▶ Computer Algebra using Maple Part II: Analysis with Maple; basic programming

Winfried Auzinger, Kevin Sturm (SS 2019)

1 Limits and series

```
> restart;
```

1.1 Limits

```
> infinity; # symbol for infinity
```

$$\infty$$

Limits of a simple sequence: limit, Limit

```
> limit(1+1/n,n=infinity);
```

$$1$$

Many commands have an **inert form**, leaving the expression unevaluated:

```
> Limit(1+1/n,n=infinity); # with capital first letter
```

$$\lim_{n \to \infty} \left(1 + \frac{1}{n}\right)$$

Or you may generate a nice formula like this:

```
> Limit((1+1/n)^n,n=infinity) = limit((1+1/n)^n,n=infinity);
```

$$\lim_{n \to \infty} \left(1 + \frac{1}{n}\right)^n = e$$

Variants of the limit-command:

- Limit of real functions:

```
> f := x -> (1+x)/(2+x);
```

$$f := x \mapsto \frac{x+1}{2+x}$$

```
> limit(f(x),x=0), limit(f(x),x=infinity);
```

$$\frac{1}{2}, 1$$

```
> limit(sin(x),x=infinity);
```

$$-1..1$$

- One-sided limits:

```
> limit(1/x,x=0,left), limit(1/x,x=0,right);
```

$$-\infty, \infty$$

1.2 Infinite series

sum can handle infinite series. Inert form: **Sum**

```
> Sum(1/k,k=1..infinity) = sum(1/k,k=1..infinity);
```

$$\sum_{k=1}^{\infty} \frac{1}{k} = \infty$$

```
> Sum(1/k^2,k=1..infinity) = sum(1/k^2,k=1..infinity);
```

$$\sum_{k=1}^{\infty} \frac{1}{k^2} = \frac{\pi^2}{6}$$

```
> Sum(1/k^3,k=1..infinity) = sum(1/k^3,k=1..infinity);
```

$$\sum_{k=1}^{\infty} \frac{1}{k^3} = \zeta(3)$$

Geometric series: Converges only for |q|<1.

```
> sum(q^k,k=0..infinity), sum(3^(-k),k=0..infinity)
```

$$\sum_{k=0}^{\infty} q^k, \frac{3}{2}$$

```
> sum(k*q^k,k=1..infinity), sum(k*3^(-k),k=1..infinity)
```

$$\sum_{k=1}^{\infty} k\, q^k, \frac{3}{4}$$

1.3 Infinite products

product can handle infinite products. Inert form: **Product**

```
> Product((1+1/k^2),k=1..infinity) =
  product((1+1/k^2),k=1..infinity);
```

$$\prod_{k=1}^{\infty} \left(1 + \frac{1}{k^2}\right) = \frac{\sinh(\pi)}{\pi}$$

2 Functions, differentiation, integration, ...

```
> restart;
```

2.1 More on functions

Functional composition can be denoted via **@**:

```
> (cos@ln)(x);
```

$$\cos(\ln(x))$$

```
> f := x->x^2; g := exp;
```

$$f := x \mapsto x^2$$
$$g := \exp$$

```
> h := f@g; h(x);
```

$$h := f@\exp$$
$$(e^x)^2$$

Another example (a bivariate function):

```
> f:=(x,y)->x*y; (exp@f)(u,v);
```

$$f := (x, y) \mapsto y\,x$$
$$e^{v\,u}$$

Functional powers are denoted by **@@**:

```
> (exp@@2)(1); # same like exp(exp(1))
  evalf(%);
```

$$\exp^{(2)}(1)$$
$$15.15426223$$

Generating a function from a previously computed result: Use **unapply** !!

First we try the following:

```
> f:=x->add(k^2*x^k,k=1..6);
```

$$f := x \mapsto add\left(k^2\,x^k,\, k = 1\,..6\right)$$

```
> f(y); # looks O.K.
```

$$36\,y^6 + 25\,y^5 + 16\,y^4 + 9\,y^3 + 4\,y^2 + y$$

We try to differentiate this using D (see **2.2** below):

```
> D(f)(y); # does not work
```

$$D(f)(y)$$

Now we use **unapply** :

```
> f := unapply(add(k^2*x^k,k=1..6),x):
> f(y), D(f)(y); # O.K.
```

$$36 y^6 + 25 y^5 + 16 y^4 + 9 y^3 + 4 y^2 + y, 216 y^5 + 125 y^4 + 64 y^3 + 27 y^2 + 8 y + 1$$

Here the function f generated using **unapply** *is based on the 'ready-cooked' formula. This is especially relevant when generating the expression for f involves a nontrivial, long computation.*

2.2 Derivatives

diff differentiates expressions. Inert form: **Diff**

```
> Diff(sin(x),x) = diff(sin(x),x);
```

$$\frac{d}{dx} \sin(x) = \cos(x)$$

```
> diff(sin(x),y);
```

$$0$$

```
> f := x -> exp(2*x)/cos(x)^3*sinh(3*x);
```

$$f := x \mapsto \frac{e^{2x} \sinh(3x)}{\cos(x)^3}$$

```
> diff(f(x),x);
```

$$\frac{2 e^{2x} \sinh(3x)}{\cos(x)^3} + \frac{3 e^{2x} \sinh(3x) \sin(x)}{\cos(x)^4} + \frac{3 e^{2x} \cosh(3x)}{\cos(x)^3}$$

```
> subs(x=0,diff(f(x),x)); # this is not automatically
  evaluated...
```

$$\frac{2 e^0 \sinh(0)}{\cos(0)^3} + \frac{3 e^0 \sinh(0) \sin(0)}{\cos(0)^4} + \frac{3 e^0 \cosh(0)}{\cos(0)^3}$$

```
> eval(%); # evaluate (or use simplify)
```

$$3$$

A partial derivative:

```
> Diff(x*y*cos(x-y),x) = diff(x*y*cos(x-y),x);
```

$$\frac{\partial}{\partial x} (x y \cos(x-y)) = y \cos(x-y) - x y \sin(x-y)$$

Higher [partial] derivatives:

```
> diff(x^2*y^3,x,y);
```

$$6 x y^2$$

```
> diff(exp(k*x),x,x), diff(exp(k*x),x,x,x);
```

$$k^2 e^{kx}, k^3 e^{kx}$$

Alternative syntax for higher derivatives:

```
> diff(ln(x),x$5);
```

$$\frac{24}{x^5}$$

Note: Generally, **$** serves as **repetition operator** for generating constant sequences:

```
> y$8;
```

$$y, y, y, y, y, y, y, y$$

D is the **derivative operator**. It maps **functions to functions**.

```
> D(sin);
```

$$\cos$$

```
> D(sin)(y), D(sin)(0);
```

$$\cos(y), 1$$

```
> f := u -> u^2*cos(u);
```

$$f := u \mapsto u^2 \cos(u)$$

```
> g := D(f);
```

$$g := u \mapsto 2\,u \cos(u) - u^2 \sin(u)$$

```
> g(u);
```

$$2\,u \cos(u) - u^2 \sin(u)$$

Functional power can be applied to D:

```
> (D@@0)(f); # this is identical with f
```

$$f$$

```
> (D@@2)(f); # apply second derivative operator
```

$$u \mapsto 2 \cos(u) - 4\,u \sin(u) - u^2 \cos(u)$$

Here, function F has not been specified:

```
> n := 7;
```

$$n := 7$$

```
> diff(F(x),x$n); (D@@n)(F)(x);
```

$$\frac{d^7}{dx^7} F(x)$$

$$D^{(7)}(F)(x)$$

2.3 Integrals

int integrates expressions. Inert form: **Int**

- Indefinite integrals (integration constant **not** added automatically):

```
> Int(sin(x*y),x) = int(sin(x*y),x);
```

$$\int \sin(x\,y)\,\mathrm{d}x = -\frac{\cos(x\,y)}{y}$$

```
> f := (u,k) -> u^k*exp(u);
```

$$f := (u, k) \mapsto u^k\,\mathrm{e}^u$$

```
> int(f(x,4),x) + C; # integration constant added manually
```

$$\left(x^4 - 4\,x^3 + 12\,x^2 - 24\,x + 24\right)\mathrm{e}^x + C$$

```
> int(f(x,m),x); # solution for general m
```

$$-(-1)^{-m}\left(x^m\,(-1)^m\,m\,\Gamma(m)\,(-x)^{-m} - x^m\,(-1)^m\,\mathrm{e}^x - x^m\,(-1)^m\,m\,(-x)^{-m}\,\Gamma(m, -x)\right)$$

```
> int(exp(-x^2/2),x); # erf = Error function
```

$$\frac{\sqrt{\pi}\,\sqrt{2}\,\mathrm{erf}\left(\frac{\sqrt{2}\,x}{2}\right)}{2}$$

```
> int(exp(sin(x)),x); # not representable
```

$$\int \mathrm{e}^{\sin(x)}\,\mathrm{d}x$$

- Definite integrals:

```
> int(sin(x)*cos(x)^2,x=0..Pi);
```

$$\frac{2}{3}$$

- Improper integrals:

```
> int(ln(x),x=1..infinity); # divergent
```

$$\infty$$

```
> int(1/sqrt(x),x=0..1); # convergent
```

$$2$$

```
> int(exp(-x^2/2),x=-infinity..infinity); # convergent
```

$$\sqrt{2}\,\sqrt{\pi}$$

- Numerical approximation:

```
> int(exp(sin(x)),x=0..1,numeric);
```

$$1.631869608$$

```
> evalf(int(exp(sin(x)),x=0..1)); # alternative version
```

$$1.631869608$$

Multiple integrals:

```
> int(x*y,x,y); # indefinite
```

$$\frac{x^2 y^2}{4}$$

```
> Int(x*y,x=0..y,y=0..1) = int(x*y,x=0..y,y=0..1); # definite
```

$$\int_0^1 \int_0^y x\,y\,dx\,dy = \frac{1}{8}$$

2.4 Series expansions

taylor performs univariate taylor expansion, with remainder:

```
> taylor(exp(x),x); # default: expanision about 0
```

$$1 + x + \frac{1}{2} x^2 + \frac{1}{6} x^3 + \frac{1}{24} x^4 + \frac{1}{120} x^5 + O(x^6)$$

```
> taylor(F(u),u);
```

$$F(0) + D(F)(0)\,u + \frac{1}{2} D^{(2)}(F)(0)\,u^2 + \frac{1}{6} D^{(3)}(F)(0)\,u^3 + \frac{1}{24} D^{(4)}(F)(0)\,u^4 + \frac{1}{120} D^{(5)}(F)(0)\,u^5 + O(u^6)$$

More generally:

```
> taylor(exp(x),x=1,8); # expansion of order 8 about x=1
```

$$e + e\,(x-1) + \frac{1}{2} e\,(x-1)^2 + \frac{1}{6} e\,(x-1)^3 + \frac{1}{24} e\,(x-1)^4 + \frac{1}{120} e\,(x-1)^5 + \frac{1}{720} e\,(x-1)^6 + \frac{1}{5040} e\,(x-1)^7 + O((x-1)^8)$$

The environment variable **Order** represents the length of a series expansion (default=6).

```
> Order; Order := 10;
```

$$6$$

$$Order := 10$$

```
> tay := taylor(exp(x*y),x);
```

$$tay := 1 + y\,x + \frac{1}{2} y^2 x^2 + \frac{1}{6} y^3 x^3 + \frac{1}{24} y^4 x^4 + \frac{1}{120} y^5 x^5 + \frac{1}{720} y^6 x^6 + \frac{1}{5040} y^7 x^7 + \frac{1}{40320} y^8 x^8 + \frac{1}{362880} y^9 x^9 + O(x^{10})$$

If you want to use the taylor polynomial (without remainder) for further computations, remove remainder using **convert:**

```
> taypol := convert(tay,polynom);
```

$$taypol := 1 + x\,y + \frac{1}{2} x^2 y^2 + \frac{1}{6} y^3 x^3 + \frac{1}{24} y^4 x^4 + \frac{1}{120} y^5 x^5 + \frac{1}{720} y^6 x^6 + \frac{1}{5040} y^7 x^7$$

$$+\frac{1}{40320}y^8x^8+\frac{1}{362880}y^9x^9$$

mtaylor performs multivariate taylor expansion, without remainder:

```
> mtaylor(cos(x+y),[x,y],3);
```

$$1-\frac{1}{2}x^2-xy-\frac{1}{2}y^2$$

```
> Order := 4;
```

$$Order := 4$$

series is more general than taylor:

```
> f := 1/sin; # the function x -> 1/sin(x)
```

$$f := \frac{1}{\sin}$$

```
> taylor(f(x),x=0);
Error, does not have a taylor expansion, try series()
> series(f(x),x=0); # This is a 'Laurent series'
```

$$x^{-1}+\frac{1}{6}x+O(x^3)$$

```
> series(sqrt(x)*exp(x),x=0); # this function does not
                              # admit a Taylor expansion about
  x=0
```

$$\sqrt{x}+x^{3/2}+\frac{x^{5/2}}{2}+\frac{x^{7/2}}{6}+O(x^{9/2})$$

2.5 Solving differential equations

dsolve can solve differential equations symbolically as well as numerically.
Syntax similar to **solve**. Versatile, many options.

Example for an exact (symbolic) solution:

```
> ode := D(u)(t)=u(t)^2; # the ODE u' = u^2
```

$$ode := \mathrm{D}(u)(t) = u(t)^2$$

```
> ic := u(0)=1; # initial condition u(0)=1
```

$$ic := u(0) = 1$$

Solve for function u(t) :

```
> dsolve([ode,ic],u(t));
```

$$u(t) = -\frac{1}{t-1}$$

```
> assign(%): u(t);
```

$$-\frac{1}{t-1}$$

The same example, but numerical solution. Note that the solution exists only up to t=1.

```
> u:='u':
  num_sol := dsolve([ode,ic],u(t),numeric,range=0..1);
Warning, cannot evaluate the solution further right of .99999999, probably a singularity
```

$$num\_sol := \mathbf{proc}(x\_rkf45) \ \ldots \ \mathbf{end\ proc}$$

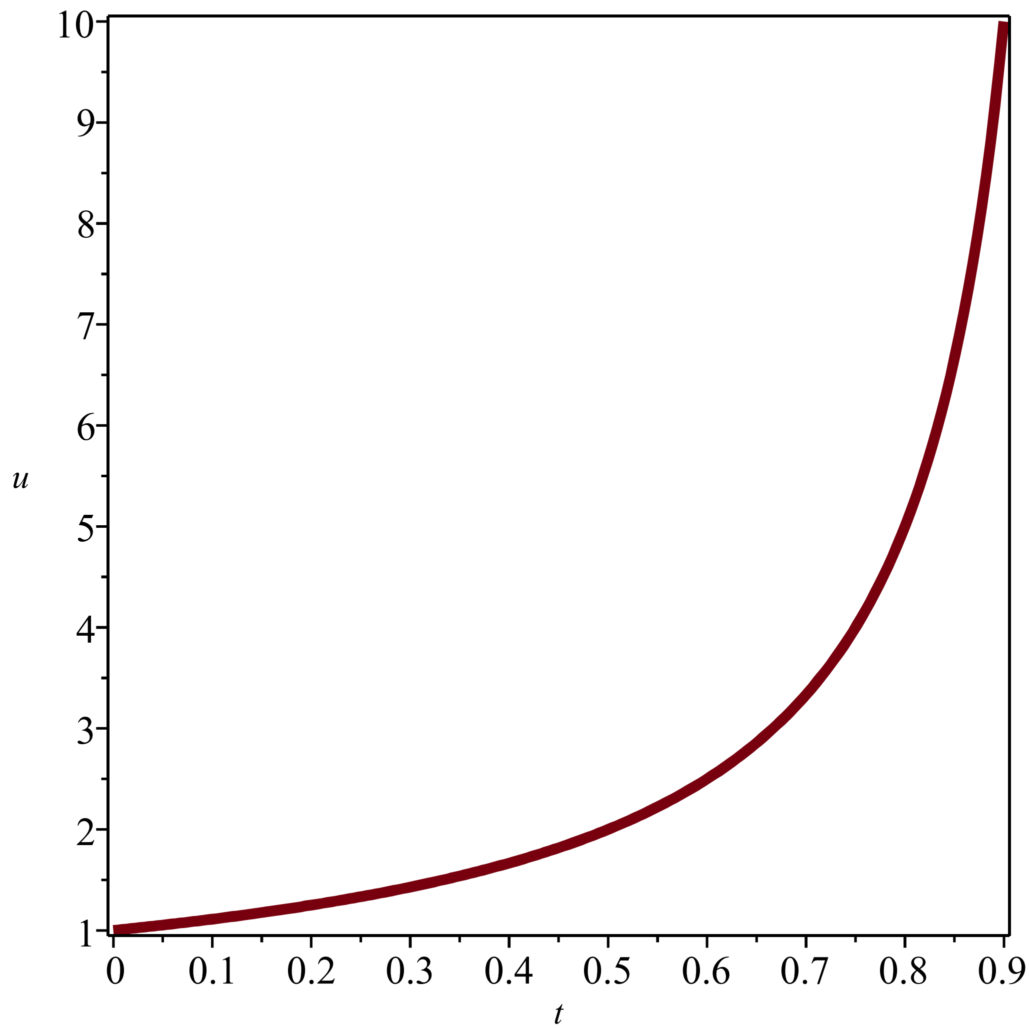
In this case an adaptive Runge-Kutta method has been used (the default for numerical solution), and the output of dsolve is a **procedure** which you can call for retrieving values:

```
> num_sol(0.99999);
```

$$[t = 0.99999, u(t) = 100056.896715508]$$

The function **odeplot** from the **plots** package can be directly used to plot the solution:

```
> plots[odeplot](num_sol,t=0..0.9,axes=boxed,thickness=4);
```



u
t
1
2
3
4
5
6
7
8
9
10
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9

3 Control structures

```
> restart;
```

Maple includes a powerful **programming language**.

Statements can be combined using

- **if ... then ... else ... elif ... end [if]** (conditional construct)
- **do ... end [do]** (basic loop construct)
- **for ... do ... end [do]** (repetition: explicit for-loop)
- **while ... do ... end [do]** (repetition: while-loop)

and related / more general constructs.

; and **:** after a construct work like after single command.

3.1 Conditional constructs

if - construct: **if ... then ... else ... elif ... end [if]**

Instruction by examples:

```
> a := 1;
```

$$a := 1$$

```
> if a=1 then
     print("a has value 1 assigned")
  end if;
```

"a has value 1 assigned"

```
> a := 2;
```

$$a := 2$$

```
> if a=1 then
     print("a has value 1 assigned")
  end if;
> a := 1;
```

$$a := 1$$

```
> if a=1 then
     print("a has value 1 assigned")
  else
     print("a has NOT value 1 assigned")
  end if;
```

"a has value 1 assigned"

```
> a := 2;
```

$$a := 2$$

```
> if a=1 then
     print("a has value 1 assigned")
  else
     print("a has NOT value 1 assigned")
  end if;
```

"a has NOT value 1 assigned"

Or more generally. Note: **A test may also fail.**

```
> a := 'a';
```

$$a := a$$

```
> if a=1 then
     print("a has value 1 assigned")
  elif (a=2 or a=3) then
     print("a has value 2 or 3 assigned")
  elif (a>=4 and a<=10) then
     print("a has value between 4 and 10 assigned")
  else
     print("none of above conditions satisfied");
     print("other value for a")
  end if;
Error, cannot determine if this expression is true or false: 4 <= a and a <= 10
```

Alternative, short version: **ifelse**

```
> a := 'a';
```

$$a := a$$

```
> ifelse(a=0,
         print("0"),          # if branch
         print("ungleich 0") # else branche
        );
```

"ungleich 0"

Note: <> **0 is generic.**

3.2 Basic do-loop

do ... end [do] is only reasonable when combined with a **break** condition. **next** is also available.

Example: generate random numbers, accept only odd ones,
stop if 10 numbers have been accepted.

```
> r:=rand(1..100); # initialize random number generator
```

$r :=$ **proc**( )
proc( ) **option** *builtin* = *RandNumberInterface*; **end proc**(6, 100, 7) + 1
end proc

```
> r();
```

93

```
> i:=0:
  do
     random_number:=r():
     if is(random_number,even) then next end if;
     i:=i+1:
     print(random_number):
     if i=10 then break end if
  end do:
```

45
59
69
27
17
43
83
25
93
93

break and **next** can also be uses in for- and while loops (next section).

3.3 for- and while loop

for - loop: **for ... do ... end [do]**

Instruction by examples:

```
> summe := 0:
  for i from 1 to 10 do
      summe := summe + i;
  end do:
  summe;
```

55

```
> s := alpha,beta:
  for j from 2 by 2 to 10 do
      s := 2*s[1],3*s[2]
  end do:
  s;
```

$32\,\alpha, 243\,\beta$

```
> for x from 0 to 1 by 0.1 do x end do: x;
```

1.1

```
> for i from 10 to 0 by -3 do end do: i;
```

-2

```
> p := 1;
  for i from 1 to 10 while p<1000 do
```

```
      p:=p*i;
      lprint(i,p):
  end do:
```

$$p := 1$$

```
1, 1
2, 2
3, 6
4, 24
5, 120
6, 720
7, 5040
```

```
> for letter from "a" to "c" do letter end do;
```

"a"

"b"

"c"

Variant: for loop scanning data structure (e.g., sequence,list or set):

```
> s := seq(i,i=1..3);
```

$$s := 1, 2, 3$$

```
> for i in s do i^2 end do;
```

1

4

9

```
> Letters := ["x","y","z"];
```

$$Letters := [\text{"x"}, \text{"y"}, \text{"z"}]$$

```
> Names := [];
```

$$Names := [\ ]$$

```
> for letter in Letters do
      n := convert(letter,name);
      Names := [op(Names),n];
  end do:
  Names;
```

$$[1.1, y, z]$$

(Remark: op(*list)* returns contents of *list* as expression sequence.)

Control structures are mainly used within **procedures**.

4 Procedures

```
> restart;
```

4.1 Functions revisited

Functions are `simple procedures'. You can use **if,** but no other control structures, and you cannot define variables within a function definition.

Examples:

```
> f := x -> if x=0 then 0 else 1 end if:
> f(0),f(2);
```

0, 1

```
> f := x->for i from 1 to x do i end do:
Error, reserved word `for` unexpected
```

In general, use procedures (**proc**) depending on one or several arguments and giving back one or several objects as results

4.2 Basics of procedures

Basic syntax of a procedure:

proc*(arguments)*
local *local_variables; # if applicable*
global *global_variables; # if applicable*
options ... *# if applicable*
sequence of commands processing the arguments and computing a result
return *result*
end [proc];

Some simple examples:

```
> proc(x) x^2 end proc; # same as x -> x^2, unnamed
```

proc(x) x^2 **end proc**

```
> %(5);
```

25

```
> hello_world := # we assign a name to this procedure
  proc()         # no arguments
    print("Hello, World")
  end proc:
> hello_world();
```

"Hello, World"

```
> mylimit := proc(sequence,variable,limitpoint)
```

```
  # return a limit in inert and evaluated form
    return Limit(sequence,variable=limitpoint) =
           limit(sequence,variable=limitpoint)
  end proc:
> print(mylimit);
```

proc(*sequence*, *variable*, *limitpoint*)

return $Limit(sequence, variable = limitpoint) = limit(sequence, variable = limitpoint)$

end proc

```
> mylimit(1+1/'n','n',infinity);
```

$$\lim_{n \to \infty} \left(1 + \frac{1}{n}\right) = 1$$

```
> mynorm := proc(l,output)
  # expects a list l
  # computes sqrt(sum of squares)
  local i,s:=0;
  for i from 1 to numelems(l) do
      s := s + l[i]^2
  end do;
  s := sqrt(s);
  if (output="f") then # apply evalf
     s := evalf(s)
  end if;
  return s;
  end proc:
> mynorm([a,b,c],"");
```

$$\sqrt{a^2 + b^2 + c^2}$$

```
> mynorm([1,2,3],"");
```

$$\sqrt{14}$$

```
> v := mynorm([1,2,3],"f");
```

$$v := 3.741657387$$

General rules for procedures:

- The code of a procedure is usually part of a worksheet and is edited like a normal statment sequence.

- In particular, use **<shift><enter> for new line**, <enter> only at the end of specifying the procedure.

- There are no general typing rules.

- Variables local to the procedure must be declared using **local**.

- Global variables (existing outside) can be accessed (read/write); must be declared.

  But: changing the value of a global variable is usually not recommended: 'hidden' return value

- If the **return** statement is missing: the value last computed is returned

- **end proc:** instead of **end proc;** suppresses output of code on screen

- Procedures may call other procedures and may be recursive.

- Procedures must be called with correct number of arguments (>= 0).

- In principle, arguments of arbitrary types can be passed,
but only reasonable if operation of the procedure is well-defined for these types.

Special topics are discussed later on:

- Available options, docmentation of procedures

- Type checking (automatic or manual)

- Precise rules for passing arguments

- Variable argument lists

- Debugging

- ...

4.3 Some examples for procedures

A **recursive** procedure:

Recursive summation of objects in a list.
Note that each recursive procedure must include
a termination condition for the recursion.

This simple code works only for lists of length 2^n.
This is only an exercise - not more efficient than add.

```
> recursive_sum := proc(list)
  local len:=numelems(list);
  # stop recursion if length 1
  if len=1 then
     return list[1]
  else
     return  recursive_sum(list[1..len/2])
           + recursive_sum(list[len/2+1..len]);
  end if;
  end proc:
> recursive_sum([1]);
```

$$1$$

```
> recursive_sum([1,2,3,4]);
```

$$10$$

```
> recursive_sum([1,2,3,4,5,6,7,8]);
```

$$36$$

```
> add([$1..8]);
```

$$36$$

Example:

Differentiate the composition of 3 given functions
and evaluate the result to float at a given point:

```
> chaindiff := proc(f1,f2,f3,eval)
    return evalf(subs(x=eval,diff((f1@f2@f3)(x),x)))
  end proc:
> chaindiff(sin,cos,z->z^2,3);
```

$$-1.515408329$$

Example:

Take a list of values [a0,a1,a2,...,an] and a name or values x and generate the polynomial expression

a0 + a1*x + a2*x^2 + ... + an*x^(n-1)

```
> pol_from_list := proc(list,x)
  local i,pol:=0; # pol is initialized to 0
  for i from 1 to numelems(list) do
      pol := pol + list[i]*x^(i-1)
  end do;
  return pol
  end proc:
> pol_from_list([1,2,3,4],y);
```

$$4 y^3 + 3 y^2 + 2 y + 1$$

```
> pol_from_list([a,b,c,d],8);
```

$$a + 8 b + 64 c + 512 d$$

Example:

Generate and show Pascal triangle of depth N:
(no return value.)

Parameter eval (true/false) controls evaluation of binomial coefficients (**binomial**).
If unevalueted, binomial(n,k) is shown displayed as B(n,k).

```
> pascal_triangle := proc(N,eval)
  local n,B;
  if eval then B:=binomial end if;
  for n from 0 to N do
      print(seq(B(n,k),k=0..n))
  end do
  end proc:
> pascal_triangle(4,false),
  pascal_triangle(4,true);
```

$$B(0, 0)$$

$$B(1, 0), B(1, 1)$$

$$B(2, 0), B(2, 1), B(2, 2)$$

$$B(3, 0), B(3, 1), B(3, 2), B(3, 3)$$

$$B(4,0), B(4,1), B(4,2), B(4,3), B(4,4)$$

$$1$$

$$1, 1$$

$$1, 2, 1$$

$$1, 3, 3, 1$$

$$1, 4, 6, 4, 1$$

Example:

A procedure which returns a function, namely the indefinite integral of a given function:

```
> defint := proc(f)
  return x->int(f(x),x)
  end proc:
> g:=defint(cos);
```

$$g := x \mapsto \int \cos(x)\, dx$$

```
> g(x);
```

$$\sin(x)$$

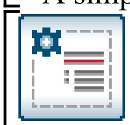
4.4 The code editor

... works, but not very convenient.

Usage: **Insert > Code Edit Region**

The control via right mouse button.

A simple example:



```
p:=proc(x)
```

$$p := \textbf{proc}(x)\ \textbf{return}\ x\hat{}2\ \textbf{end proc}$$

```
> p(2);
```

$$p(2)$$

== end of Part II ==