USER-DEFINED FUNCTIONS IN SMATH STUDIO

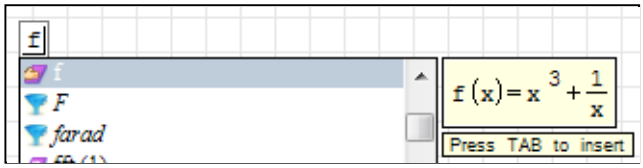
[rev.08 | 2016.12.08 | SS 0.98.6179]

Inline functions

To define a function, type function's name and open a parenthesis "(", write the argument name, move on the right of the closing parenthesis and then type the define operator ":".

[1.1] $f(x) := x^3 + \frac{1}{x}$

$f(2) = 8.5$ <- type f and look at the dynamic assistance, you will find the functiom name and his content.



IMPORTANT

type = to evaluate numerically $f(2) = 8.5$

type CTRL+. to evaluate symbolically $f(2) = \frac{17}{2}$

evaluation can be changed by context menu:
1. right click on the math region
2. Optimization > [Symbolic/Numeric/None]

Like variables functions are case sensitive, hence f(x)≠F(x)

[1.2] $F(x) := x + 3$

$f(2) = 8.5$

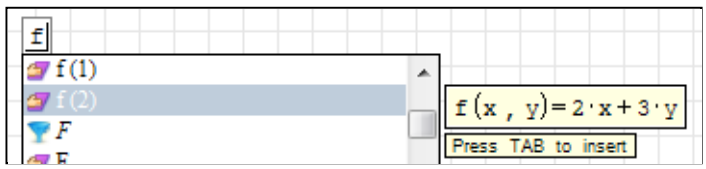
$F(2) = 5$

It is possible to define several functions with the same name, if they have a different number of arguments.

[1.3] $f(x, y) := 2 \cdot x + 3 \cdot y = 2 \cdot x + 3 \cdot y$

$f(1, 2) = 8$

$f(2) = 8.5$ <- f(x) is still defined; if you look at the dynamic assistance, 2 items are available now, f(1) and f(2). The number shows the arguments required by each function, whereas the name without numbers is used when there is just one function (see [1.1])



It is possible to define again the function

[1.4] $f(x) := x^2 - 1$

$f(2) = 3$

Evaluable things such math regions are ordered in the canvas from left to right and then from top to bottom, with reference to the top-left corner of the region.

In SS a valid variable name cannot start with a number; it cannot contains whitespaces nor the character used as argument separator in functions, as well as @ and the math symbols: ()[]{}+-/*=:≡<>≤≥≠¬&|¤±...

When you define a function, argument's name can be only: a variable name or a function; units and empty operand can be used only as placeholders.

IMPORTANT

[1.5]

[A] $f(1) := a + b + 3 \cdot i$ <- the name of an argument cannot be a number

$f(1.5) := a + b + 3 \cdot i$ lastError = "Syntax is incorrect." lastError contains the last error detected up to the point where it is used

if your intent is to define the value of a function in a point, use an if statement, booleans or other functions

$f(x) := \text{if } x = 2 \; 10 \; \text{else} \; 2 \cdot x$

$g(x) := (x = 2) \cdot (10) + (x \neq 2) \cdot (2 \cdot x)$

$f(1.9) = 3.8$ $g(1.9) = 3.8$

$f(2) = 10$ $g(2) = 10$

$f(2.1) = 4.2$ $g(2.1) = 4.2$

[B] $f(x3) := x3 + 4$ <- the variable name can contain numbers (not as 1st character)

$f(2) = 6$

[C] $f(\_3) := \_3 + 5$ <- any non-number character is allowed as first character

$f(2) = 7$

[D] $f(\#3) := \#3 + 6$ <- # is the character of the empty placeholder; to use it as 1st character you have to type at least the one that will be the 2nd character, move back the cursor, and then you can type # in front of it

$f(2) = 8$

[E] $f(\blacksquare) := x + 9$ <- dummy argument (argument not used in the function - will be discussed later in [3.14])

$f(2) = 9 + x$

[F] $f(nothing) := x + 7$ <- dummy argument

$f(2) = 7 + x$

[G] $f(g(x)) := 2 \cdot g(0) + 3 \cdot g\left(\frac{\pi}{5}\right)$ <- a function is a valid argument (will be further discussed from [3.10] to [3.13])

$f(F(x)) = 16.885$ $2 \cdot F(0) + 3 \cdot F\left(\frac{\pi}{5}\right) = 16.885$

$f(\sin(x)) = 1.7634$ $2 \cdot \sin(0) + 3 \cdot \sin\left(\frac{\pi}{5}\right) = 1.7634$

$f(\exp(x)) = 7.6234$ $2 \cdot \exp(0) + 3 \cdot \exp\left(\frac{\pi}{5}\right) = 7.6234$

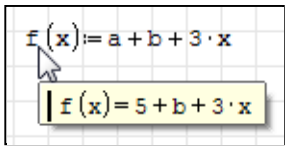
You can access variables from the canvas (not passed by the arguments), defined or even not yet defined (from left to right and from top to bottom, see [1.4]).

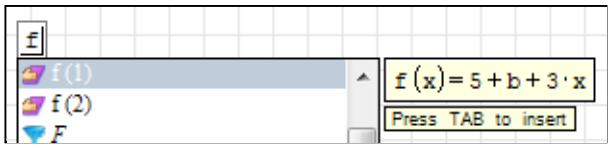
[1.6] $a := 5$

$f(x) := a + b + 3 \cdot x$ <- hold the mouse over the function to see how "a" and "b" are stored; you can do the same from the dynamic assistance

[A] $b := 2$

$f(2) = 13$





IMPORTANT

[B] $b := 3$

$f(2) = 14$ <- if a variable is not yet defined when you define the function, it is kept as symbolical link (the name is stored); hence if afterwards you change the value of the variable, the function will returns a different result.

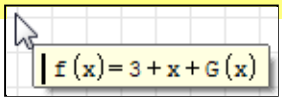
IMPORTANT

[C] $a := 10$

$f(2) = 14$ <- if a variable is already defined when you define the function, the value is stored; hence if afterwards you change the value of the variable, the function will returns always the same result.

[1.7] $f(x) := F(x) + G(x)$ <- this principle applies to anything in the right hand side of assignments, even functions

$f(1) = 4 + G(1)$



[A] $F(x) := 5$

$f(1) = 4 + G(1)$ <- the value of F(x) was stored as it was at f(x) definition, thus when you change F(x) the new value is NOT used in f(x)

[B] $G(x) := 2 \cdot x$

$f(1) = 6$

[C] $G(x) := 3 \cdot x$

$f(1) = 7$ <- G(x) wasn't defined at f(x) definition, thus his name is stored and when you change it the new content is used in f(x)

You can use vectorize() to extend scalar logic to matrices/vectors elements avoiding the use of loops; this can be done applying vectorize on you function or inside your function.

[1.8] $f(x) := |x| - 1$

$$M := \begin{bmatrix} 1 & 2 & 3 \\ -1 & -2 & -3 \end{bmatrix}$$

$f(M) =$ ▮ lastError= "Argument must be scalar."

$$\overrightarrow{f(M)} = \begin{bmatrix} 0 & 1 & 2 \\ 0 & 1 & 2 \end{bmatrix}$$ <- the scalar logic is extended to any element of the matrix, because requested by vectorize(...)

$\overrightarrow{f(2)} = 1$

[1.9] $f(x) := \overrightarrow{|x|} - 1$

$$f(M) = \begin{bmatrix} 0 & 1 & 2 \\ 0 & 1 & 2 \end{bmatrix}$$ <- the scalar logic is always extended to any matrix's element

$f(2) = 1$

Dependent variables

Like in paper-math you may want to have a function without explicit arguments: $y := f(x)$

[2.1] $y := 2 \cdot x + 3$ $\qquad \dot{x} := \frac{d}{d\,t} x(t)$

[A] $x := 2$

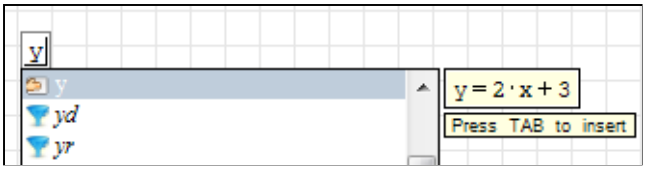
$y = 7$

[B] $x := 3$

$y = 9$

[C] $\frac{d}{d\,x} y = 2$ <- even if x is defined at the execution, y keeps the symbolical link (look at the dynamic assistance and [1.5][B])

$\int_0^2 y \, d\,x = 10$



Like in functions, it is possible to use defined/undefined variables from the canvas.

[2.2] $b := 5$

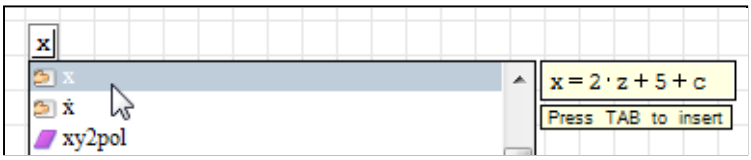
$x := 2 \cdot z + b + c = 5 + c + 2 \cdot z$ <- hold the mouse over the variable to see how "b" and "c" are managed: c was undefined and his name is stored, b was defined and his value is replaced and stored.

[A] $c := 2$

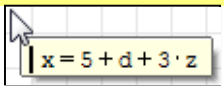
$x = 7 + 2 \cdot z$

[B] $c := 4$

$x = 9 + 2 \cdot z$



[2.3] $x := b + d + G(z)$ $\qquad G(z) = 3 \cdot z$ <- hold the mouse over the definition of x



[A] $z := 3 \quad d := 3$

$x = 17$

d wasn't defined before x's definition
=> his name is stored
=> defining d with different values, you obtain different results when you evaluate again x

[B] $d := 2$

$x = 16$

[C] $z := 2$

$x = 13$

[D] $b := 3$

$x = 13$

b was defined before x's definition
=> his value (5) is stored in x
=> doesn't matter if you define b in another way

[E] $G(z) := 6 \cdot z$

$x = 13$

G(z) was defined before x's definition
=> his value (3*z) is stored in x
=> doesn't matter if you define G(z) in another way

To make complex tasks, it is possible to use the line() function in the Right Hand Side. This function and his block of statements applied right after the assignment operator is called Procedure.
A procedure acts like a logical function (is fully evaluated anytime is called).
line() is available even in the "Programming" toolbox on the right side of the program.

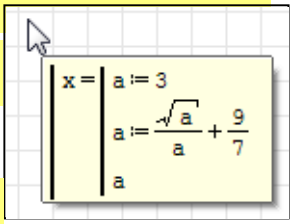
Type an "argument separator" character to add a placeholder in the line.

[2.4] $x := \left| \begin{array}{l} a := 3 \\ a := \frac{\sqrt{a}}{a} + \frac{9}{7} \\ a \end{array} \right.$

$x = 1.863065$

<- hold the mouse over the region to see how the procedure is stored; as you can see the whole logic is kept in memory, no replacements are made

<- The last placeholder is used to output the result



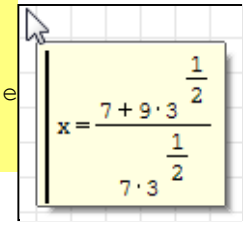
[A] $v := x$

$x = 1.863065$

[B] $x := x$

$x = 1.863065$

<- if the logic in the procedure doesn't depends from outer variables, you can speed-up calculations inside loops by evaluating it at once before his repetitive use, assigning the variable to another one (even with the same name)



What does it means here "doesn't depends from outer variables"?
When the procedure will be evaluated if a variable is not defined inside the procedure but is available in the parent level, his value will be used.

[2.5] $x := \left| \begin{array}{l} a := 3 \\ a + b \end{array} \right.$

<- even if it is available in the canvas, "b" is an unknown inside the procedure (not defined anywhere locally, is a "local unknown")

[A] $b := 4$

$x = 7$ <- 3+4=7; "b" is used with his canvas' value

$a = 10$ <- the canvas value of "a" is still the same

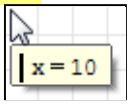
[B] $b := 7$

$x = 10$ <- 3+7=10; changing the value of b changes even the output value of x

[C] $x := x$ <- store the value of x inside a new variable named x

$b := 100$

$x = 10$ <- the dependancy from b is gone (see [2.4])



What if the local unknown is unknown even in the parent level?

[2.6] $x := \left| \begin{array}{l} a := 3 \\ a + w \end{array} \right.$

$w =$ ▮ lastError= "w - not defined."

IsDefined(w) = 0

$x = 3 + w$ <- the local unknow is exposed to the canvas

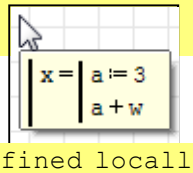
$y := x$ <- in this case is even possible to partially evaluate x and store it in a new variable (even "x") to speed up further calculations; when some numerical function acts on the unknown such evaluation it might be not possible and the new variable will point the original (y=x)

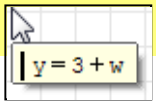
$w := 5$

$x = 8$

$w := 10$

$x = 13$ <- as for [2.5], since "w" wasn't defined locally, the current value will be used





Now let's go deeper; we know that a canvas value may be assigned (if available) to a local unknow when the procedure is being evaluation, but in what point of our function the canvas value is applied to our unknown when executed?

[2.7]

```
x:= | a:= b+3
    | b:= 20
    | a:= a+b
    | a
```

<- "b" is unknown in the procedure
<- from this point on, "b" is known, his local value will be used in following assignments

[A] b:= 2

x = 25 <- (2+3)+20=25

b = 2 <- the canvas value of "b" is still the same

[B] b:= 100

x = 123 <- (100+3)+20=123

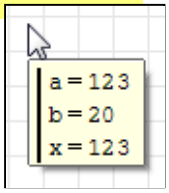
If line() is no more applied right after the definition operator, becomes a passive wrapper for his content; anything will be immediatly evaluated

[2.8]

```
x:=+| a:= b+3
    | b:= 20
    | a:= a+b
    | a
```

<- notice the + before the line(); a parenthesis will do the same (or sys(), etc...)

<- last placeholder is still the output value



a = 123 <- assignments made inside line() are available on the canvas, as well as "x"

b = 20

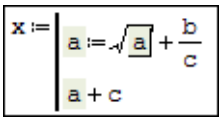
x = 123

In a procedure you can rename a variable at once using the F8 key.

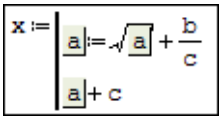
[2.9]

$$x:=\left|\begin{array}{l} a:=\sqrt{a}+\frac{b}{c} \\ a+c \end{array}\right.$$

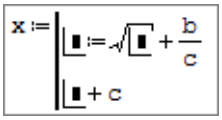
<- move the cursor above a variable (i.e. a); you should see all the occourrences of that item light-grayed

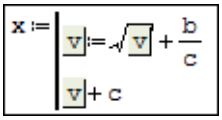


press the F8 key of your keyboard, aa cursor will appears in each of them

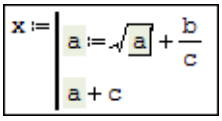


navigate with the keyboard arrows, type the new name, delete unwanted characters





use the escape key to go back to the normal mode (the same can be addressed with a mouse click)



Programming functions

Procedures can be used even to to make complex tasks in functions:

[3.1] $f(x) := \begin{vmatrix} a := 3 \cdot x + 4 \\ a \end{vmatrix}$ <- x is an argument of f
<- The last placeholder is used to output values

$f(2) = 10$

Arguments in programming functions are passed by reference, this means that an input variable can be modified from inside the function when you use the assignment operator (:=) to that variable inside the function

IMPORTANT

[3.2] $f(x) := \begin{vmatrix} x := x + 1 \\ out := x^2 \\ out \end{vmatrix}$ <- assignment operator applied to an input argument (x:=x+1)

[A] $a := 2$

$f(a) = 9$ <- result of the function

$a = 3$ <- Pass by reference side effect

[B] $a := 1$

```
while a < 10
  b := f(a)
```

Note that the sintax is very compact

$b = 100$ <- result of the function after the loop

$a = 10$ <- Pass by reference side effect

The loop:

1st iteration -> in the canvas a:=1

in the function x:= 2

since assignment is used on x -> a:= 2

$out := 2^2 = 4$

2nd iteration -> in the canvas a:=2

in the function x:= 3

since assignment is used on x -> a:= 3

$out := 3^2 = 9$

and so on ...

[C] $a := 1$

```
while a < 10
  a := f(a)
```
<- result of the function is also the input variable

$a = 25$ <- Pass by reference side effect overridden by the assignment on the canvas (because is the last executed)

The loop:

1st iteration -> in the canvas a:=1

in the function x:= 2

since assignment is used on x -> a:= 2

out:= $2^2 = 4$

since the result is assigned to a -> a:= 4

2nd iteration -> in the canvas a:=4

in the function x:= 5

since assignment is used on x -> a:= 5

out:= $5^2 = 25$

since the result is assigned to a -> a:= 25

a=25 >= 10 -> [END OF THE LOOP]

A variable not defined in the canvas can be set in the canvas from inside the function, providing this variable is used as function's argument and a value assigned to it in the function

[3.3] $f(x, msg) :=$

```
a := x^2 - 10
if a < 0
  msg := "result not allowable"
else
  msg := "ok"
a
```

$f(5, message) = 15$ <- result of the function

$message = "ok"$ <- Pass by reference side effect

If you expect to use the input variable as target of the calculations inside the function and you want to avoid side effects, you have to transfer the value to another variable and use the latest

[3.4] $f(x) :=$

$v := x$

$v := v + \frac{\sqrt{v}}{2} + v^2$

v

$a := 2$

$f(a) = 6.7071$

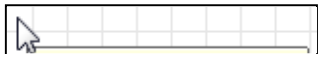
$a = 2$ <- No side effects (no assignment used on the function arguments)

Like for simple procedures, on evaluation "local unknowns" can get their value from the parent level (see [2.5])

[3.5] $f(x) :=$

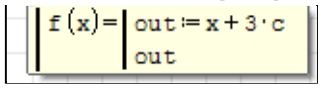
$out := x + 3 \cdot c$ <- c is an unknown in the function (not defined locally)

out



[A] $a := 2$

$c := 3$



$f(a) = 11$ <- symbolically the output should be f(2)=2+3*c
c exists in the canvas -> f(2)=2+3*3=11

[B] $c := 4$

$f(a) = 14$ <- symbolically the output should be f(2)=2+3*c
c exists in the canvas -> f(2)=2+3*4=14

[3.6] $f(x) :=$
$v := x + 3 \cdot k$
$k := 5$
$v := v + k$
v

<- here k is unknown in the function => stored as name
<- from this point k is known in the function (local variable)

[A] $f(2) = 22$ <- 2+3*5+5=22

$k =$ ▮ lastError= "k - not defined." <- no side effects

IMPORTANT

The use of local unknowns (or better "unassigned local variables") might lead you to unwanted results in some circustances.

One typical case is using features like "dynamic arrays" of matrices/systems.
This feature allow to assign elements' values without initializing the matrix/system.
If the target matrix is not defined locally, a matrix from the canvas may be used either intentionally or unintentionally; if you need to avoid this behavior you have to define the variable before using it (see example below)

[3.7] $f(x) := \left| \begin{array}{l} M_{1\,1} := x \\ M_{1\,2} := 2 \cdot x \\ M \end{array} \right.$ $g(x) := \left| \begin{array}{l} M := 0 \\ M_{1\,1} := x \\ M_{1\,2} := 2 \cdot x \\ M \end{array} \right.$

[A] $M := \begin{bmatrix} 1 & 1 \end{bmatrix}$

$f(2) = \begin{bmatrix} 2 & 4 \end{bmatrix}$ $g(2) = \begin{bmatrix} 2 & 4 \end{bmatrix}$ <- you didn't notice differences

[B] $M := \begin{bmatrix} 1 & 1 & 1 \end{bmatrix}$

$f(2) = \begin{bmatrix} 2 & 4 & 1 \end{bmatrix}$ $g(2) = \begin{bmatrix} 2 & 4 \end{bmatrix}$ <- in f(x), M from canvas is used as template

[C] $M := \begin{bmatrix} 1 \\ 1 \end{bmatrix}$

$f(2) = \begin{bmatrix} 2 & 4 \\ 1 & 0 \end{bmatrix}$ $g(2) = \begin{bmatrix} 2 & 4 \end{bmatrix}$

Another typical case might happen working with symbolical functions.
You might use a local unknown in the RHS of something, with the precise scope of being an unknown. If at the evaluation of the function a variable with the same name exists on the parent level, the result will be different from the one you planned (see below)

[3.8] $f(x) := \left| \begin{array}{l} v1 := x + 2 \cdot y1 \\ v2 := x + 2 \cdot y2 \\ r1 := \dfrac{d}{d\,y1}(v1) \\ r2 := \dfrac{d}{d\,y2}(v2) \\ r3 := \dfrac{d}{d\,y2}(x + 2 \cdot y2) \\ \begin{cases} r1 \\ r2 \\ r3 \end{cases} \end{array} \right.$

$y2 := 5$ $\text{IsDefined}(y1) = 0$ $\text{IsDefined}(y2) = 1$

$f(3) = \begin{cases} 2 \\ 0 \\ 2 \end{cases}$

v1 stores 3+2*y1, because y1 doesn't exists anywhere;

v2 stores 13 (3+2*5), because y2=5 exists on the top level;

r1 stores 2 (expected value), because v1 contains the unknown y1;

r2 stores 0, because you are differentiating 13 (a contant);

r3 stores 2 (expected value), because the order of computation; first of all differentiation is made (calculation of the RHS), then the assignments looks to replace unknowns from RHS to store it in the variable LHS variable

This might become a huge problem on big worksheets or if you share the document and who will reuse your code doesn't realize it (maybe even you, after a long time).

IMPORTANT

What you can do to avoid this problem?
- The strong way: initialize local variables (M:0, y2:line(y2,1,1), Clear(y1,y2)*, ...);
- A less strong way: use "unique" names (x#, #x, _x, ...);
Most important: always document your functions (right click -> show description);

* Clear(...) is a function from "Custom Functions" plug-in;
basically yo can do almost the same with the following syntax:

$x := \left| x \right.$

[3.9]

$$f1(x) := \left| \begin{array}{l} \text{Clear}(y1, y2) \\ v1 := x + 2 \cdot y1 \\ v2 := x + 2 \cdot y2 \\ r1 := \frac{d}{d\,y1}(v1) \\ r2 := \frac{d}{d\,y2}(v2) \\ r3 := \frac{d}{d\,y2}(x + 2 \cdot y2) \\ \left\{ \begin{array}{l} r1 \\ r2 \\ r3 \end{array} \right. \end{array} \right.$$

$$f2(x) := \left| \begin{array}{l} y1 := \left| y1 \right. \\ y2 := \left| y2 \right. \\ v1 := x + 2 \cdot y1 \\ v2 := x + 2 \cdot y2 \\ r1 := \frac{d}{d\,y1}(v1) \\ r2 := \frac{d}{d\,y2}(v2) \\ r3 := \frac{d}{d\,y2}(x + 2 \cdot y2) \\ \left\{ \begin{array}{l} r1 \\ r2 \\ r3 \end{array} \right. \end{array} \right.$$

note: this doesn't fix the issue, makes it just more difficult to happen (still possible)

$$f3(x) := \left| \begin{array}{l} v1 := x + 2 \cdot y\#1 \\ v2 := x + 2 \cdot y\#2 \\ r1 := \frac{d}{d\,y\#1}(v1) \\ r2 := \frac{d}{d\,y\#2}(v2) \\ r3 := \frac{d}{d\,y\#2}(x + 2 \cdot y\#2) \\ \left\{ \begin{array}{l} r1 \\ r2 \\ r3 \end{array} \right. \end{array} \right.$$

$y2 := 5$

$$f1(3) = \left\{ \begin{array}{l} 2 \\ 2 \\ 2 \end{array} \right.$$

$$f2(3) = \left\{ \begin{array}{l} 2 \\ 2 \\ 2 \end{array} \right.$$

$$f3(3) = \left\{ \begin{array}{l} 2 \\ 2 \\ 2 \end{array} \right.$$

Can I pass a function as function argument? Yes, you can

[3.10]

$$f(g(x)) := \left| \begin{array}{l} out := g(4) \\ out \end{array} \right.$$

$test(x) := x^3 = x^3$

$test(4) = 64$

$f(test(X)) = 64$ <- note the uppercase X; because x is defined in the canvas, using test(x) I'd pass a number as function's argument

[3.11]

$$f(g(1)) := \left| \begin{array}{l} out := 2 \cdot g(3) \\ out \end{array} \right.$$

<- you can define the type of input function by defining the number of the arguments required for that function

$f(test(Z)) = 54$

[3.12]

$$f(g(2)) := \left| \begin{array}{l} out := 2 \cdot g(2, 3) \\ out \end{array} \right.$$

<- 2-args function required as input

$test(x, y) := 2 \cdot x + 3 \cdot y = 2 \cdot x + 3 \cdot y$

$test(2, 3) = 13$

$f(test(X, Y)) = 26$

Can I pass a dependent variable as function argument? Yes, you can, providing that the function's argument is declared as a function with 0 arguments.

[3.13]

```
f(x(0)):= | m:= 5
          | out:= 2·x
          | out
```

<- 0-args function required as input; note that here you have to know the names of the indipendent variables behind the dependent variable, if you want to define them from inside the function

```
k:= 3·m = 3·m
f(k) = 30
```

<- 2*3*5=30

```
f(3) = 6
```

<- numbers are allowed as input

There is also the possibility to point out dummy arguments (dummy argument = something not used in the function)

[3.14]

```
f(▮):= | out:= 2·a
       | out
```

<- you may use a variable and then leave it unused, however this point out immediatly that everything in the argument will be ignored in the function (you can use even an unit)

```
a = 2
f("") = 4
f(▮) = ▮
```

<- you still need to add a value to evaluate the function

```
lastError= "Fill in all empty elements."
a:= 3
f(1) = 6
f(0) = 6
```

Like for variables, it is possible to use functions already defined in the worksheet, or define locally other functions (nesting)

[3.15]

```
g(x):= | out:= 2·x
       | out

f(x):= | out:= 3·g(x)
       | out

g(4) = 8
f(4) = 24
```

[3.16]

```
f(x):= | g(y):= | out:= 5·y
       |         | out
       | out:= 3·g(x)
       | out

f(4) = 60
```

PROS of nesting functions:
- you have g(y) right here;
- you may keep a different function with the same name on the canvas, without interferences;
- you may change the function locally several times (every time you need it)
- if you don't need the nested function elsewhere, there's no need to have it on the canvas;

CONS:
- if g(y) is used elsewhere, you have to define it again;
- every time you call f(x), time will be used to define again g(y);
- local functions aren't available in the dynamic assistant;
- (cosmetic) in some cases, the size of the function may become too big for the page layout;

The function itself can be used inside it to make recursive functions

[3.17]

```
f(x):= | out:= 2·(x+1)
       | if x<10
       |   f(out)
       | else
       |   out
```

WARNING There's a know bug related to the control of recursive functions when they falls in infinite loops, that ends in an unrecoverable crash of the worksheet;
check carefully the logic of your function
and save the worksheet before the evaluation
(disable auto calculation just-in-case)

$f(1)=22$ <-- f.e. here it fails if you use a value < -1

As for inline functions, you can use vectorize() on your function or inside your function

[3.18]

```
f(x):= | out:= ln(1+|x|)+1
       | out
```

$$M:=\begin{bmatrix} 2 & 3 & 4 \\ -2 & -3 & -4 \end{bmatrix}$$

$f(2)=2.0986$

$$f(M)=\begin{bmatrix} 2.0986 & 2.3863 & 2.6094 \\ 2.0986 & 2.3863 & 2.6094 \end{bmatrix}$$

[3.19]

```
f(x):= | out:= ln(1+|x|)+1
       | out
```

$f(2)=2.0986$

$$\overrightarrow{f(M)}=\begin{bmatrix} 2.0986 & 2.3863 & 2.6094 \\ 2.0986 & 2.3863 & 2.6094 \end{bmatrix}$$

Absolute definitions

This feature send the value to the top of the worksheet, making it available before the definition. To create an absolute definitions you have to type the TILDE (~) character before the name.

The value sent at the top of the worksheet will be available at the next recalculation.
-> This features makes available values through worksheet's recalculations.

Pay attention to this, because actions on the worksheet may trigger partial evaluations of the worksheet, making results different from what you may think.

[4.1] Number of evaluation performed in the Worksheet

```
someVar = 1                          <- the variable is available above the definition
                                        (press F9)
if ¬ IsDefined(someVar)
   someVar := 0
else                                 <- absolute definition -> value will be sent to the begin
   someVar := someVar + 1               of the worksheet at the next recalculation (press F9)

someVar = 2                          <- the variable is updated by the worksheet flow
```

This feature is available also for functions

[4.2]

```
f(6) = 15                            <- the function is available above the definition
                                        (press F9)
f(x) := | out := 2·x + 3             <- absolute function
        | out

f(6) = 15
```

Here there are some examples of how something simple can be made in several ways.
Namely we want to know Area, Perimeter and Centroid (refered to the bottom-left corner) of a rectangle, given his base and height.

requires following values:
- b: base of rectangle
- h: height of rectangle

result:
- a system containing: centroid, perimeter, area

RectangleProperties_1(b, h) :=
if (b > 0) ∧ (h > 0)
  A := b · h
  P := 2 · (b + h)
  C := [b/2; h/2]
else
  A := (P := (C := "-"))
{C; P; A}

<- chained definition; some alternatives:

A := "-"
P := "-"
C := "-"

[A := "-" P := "-" C := "-"]

↑NOTE: behavior of definitions inside matrices changed since SMath Studio 0.98

RP := RectangleProperties_1(10 cm, 5 cm) = {[0.05 m; 0.025 m]; 0.3 m; 0.005 m^2}

c := RP_1 = [5; 2.5] cm p := RP_2 = 30 cm a := RP_3 = 50 cm^2

RP := RectangleProperties_1((- 10) cm, 5 cm) = {"-"; "-"; "-"}

c := RP_1 = "-" p := RP_2 = "-" a := RP_3 = "-"

```
requires following values:
- b: base of rectangle
- h: height of rectangle
- out: a variable to store a system containing: centroid, perimeter, area
result:
- error message
```

RectangleProperties_2(b , h , out) :=

```
if (b > 0) ∧ (h > 0)
    A := b · h
    P := 2 · (b + h)
    C := [b/2; h/2]
    msg := "done"
else
    A := (P := (C := "-"))
    msg := "wrong dimensions"
out := {C; P; A}
msg
```

RectangleProperties_2(10 *cm* , 5 *cm* , res) = "done"

c := res $_1$ = $\begin{bmatrix} 5 \\ 2.5 \end{bmatrix}$ *cm* p := res $_2$ = 30 *cm* a := res $_3$ = 50 cm^2

RectangleProperties_2((− 10) *cm* , 5 *cm* , res) = "wrong dimensions"

c := res $_1$ = "-" p := res $_2$ = "-" a := res $_3$ = "-"

```
requires following values:
- b: base of rectangle
- h: height of rectangle
- C: a variable to store the Centroid
- P: a variable to store the Perimeter
- A: a variable to store the Area
result:
- error message
```

RectangleProperties_3(b , h , C , P , A) :=

```
if (b > 0) ∧ (h > 0)
    A := b · h
    P := 2 · (b + h)
    C := [b/2; h/2]
    msg := "done"
else
    A := (P := (C := "-"))
    msg := "wrong dimensions"
msg
```

RectangleProperties_3(20 *cm* , 10 *cm* , c , p , a) = "done"

$c = \begin{bmatrix} 10 \\ 5 \end{bmatrix}$ *cm* $p = 60$ *cm* $a = 200$ *cm*^{2}

RectangleProperties_3((−20) *cm* , 10 *cm* , c , p , a) = "wrong dimensions"

c = "-" *cm* p = "-" *cm* a = "-" *cm*^{2}

```
requires following values:
- b: base of rectangle, from outside the function
- h: height of rectangle, from outside the function
- C: a variable to store the Centroid
- P: a variable to store the Perimeter
- A: a variable to store the Area
result:
- error message
```

RectangleProperties_4(C , P , A) :=
if (b > 0) ∧ (h > 0)
  A := b · h
  P := 2 · (b + h)
  C := [b/2; h/2]
  msg := "done"
else
  A := (P := (C := "-"))
  msg := "wrong dimensions"
msg

b := 10 cm

h := 20 cm

RectangleProperties_4(c , p , a) = "done"

$c = \begin{bmatrix} 5 \\ 10 \end{bmatrix}$ cm p = 60 cm $a = 200\ cm^2$

b := (− 10) cm

h := 20 cm

RectangleProperties_4(c , p , a) = "wrong dimensions"

c = "-" cm p = "-" cm $a = "-"\ cm^2$

requires following values:
- b: base of rectangle, from outside the function
- h: height of rectangle, from outside the function

result:
- a system containing: centroid, perimeter, area

RectangleProperties_5(b, h) :=
if (b > 0) ∧ (h > 0)
  A := b · h
  P := 2 · (b + h)
  C := [b/2; h/2]
else
  A := (P := (C := "-"))
{C; P; A}

b := 10 cm

h := 20 cm

RP := RectangleProperties_5("") = {[0.05 m; 0.1 m]; 0.6 m; 0.02 m^2}

c := RP_1 = [5; 10] cm p := RP_2 = 60 cm a := RP_3 = 200 cm^2

b := (−10) cm

h := 20 cm

RP := RectangleProperties_5("") = {"-"; "-"; "-"}

c := RP_1 = "-" p := RP_2 = "-" a := RP_3 = "-"

$$DB := \begin{bmatrix} "b" & "h" \\ 10 & 20 \\ 20 & 10 \\ 5 & 40 \\ 40 & 5 \\ 25 & 25 \end{bmatrix}$$

<- input database

output databases ↴

out1:=(out2:=(out3:=(out4:=(out5:=["C" "P" "A"]))))

```
for j ∈ 2 .. rows(DB)
    b := DB[j,1]
    h := DB[j,2]
    "example 01"
    R1 := RectangleProperties_1(b, h)
    out1 := stack(out1, [R1_1 R1_2 R1_3])
    "example 02"
    RectangleProperties_2(b, h, R2)
    out2 := stack(out2, [R2_1 R2_2 R2_3])
    "example 03"
    RectangleProperties_3(b, h, c3, p3, a3)
    out3 := stack(out3, [c3 p3 a3])
    "example 04"
    RectangleProperties_4(c4, p4, a4)
    out4 := stack(out4, [c4 p4 a4])
    "example 05"
    R5 := RectangleProperties_5()
    out5 := stack(out5, [R5_1 R5_2 R5_3])
```

IMPORTANT

<- NOTE: functions inside procedures can be evaluated without assignment/evaluation operators

$$DB = \begin{bmatrix} "b" & "h" \\ 10 & 20 \\ 20 & 10 \\ 5 & 40 \\ 40 & 5 \\ 25 & 25 \end{bmatrix}$$

$$out1 = \begin{bmatrix} "C" & "P" & "A" \\ \begin{bmatrix} 5 \\ 10 \end{bmatrix} & 60 & 200 \\ \begin{bmatrix} 10 \\ 5 \end{bmatrix} & 60 & 200 \\ \begin{bmatrix} 2.5 \\ 20 \end{bmatrix} & 90 & 200 \\ \begin{bmatrix} 20 \\ 2.5 \end{bmatrix} & 90 & 200 \\ \begin{bmatrix} 12.5 \\ 12.5 \end{bmatrix} & 100 & 625 \end{bmatrix}$$

$$out2 = \begin{bmatrix} "C" & "P" & "A" \\ \begin{bmatrix} 5 \\ 10 \end{bmatrix} & 60 & 200 \\ \begin{bmatrix} 10 \\ 5 \end{bmatrix} & 60 & 200 \\ \begin{bmatrix} 2.5 \\ 20 \end{bmatrix} & 90 & 200 \\ \begin{bmatrix} 20 \\ 2.5 \end{bmatrix} & 90 & 200 \\ \begin{bmatrix} 12.5 \\ 12.5 \end{bmatrix} & 100 & 625 \end{bmatrix}$$

$$out3 = \begin{bmatrix} "C" & "P" & "A" \\ \begin{bmatrix} 5 \\ 10 \end{bmatrix} & 60 & 200 \\ \begin{bmatrix} 10 \\ 5 \end{bmatrix} & 60 & 200 \\ \begin{bmatrix} 2.5 \\ 20 \end{bmatrix} & 90 & 200 \\ \begin{bmatrix} 20 \\ 2.5 \end{bmatrix} & 90 & 200 \\ \begin{bmatrix} 12.5 \\ 12.5 \end{bmatrix} & 100 & 625 \end{bmatrix}$$

$$out4 = \begin{bmatrix} "C" & "P" & "A" \\ \begin{bmatrix} 5 \\ 10 \end{bmatrix} & 60 & 200 \\ \begin{bmatrix} 10 \\ 5 \end{bmatrix} & 60 & 200 \\ \begin{bmatrix} 2.5 \\ 20 \end{bmatrix} & 90 & 200 \\ \begin{bmatrix} 20 \\ 2.5 \end{bmatrix} & 90 & 200 \\ \begin{bmatrix} 12.5 \\ 12.5 \end{bmatrix} & 100 & 625 \end{bmatrix}$$

$$out5 = \begin{bmatrix} "C" & "P" & "A" \\ \begin{bmatrix} 5 \\ 10 \end{bmatrix} & 60 & 200 \\ \begin{bmatrix} 10 \\ 5 \end{bmatrix} & 60 & 200 \\ \begin{bmatrix} 2.5 \\ 20 \end{bmatrix} & 90 & 200 \\ \begin{bmatrix} 20 \\ 2.5 \end{bmatrix} & 90 & 200 \\ \begin{bmatrix} 12.5 \\ 12.5 \end{bmatrix} & 100 & 625 \end{bmatrix}$$