

USER-DEFINED FUNCTIONS IN SMATH STUDIO

rev.06 / 2016.03.21

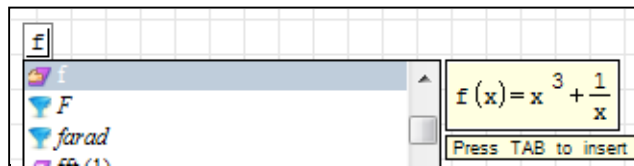
Inline functions

To define a function, type the function 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):= x3 + 1/x
```

```
f(2)=8.5
```

<- type f and look at the dynamic assistance, the function is stored as f; this means there aren't other functions with the same name



Like variables, functions are case sensitive, hence $f(x) \neq F(x)$

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

```
f(2)=8.5
```

```
F(2)=5
```

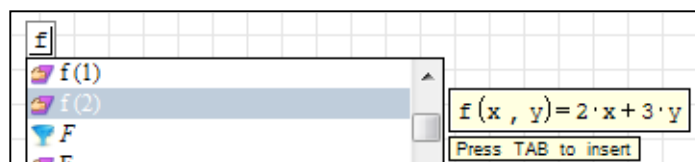
It is possible to define several functions with the same name, if they have a different number of arguments

```
[1.3] f(x, y):= 2·x + 3·y = 2·x + 3·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 needed by each function



The argument name can be only: a variable, a function, an unit or a math string. 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: $()[]\{\}+/-/*:=<>\leq\geq\neq\&|\mp\pm\dots$

IMPORTANT

```
[1.4]
```

```
[A] f(x3):= x3 + 4
```

```
f(2)=6
```

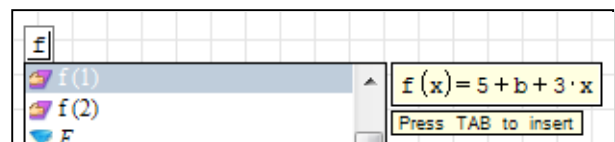
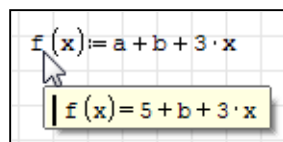
<- a variable name can contain numbers, but not as 1st character

- [B] $f(_3) := _3 + 5$ $f(2) = 7$ \leftarrow any non-number character is allowed as first character
- [C] $f(\#3) := \#3 + 6$ $f(2) = 8$ \leftarrow # is the character of the empty placeholder, to use it you have to type at least the character that will be the 2nd character, move back the cursor, and then you can type # in front of it
- [D] $f(m) := 7 + 8 \cdot m$ $f(2) = 23$ $f(2 \cdot m) = 7 + 16 \cdot m$ \leftarrow if you use an unit as argument name, it will be treated as any standard variable.
 \leftarrow no conflicts with the input values
- [E] $f("") := x + 9$ $f(2) = 9 + x$ \leftarrow dummy argument (argument not used in the function)
- [F] $f(2) := a + b + 3 \cdot i$ \leftarrow you cannot use numbers as argument name
 $f(1.5) := a + b + 3 \cdot i$ $\text{lastError} = \text{"Syntax is incorrect."}$
 if your intent is to define the value of a function in a point, use an if statement, booleans or other functions
- $$f(x) := \begin{cases} 10 & \text{if } x = 2 \\ 2 \cdot x & \text{else} \end{cases} \quad g(x) := (x = 2) \cdot (10) + (x \neq 2) \cdot (2 \cdot x)$$
- $$\begin{array}{ll} f(1.9) = 3.8 & g(1.9) = 3.8 \\ f(2) = 10 & g(2) = 10 \\ f(2.1) = 4.2 & g(2.1) = 4.2 \end{array}$$
- [G] $f(i) := a + b + 3 \cdot i$ \leftarrow i, like a number, cannot be used as argument name (you can use the names of other constants such as π or e)
 $\text{lastError} = \text{"Syntax is incorrect."}$
- [H] $f(g(x)) := 2 \cdot g(0) + 3 \cdot g\left(\frac{\pi}{5}\right)$ \leftarrow a function is a valid argument
- $$f(\sin(x)) = 1.7634 \quad 2 \cdot \sin(0) + 3 \cdot \sin\left(\frac{\pi}{5}\right) = 1.7634$$
- $$f(\exp(x)) = 7.6234 \quad 2 \cdot \exp(0) + 3 \cdot \exp\left(\frac{\pi}{5}\right) = 7.6234$$
- $$f(F(x)) = 16.885 \quad 2 \cdot F(0) + 3 \cdot F\left(\frac{\pi}{5}\right) = 16.885$$

You can access variables from the canvas (not in the arguments), defined or not yet defined

- [1.5] $a := 5$
 $f(x) := a + b + 3 \cdot x$ \leftarrow 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] $b := 3$
 $f(2) = 14$

<- when a canvas variable is not defined at function's definition, it is kept as symbolical link (the name is stored); hence if you change his value the function will returns a different result

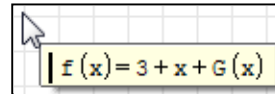
IMPORTANT

[C] $a := 10$
 $f(2) = 14$

<- when a canvas variable is defined at function's definition, the value is stored; hence changes on variable's value doesn't affects function's results when executed.

[1.6] $f(x) := F(x) + G(x)$
 $f(1) = 4 + G(1)$

<- this principle applies also to functions used in your function



$$f(x) = 3 + x + G(x)$$

[A] $F(x) := 5$
 $f(1) = 4 + G(1)$

<- F(x) was stored as it was at f(x) definition, thus when you change it the new content 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 when you change it the new content is used in f(x)

You can use `vectorize()` on your function or inside your function

[SMath Studio ≥ 0.98]

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

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

$f(M) = \blacksquare$

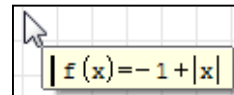
`lastError= "Argument must be scalar."`

$$\overrightarrow{f(M)} = \begin{pmatrix} 0 & 1 & 2 \\ 0 & 1 & 2 \end{pmatrix}$$

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

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

<- hold the mouse over and look at how it is stored



$$f(x) = -1 + |x|$$

$f(M) = \blacksquare$

`lastError= "Argument must be scalar."`

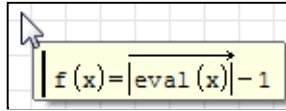
what happens here? `vectorize()` is a generic function that may be applied both on numerical and symbolical data -> the evaluation in place of `vectorize` done to store $f(x)$ returns $|x|$ because the argument is a generic unknown (like a scalar)

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

[1.9]

$$f(x) := \overrightarrow{\text{eval}(x)} - 1$$

<- hold the mouse over and look at how it is stored



$$f(x) = \overrightarrow{\text{eval}(x)} - 1$$

$$f(M) = \begin{pmatrix} 0 & 1 & 2 \\ 0 & 1 & 2 \end{pmatrix}$$

the evaluation of vectorize done to store $f(x)$ can't returns a result -> vectorize is stored in the function and will be evaluated once called

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

Dependent variables

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

[2.1] $y := 2 \cdot x + 3$

$$\dot{x} := \frac{d}{dt} x(t)$$

[A] $x := 2$

$$y = 7$$

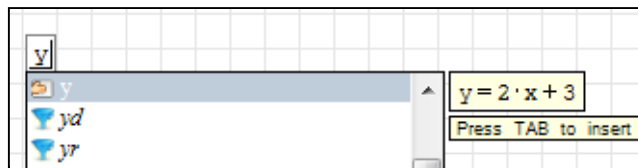
[B] $x := 3$

$$y = 9$$

[C] $\frac{d}{dx} y = 2$

<- even if x is defined at the execution, y keeps the symbolical link (look at the dynamic assistance and [1.4][B])

$$\int_0^2 y \, dx = 10$$



Like in functions, it is possible to use defined/undefined variables (as well as functions) from the canvas

[2.2] $b := 5$

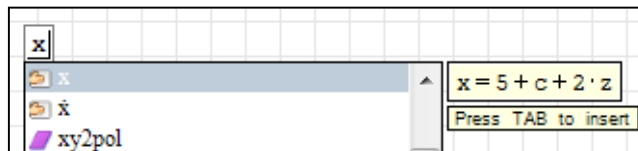
$x := 2 \cdot z + b + c = 5 + c + 2 \cdot z$ <- hold the mouse over the variable to see how "b" and "c" are stored

[A] $c := 2$

$$x = 7 + 2 \cdot z$$

[B] $c := 4$

$$x = 9 + 2 \cdot z$$



[2.3] $x := b + d + G(z)$

$$z := 3 \quad d := 3$$

$$x = 17$$

$$d := 2$$

$$x = 16$$

$$z := 2$$

$$x = 13$$

$$b := 3$$

$$x = 13$$

Programming functions

To make complex tasks using functions, it is possible to use the `line()` function in the RHS (Right Hand Side) of a function.

`line()` is available from the "Programming" toolbox on the right side of the screen

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

[see: line\(\)](#)

```
[3.1]  f(x):= | a:= 3·x+4  <- x is an argument of f
          | a             <- 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):= | x:= x+1  <- assignment operator applied to an input argument
          | out:= x^2
          | out
```

```
[A]    a:= 2
        f(a)= 9  <- result of the function
        a= 3     <- Pass by reference side effect
```

```
[B]    a:= 1
        while a<10  Note that the syntax is very compact
          b:= f(a)
        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):=

xt:= x

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

xt

a:= 2

f(a)= 6.7071

a= 2

<- No side effects (no assignment used on the function arguments)

Local unknowns and local variables

[3.5] $f(x) := \begin{cases} \text{out} := x + 3 \cdot z \\ \text{out} \end{cases}$ <- z is an unknown in the function (not defined anywhere locally) and even in the canvas

$z = 2$ $\text{lastError} = \text{"Argument must be scalar."}$

[A] $a := 2$
 $f(a) = 8$ <- The local z now is exposed to the canvas

[B] $z := 6$ <- now z is defined on the canvas
 $f(a) = 20$ <- symbolically the output should be $f(2) = 2 + 3 \cdot z$
 z exists in the canvas -> $f(2) = 2 + 3 \cdot 6 = 20$

[C] $z := 7$
 $f(a) = 23$ <- symbolically the output should be $f(2) = 2 + 3 \cdot z$
 z exists in the canvas -> $f(2) = 2 + 3 \cdot 7 = 23$

What if a local unknown exists in the canvas at function definition? Nothing changes, since the function is not executed, a local unknown still remains an unknown, hence the name is stored (not the value)

[3.6] $c := 5$
 $f(x) := \begin{cases} \text{out} := x + 3 \cdot c \\ \text{out} \end{cases}$ <- c is an unknown in the function (not defined anywhere locally)

[A] $a := 2$
 $c := 3$
 $f(a) = 11$ <- symbolically the output should be $f(2) = 2 + 3 \cdot c$
 c exists in the canvas -> $f(2) = 2 + 3 \cdot 3 = 11$

[B] $c := 4$
 $f(a) = 14$ <- symbolically the output should be $f(2) = 2 + 3 \cdot c$
 c exists in the canvas -> $f(2) = 2 + 3 \cdot 4 = 14$

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

[3.7] $f(x) := \begin{cases} \text{out} := x + 3 \cdot k \\ k := 5 \\ \text{out} := \text{out} + k \\ \text{out} \end{cases}$ <- here k is unknown in the function => stored as name
 <- from this point k is known in the function (a local variable)

[A] $f(2) = 22$ <- $2 + 3 \cdot 5 + 5 = 22$
 $k = \blacksquare$ $\text{lastError} = \text{"k - not defined."}$

IMPORTANT

[B] k:= 4

f(2)=19

<- (2+3*4)+5=19

What's happening during the execution?

In the first out's assignment, k is not defined -> when executed a value is found in the canvas and it is stored in out; in the 2nd assignment of out k is no more a local unknown, thus the new value is used for any further use of k.

k= 4

<- you can provide side effects on the canvas only applying assignments to the input arguments

>>> If you use LOCAL UNKNOWNNS inside functions and you DON'T want replacements from outside, is better you use unusual names for those local unknowns (such as _x, _x_, #x, x#, etc...)

>>> This applies also when you use 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 (intentionally or unintentionally); if you need to avoid this behavior -> define the variable before using it (f.e. you can use M:0 where M will be your matrix - see [3.8])

[3.8] $f(x) := \begin{cases} M_{1\ 1} := x \\ M_{1\ 2} := 2 \cdot x \\ M \end{cases}$

$g(x) := \begin{cases} M := 0 \\ M_{1\ 1} := x \\ M_{1\ 2} := 2 \cdot x \\ M \end{cases}$

M:=(1 1)

f(2)=(2 4)

g(2)=(2 4)

M:=(1 1 1)

f(2)=(2 4 1)

g(2)=(2 4)

M:= $\begin{pmatrix} 1 \\ 1 \end{pmatrix}$ f(2)= $\begin{pmatrix} 2 & 4 \\ 1 & 0 \end{pmatrix}$

g(2)=(2 4)

[3.9] $f(x) := \begin{cases} \text{out} := x + 3 \cdot k \\ k := 2 \cdot (k + 1) \\ \text{out} := \text{out} + k \\ \text{out} \end{cases}$

<- k is a local unknown: if available, canvas' values will be used
<- a local variable is defined using a local unknown

k= 4

f(2)=24

<- (2+3*4)+(2*(4+1))=24

k= 4

<- you can use safely assignment operators on canvas variables not passed by function's arguments

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

[3.10] $f(g(x)) := \begin{cases} \text{out} := g(4) \\ \text{out} \end{cases}$

$\text{test}(x) := x^3 = x^3$

$\text{test}(4) = 64$

$f(\text{test}(x)) = 64$

<- 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] $f(g(1)) := \begin{cases} \text{out} := g(3) \\ \text{out} \end{cases}$

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

$f(\text{test}(z)) = 27$

[3.12] $f(g(2)) := \begin{cases} \text{out} := g(2, 3) \\ \text{out} \end{cases}$

<- 2-args function required as input

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

$\text{test}(2, 3) = 13$

$f(\text{test}(x, y)) = 13$

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] $f(x(0)) := \begin{cases} m := 5 \\ \text{out} := 2 \cdot x \\ \text{out} \end{cases}$

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

$k := 3 \cdot m = 3 \cdot m$

$f(k) = 30$

<- $2 \cdot 3 \cdot 5 = 30$

$f(3) = 6$

<- 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] $f("") := \begin{cases} \text{out} := 2 \cdot x \\ \text{out} \end{cases}$

<- 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

$x = 5$

$f(0) = 10$

<- x was defined before declaring [3.13], thus the use of a dependent variable $f=2 \cdot x$ wouldn't be possible.

$x := 3$

$f(1) = 6$

$f(\text{"something"}) = 6$

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

[3.15] $g(x) := \begin{cases} \text{out} := 2 \cdot x \\ \text{out} \end{cases}$

$f(x) := \begin{cases} \text{out} := 3 \cdot g(x) \\ \text{out} \end{cases}$

$g(4) = 8$

$f(4) = 24$

[3.16] $f(x) := \begin{cases} g(y) := \begin{cases} \text{out} := 5 \cdot y \\ \text{out} \end{cases} \\ \text{out} := 3 \cdot g(x) \\ \text{out} \end{cases}$
 $f(4) = 60$

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 are not 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] $f(x) := \begin{cases} \text{out} := 2 \cdot (x+1) \\ \text{if } x < 10 \\ \quad f(\text{out}) \\ \text{else} \\ \quad \text{out} \end{cases}$

WARNING

There's a known bug related to the control of recursive functions when they fall 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.

$f(1) = 22$

<-- 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

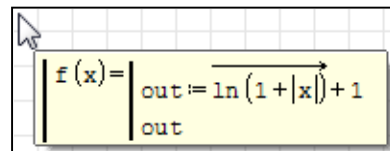
[3.18] $f(x) := \begin{cases} \text{out} := \overrightarrow{\ln(1+|x|)} + 1 \\ \text{out} \end{cases}$

<- you can't fall in the issue noticed in [1.7], because no evaluation is made here (at function's definition)

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

$f(2) = 2.0986$

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



[3.19] $f(x) := \begin{cases} \text{out} := \ln(1+|x|) + 1 \\ \text{out} \end{cases}$

$f(2) = 2.0986$

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

Inside a function you can rename a variable at once using the F8 key

[3.20] $f(x) := \begin{cases} a := x + 3 \\ a := a^2 + 2 \\ a \end{cases}$

<- move the cursor above a variable (x or a); you should see all the occurrences of that item light-grayed

$f(x) := \begin{cases} \underline{a} := x + 3 \\ \underline{a} := \underline{a}^2 + 2 \\ \underline{a} \end{cases}$

press the F8 key of your keyboard, a cursor should appear in each of them

$f(x) := \begin{cases} \underline{a} := x + 3 \\ \underline{a} := \underline{a}^2 + 2 \\ \underline{a} \end{cases}$

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

$f(x) := \begin{cases} \underline{} := x + 3 \\ \underline{} := \underline{}^2 + 2 \\ \underline{} \end{cases}$

$f(x) := \begin{cases} \underline{y} := x + 3 \\ \underline{y} := \underline{y}^2 + 2 \\ \underline{y} \end{cases}$

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

$f(x) := \begin{cases} y := x + 3 \\ y := y^2 + 2 \\ y \end{cases}$

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

someVar := **someVar** + 1

<- absolute definition -> value will be sent to the begin of the worksheet at the next recalculation (press F9)

someVar = 0

<- the variable is updated by the worksheet flow

This feature is available also for functions

[4.2] **f(6) = 1** <- the function is available above the definition (press F9)

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

<- absolute function

f(6) = 15

TIPS:

- if you want to avoid side effects on the canvas:
 - 1) don't use assignments on input arguments
 - 2) you may use local variables with the names of canvas variables (see [3.9])
- if you are running NUMERICAL calculations inside the function:
 - if you don't want/need values from outside, you have to check that everything is defined inside the function ("pure" function -> copy the function in an empty worksheet, run it directly -> shouldn't run if something is missing in the function)
 - if you use unknowns to get values from outside the function:
 - 1) document somewhere the dependancies (f.e. in the description)
 - 2) test it in an empty worksheet, define the dependencies and check the consequences of the loss of any single dependancy (A,B,C -> keep A and B, remove C, then keep A and C and remove B, etc..)
- if you are running SYMBOLICAL calculations inside the function, you may need to choose hardly predictable unknowns' names (such as $x\#$, $_x$, $_x_\$, $\#x$, $\$x$, ...) to avoid unintentional replacements from the canvas;
- be careful mixing features such as "dynamic arrays" and programming-functions

Here there are some examples of how something simple can be made in several ways. Namely we want to know Area, Perimeter and Centroid (referred 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:=  $\begin{pmatrix} \frac{b}{2} \\ \frac{h}{2} \end{pmatrix}$ 
  else
    A:=(P:=(C:=" - "))
    { C
    { P
    { A
```

<- chained definition; some alternatives:

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

```
(A:=" - " P:=" - " C:=" - ")
```

↑NOTE: behavior of definitions inside matrices changes since SMATH Studio 0.98

```
RP:=RectangleProperties_1(10 cm, 5 cm)= $\begin{pmatrix} 0.05 m \\ 0.025 m \\ 0.3 m \\ 0.005 m^2 \end{pmatrix}$ 
```

```
c:=RP_1= $\begin{pmatrix} 5 \\ 2.5 \end{pmatrix} cm$       p:=RP_2=30 cm      a:=RP_3=50 cm2
```

```
RP:=RectangleProperties_1(10 cm, 5 cm)= $\begin{pmatrix} " - " \\ " - " \\ " - " \end{pmatrix}$ 
```

```
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:=  $\begin{pmatrix} \frac{b}{2} \\ \frac{h}{2} \end{pmatrix}$ 
    | msg:= "done"
else
    | A:=(P:=(C:="-"))
    | msg:= "wrong dimensions"
out:= { C
      | P
      | A
msg
```

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

$$c := res_1 = \begin{pmatrix} 5 \\ 2.5 \end{pmatrix} \text{ cm} \quad p := res_2 = 30 \text{ cm} \quad a := res_3 = 50 \text{ cm}^2$$

```
RectangleProperties_2(-10 cm, 5 cm, res)= "wrong dimensions"
```

$$c := res_1 = "-" \quad p := res_2 = "-" \quad a := res_3 = "-"$$

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 :=  $\begin{pmatrix} \frac{b}{2} \\ \frac{h}{2} \end{pmatrix}$ 
    msg := "done"
else
    A := (P := (C := "-"))
    msg := "wrong dimensions"
msg
```

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

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

```
RectangleProperties_3((-20) cm, 10 cm, c, p, a) = "wrong dimensions"
```

$$c = "-" \text{ cm} \qquad p = "-" \text{ cm} \qquad a = "-" \text{ cm}^2$$

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:=  $\begin{pmatrix} \frac{b}{2} \\ \frac{h}{2} \end{pmatrix}$ 
    | msg:= "done"
else
    | A:=(P:=(C:="-"))
    | msg:= "wrong dimensions"
msg
```

b:= 10 cm

h:= 20 cm

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

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

b:=(-10) cm

h:= 20 cm

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

$$c = \text{"-"} \text{ cm} \qquad p = \text{"-"} \text{ cm} \qquad a = \text{"-"} \text{ cm}^2$$

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(" ") := if (b > 0) ∧ (h > 0)
    A := b · h
    P := 2 · (b + h)
    C :=  $\begin{pmatrix} \frac{b}{2} \\ \frac{h}{2} \end{pmatrix}$ 
else
    A := (P := (C := " - "))
    { C
    } P
    { A
```

b := 10 *cm*

h := 20 *cm*

RP := RectangleProperties_5() = $\begin{cases} \begin{pmatrix} 0.05 \text{ m} \\ 0.1 \text{ m} \end{pmatrix} \\ 0.6 \text{ m} \\ 0.02 \text{ m}^2 \end{cases}$

c := RP₁ = $\begin{pmatrix} 5 \\ 10 \end{pmatrix}$ *cm* p := RP₂ = 60 *cm* a := RP₃ = 200 *cm*²

b := (-10) *cm*

h := 20 *cm*

RP := RectangleProperties_5() = $\begin{cases} " - " \\ " - " \\ " - " \end{cases}$

c := RP₁ = " - " p := RP₂ = " - " a := RP₃ = " - "

```

DB:=
  ⎡ "b" "h" ⎤
  ⎢ 10 20 ⎥
  ⎢ 20 10 ⎥
  ⎢ 5 40 ⎥
  ⎢ 40 5 ⎥
  ⎢ 25 25 ⎥
  <- 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(b, p4, a4)
```

```
  out4:= stack(out4, (c4 p4 a4))
```

```
  "example 05"
```

```
  R5:= RectangleProperties_5(b)
```

```
  out5:= stack(out5, (R5_1 R5_2 R5_3))
```

IMPORTANT

<- NOTE: in the canvas this is not possible, to trigger a result you must assign the user-defined function to a variable [:] or evaluate it [= or CTRL+.]
When you nest functions you can evaluate them without assignment/evaluation operators

```

DB=
  ⎡ "b" "h" ⎤
  ⎢ 10 20 ⎥
  ⎢ 20 10 ⎥
  ⎢ 5 40 ⎥
  ⎢ 40 5 ⎥
  ⎢ 25 25 ⎥
  out1=
    ⎡ "C" "P" "A" ⎤
    ⎢ 5 ⎥
    ⎢ 10 ⎥ 60 200
    ⎢ 10 ⎥
    ⎢ 5 ⎥ 60 200
    ⎢ 2.5 ⎥
    ⎢ 20 ⎥ 90 200
    ⎢ 20 ⎥
    ⎢ 2.5 ⎥ 90 200
    ⎢ 12.5 ⎥
    ⎢ 12.5 ⎥ 100 625
  out2=
    ⎡ "C" "P" "A" ⎤
    ⎢ 5 ⎥
    ⎢ 10 ⎥ 60 200
    ⎢ 10 ⎥
    ⎢ 5 ⎥ 60 200
    ⎢ 2.5 ⎥
    ⎢ 20 ⎥ 90 200
    ⎢ 20 ⎥
    ⎢ 2.5 ⎥ 90 200
    ⎢ 12.5 ⎥
    ⎢ 12.5 ⎥ 100 625
  out3=
    ⎡ "C" "P" "A" ⎤
    ⎢ 5 ⎥
    ⎢ 10 ⎥ 60 200
    ⎢ 10 ⎥
    ⎢ 5 ⎥ 60 200
    ⎢ 2.5 ⎥
    ⎢ 20 ⎥ 90 200
    ⎢ 20 ⎥
    ⎢ 2.5 ⎥ 90 200
    ⎢ 12.5 ⎥
    ⎢ 12.5 ⎥ 100 625

```

```

  out4=
    ⎡ "C" "P" "A" ⎤
    ⎢ 5 ⎥
    ⎢ 10 ⎥ 60 200
    ⎢ 10 ⎥
    ⎢ 5 ⎥ 60 200
    ⎢ 2.5 ⎥
    ⎢ 20 ⎥ 90 200
    ⎢ 20 ⎥
    ⎢ 2.5 ⎥ 90 200
    ⎢ 12.5 ⎥
    ⎢ 12.5 ⎥ 100 625
  out5=
    ⎡ "C" "P" "A" ⎤
    ⎢ 5 ⎥
    ⎢ 10 ⎥ 60 200
    ⎢ 10 ⎥
    ⎢ 5 ⎥ 60 200
    ⎢ 2.5 ⎥
    ⎢ 20 ⎥ 90 200
    ⎢ 20 ⎥
    ⎢ 2.5 ⎥ 90 200
    ⎢ 12.5 ⎥
    ⎢ 12.5 ⎥ 100 625

```