{\sqrt{p}\,\Gamma\left(\frac{1}{2}\right)}\int_{1}^{\infty}\frac{\cos(xt)}{\sqrt{t^2-1}}\,dt$
I0(x)	Bessel's function of 1st kind, modified x ≥ 0	$\sum_{k=0}^{\infty}\frac{x^{2k}}{4^k(k!)^2}$
K0(x)	Bessel's function of 2nd kind, modified x > 0	$\int_{0}^{\infty}\frac{\cos(xt)}{\sqrt{t^2+1}}dt$
BesselJ(x,n)	Bessel's function of 1st kind, nth order x ≥ 0 , n = 0, 1, 2…	$J_n(x)=\sum_{k=0}^{\infty}\frac{(-1)^k x^{2k+n}}{2^{2k+n}(n+k)!k!}$
BesselY(x,n)	Bessel's function of 2nd kind, nth order x > 0 , n = 1, 2…	$Y_n(x)=\frac{J_n(x)\cos(np)-J_{-n}(x)}{\sin(np)}$
BesselI(x,n)	Bessel's function of 1st kind, nth order, modified x ≥ 0 , n = 0, 1, 2…	$I_n(x)=i^{-n}J_n(ix)=e^{-np\,i/2}J_n(xe^{ip/2})$
BesselK(x,n)	Bessel's function of 2nd kind, nth order, modified x > 0 , n = 0, 1, 2…	$K_n(x)=\frac{p}{2}\frac{I_{-n}(x)-I_n(x)}{\sin(np)}$
HypGeom(z,a,b,c)	Hypergeometric function -1 < x <1 a,b >0 c ≠ 0, −1, −2…	$1+\frac{ab}{1!c}x+\frac{a(a+1)b(b+1)}{2!c(c+1)}x^2+...$
PolyCh(x,n)	Chebyshev 's polynomials ∀x , orthog. for **-1** ≤ **x** ≤**1**	$T_n(x)=\frac{n}{2}\sum_{k=0}^{[n/2]}\frac{(-1)^k}{n-k}\binom{n-k}{k}(2x)^{n-2k}$
PolyLe(x,n)	Legendre's polynomials ∀ x , orthog. for **-1** ≤ **x** ≤**1**	$P_n(x)=\frac{1}{2^n}\sum_{k=0}^{[n/2]}\frac{(-1)^k(2n-2k)!}{k!(n-k)!(n-2k)!}x^{n-2k}$
PolyLa(x,n)	Laguerre's polynomials ∀ x , orthog. for **0** ≤ **x** ≤**1**	$L_n(x)=\frac{e^x}{n!}\frac{d^n}{dx^n}\left(x^n e^{-x}\right)$
PolyHe(x,n)	Hermite's polynomials ∀ x , orthog. for **−∞** ≤ **x** ≤ **+∞**	$H_n(x)=(-1)^n e^{x^2}\frac{d^n}{dx^n}e^{-x^2}$

`AiryA(x)`	Airy function Ai(x) ∀ x	$\frac{1}{3^{2/3}p}\sum_{n=0}^{\infty}\frac{\Gamma\left(\frac{1}{3}(n+1)\right)}{n!}\left(3^{1/3}x\right)^{n}\sin\left(\frac{2(n+1)p}{3}\right)$
`AiryB(x)`	Airy function Bi(x) ∀ x	$\frac{1}{3^{1/6}p}\sum_{n=0}^{\infty}\frac{\Gamma\left(\frac{1}{3}(n+1)\right)}{n!}\left(3^{1/3}x\right)^{n}\left\|\sin\left(\frac{2(n+1)p}{3}\right)\right\|$
`Elli1(`f**`,k)`**	Elliptic integral of 1st kind ∀ ϕ 0 < k < 1	$F(f,k)=\int_{0}^{f}\frac{dq}{\sqrt{1-k^{2}\sin^{2}q}}$
`Elli2(`f**`,k)`**	Elliptic integral of 2nd kind ∀ ϕ 0 < k < 1	$E(f,k)=\int_{0}^{f}\sqrt{1-k^{2}\sin^{2}q}\ dq$
`Erf(x)`	Error Gauss's function ∀ x	$\frac{2}{\sqrt{p}}\int_{0}^{x}e^{-t^{2}}dt$
`gamma(x)`	Gamma function ∀ x x ≠ 0, −1, −2, −3…	$\Gamma(x)=\int_{0}^{\infty}t^{x-1}\,e^{-t}dt$
`gammaln(x)`	Logarithm Gamma function x >0	$\log\left(\Gamma(x)\right)$
`gammai(a,x)`	Gamma Incomplete function ∀ x a > 0	$\Gamma(a,x)=\int_{x}^{\infty}t^{a-1}\,e^{-t}dt$
`digamma(x)`	Digamma function x ≠ 0, −1, −2, −3…	$\Psi(x)=\frac{d}{dx}\log\left(\Gamma(x)\right)=\frac{\Gamma'(x)}{\Gamma(x)}$
`beta(a,b)`	Beta function a>0 , b>0	$\int_{0}^{1}t^{a-1}(1-t)^{b-1}dt$
`betaI(x,a,b)`	Beta Incomplete function ∀ x , a >0 , b >0	$\int_{0}^{x}v^{a-1}(1-v)^{b-1}dv$
`Ei(x)`	Exponential integral x ≠0	$Ei(x)=\int_{1}^{\infty}\frac{e^{-tx}}{t}dt=\int_{x}^{\infty}\frac{e^{-t}}{t}dt$

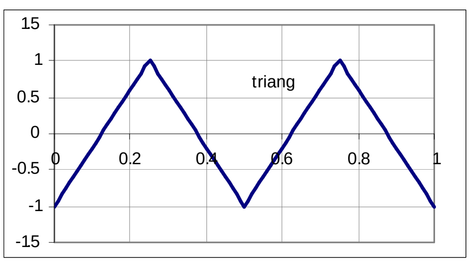
`Ein(x,n)`	Exponential integral of n order x >0 , n = 1, 2, 3…	$En(x,n) = \int_{1}^{\infty} \frac{e^{-tx}}{t^{n}}\, dt$
`zeta(x)`	zeta Riemman's function x<-1 or x>1	$z\,(x) = \frac{1}{\Gamma(x)} \int_{0}^{\infty} \frac{t^{x-1}}{e^{t}-1}\, dt$

Special Periodic Functions

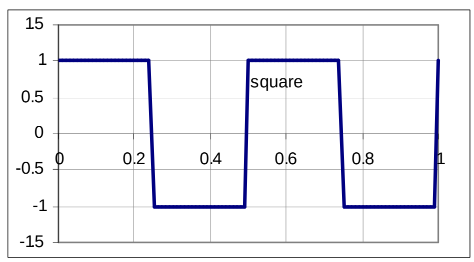
This is the list of all special periodic functions calculated by clsMathParser, along with their parameters and plots

Function name	Description	parameters
`WTRI(t,p)`	Triangular wave	t = time, p = period
`WSQR(t,p)`	Square wave	t = time, p = period
`WRECT(t,p,d)`	Rectangular wave	t = time, p = period, d= duty-cicle
`WTRAPEZ(t,p,d)`	Trapez. wave	t = time, p = period, d= duty-cicle
`WSAW(t,p)`	Saw wave	t = time, p = period
`WRAISE(t,p)`	Rampa wave	t = time, p = period
`WLIN(t,p,d)`	Linear wave	t = time, p = period, d= duty-cicle
`WPULSE(t,p,d)`	Rectangular pulse wave	t = time, p = period, d= duty-cicle
`WSTEPS(t,p, n)`	Steps wave	t = time, p = period, n = steps number
`WEXP(t,p,a)`	Exponential pulse wave	t = time, p = period, a= dumping factor
`WEXPB(t,p,a)`	Exponential bipolar pulse wave	t = time, p = period, a= dumping factor
`WPULSEF(t,p,a)`	Filtered pulse wave	t = time, p = period, a= dumping factor
`WRING(t,p,a,fm)`	Ringing wave	t = time, p = period, a= dumping factor, fm = frequency
`WPARAB(t,p)`	Parabolic pulse wave	t = time, p = period
`WRIPPLE(t,p,a)`	Ripple wave	t = time, p = period, a= dumping factor
`WAM(t,fo,fm,m)`	Amplitude modulation	t = time, p = period, fo = carrier freq., fm = modulation freq., m = modulation factor
`WFM(t,fo,fm,m)`	Frequecy modulation	t = time, p = period, fo = carrier freq., fm = modulation freq., m = modulation factor

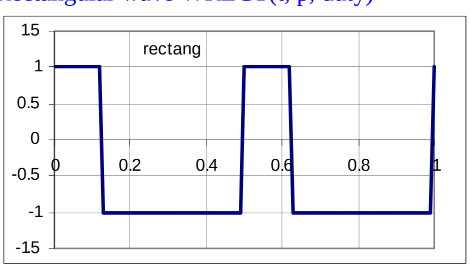
Triangular wave WTRI(t, p)



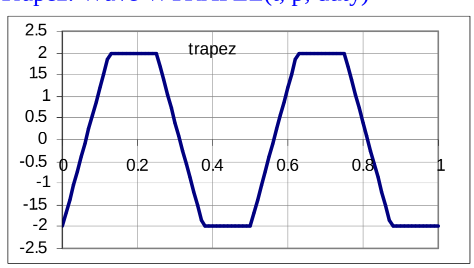
Square wave WSQR(t, p)



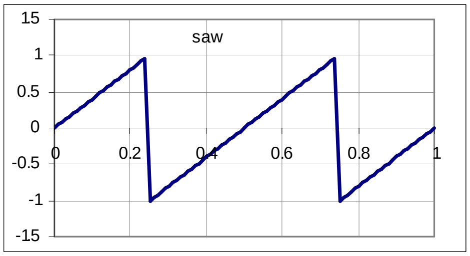
Rectangular wave WRECT(t, p, duty)



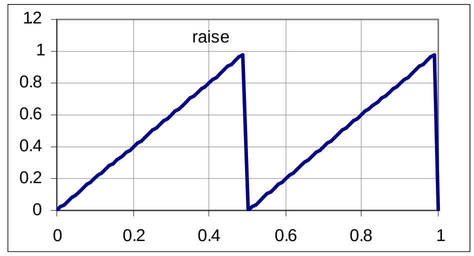
Trapez. Wave WTRAPEZ(t, p, duty)



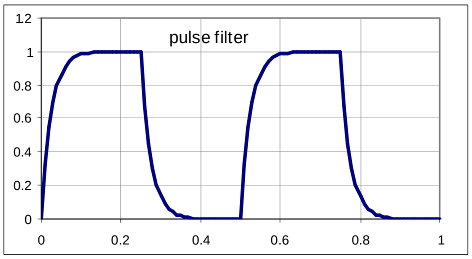
Saw wave WSAW(t, p)



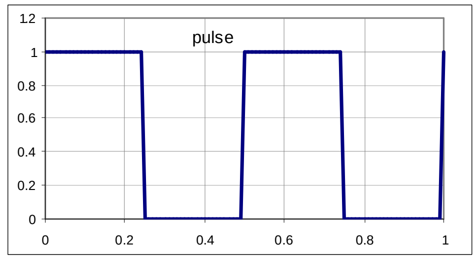
Rampa wave WRAISE(t, p)



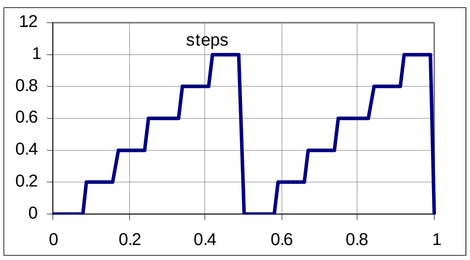
Filtered pulse wave WPULSEF(t, p, a)



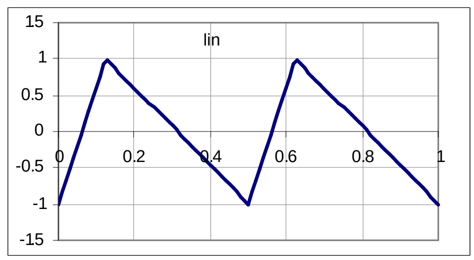
Rectangular pulse wave WPULSE(t, p, duty)



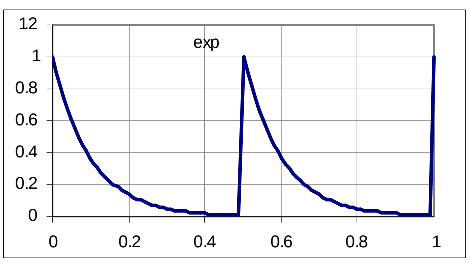
Steps wave WSTEPS(t, p, n)



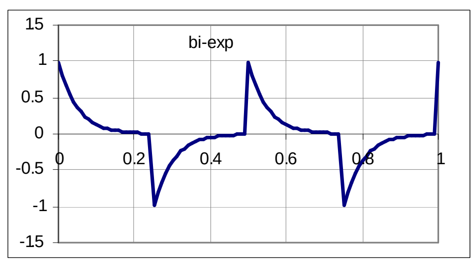
Linear wave WLIN(t, p, duty)



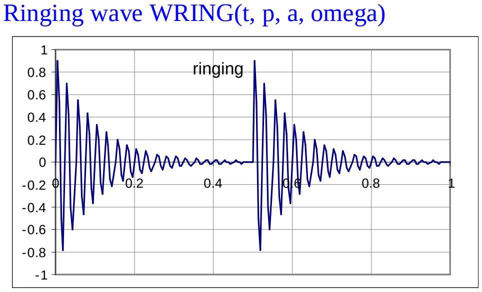
Exponential pulse wave WEXP(t,p,a)



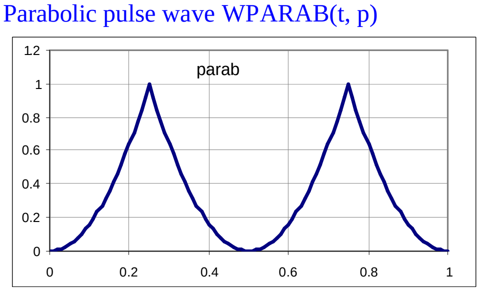
Exponential bipolar pulse wave WEXPB(t,p,a)



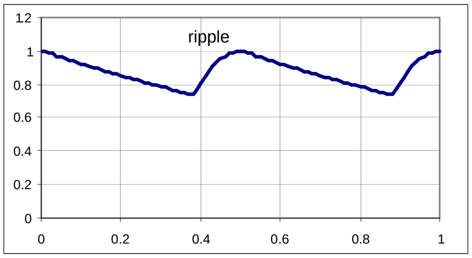
Ringing wave WRING(t, p, a, omega)



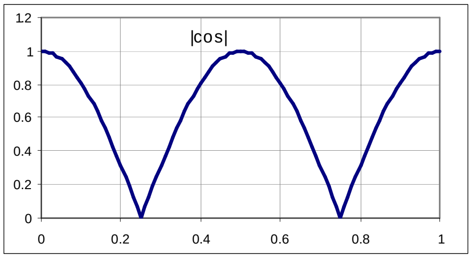
Parabolic pulse wave WPARAB(t, p)



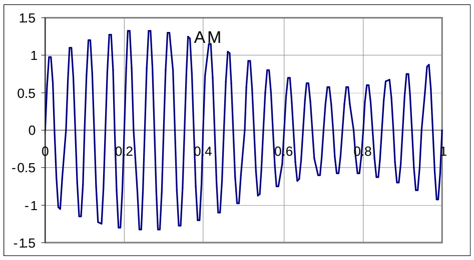
Ripple wave WRIPPLE(t, p, a)



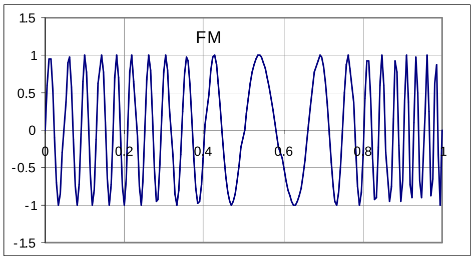
Rectifiered wave WSINREC(t, p)



Amplitude modulation WAM(t, fo, fm, m)



Frequecy modulation WFM(t, fo, fm, m)



Error Message Table

From the version 4.1, all error messages are collected into an internal table, called ErrorTbl.
The index of the table is the number returned by the property **ErrorId**
The massages can be parametric. The symbol "$" is the placeholder for parameter substitution

```
ErrorTbl(1) = "too many variables"
ErrorTbl(2) = "Variable not found"
ErrorTbl(3) = "" 'spare
ErrorTbl(4) = "abs symbols |.| mismatch"
ErrorTbl(5) = "Syntax error at pos: $"
ErrorTbl(6) = "Function < $ > unknown at pos: $"
ErrorTbl(7) = "Too many closing brackets at pos: $"
ErrorTbl(8) = "missing argument"
ErrorTbl(9) = "Too many arguments at pos: $"
ErrorTbl(10) = "" 'spare
ErrorTbl(11) = "Not enough closing brackets"
ErrorTbl(12) = "Syntax error: $"
ErrorTbl(13) = "Function < $ > missing?"
ErrorTbl(14) = "Evaluation error < $($) > at pos: $"
ErrorTbl(15) = "Evaluation error < $($" & ArgSep & " $) > at pos: $"
ErrorTbl(16) = "Evaluation error < $ $ $ > at pos: $"
ErrorTbl(17) = "Evaluation error < $($) > at pos: $"
ErrorTbl(18) = "Variable < $ > not assigned"
ErrorTbl(19) = "Constant unknown: $"
ErrorTbl(20) = "Too many operations"
ErrorTbl(21) = "Wrong DMS format"
ErrorTbl(22) = "No DMS format"
```

The internal function **getMsg**(Id, Optional p1, Optional p2, Optional p3, Optional p4) performs the parameter substitution

Example. The string `"67><"` contains a syntax error at position 4. To build this error message the parser pass the Id = 5 and the parameter p1 = 4

```
    getMsg(5, 4)  ⇒ "Syntax error at pos: 4"
```

Example. The string `"3x+4*(log(x)+1)"` raises an error for x = 0 and at position 9. To build this error message the parser pass the Id = 14, the parameter p1 = "log", p2 = 0 and p3 = 9

```
    getMsg(14, "log", 0, 9)  ⇒ " Evaluation error < log(0) > at pos: 10"
```

As we can see, the placeholder "$" are exactly substitutes with the correspondent parameters

Last printed: Feb. 2007
Italy