Computer Algebra using Maple
Part III: Data structures;
Procedures in more detail; Data output

Winfried Auzinger, Kevin Sturm (SS 2019)

1 Tables

```
> restart:
```

A **table** is an *associative array*, i.e., indices may be arbitrary objects, not only integers.

Example:

```
> day := table([m="Monday",t="Tuesday",w="Wednesday"]);
```

$$day := \mathrm{table}([m = \text{"Monday"}, t = \text{"Tuesday"}, w = \text{"Wednesday"}])$$

```
> is(day,table);
```

$$true$$

```
> day[m];
```

"Monday"

Default indices are 1,2,3,... :

```
> month:=table(["January","February","March"]);
```

$$month := \mathrm{table}([1 = \text{"January"}, 2 = \text{"February"}, 3 = \text{"March"}])$$

```
> month[2];
```

"February"

```
> indices(month); # all indices
```

$$[1], [2], [3]$$

```
> entries(month); # all entries
```

["January"], ["February"], ["March"]

In contrast to lists, tables are organized **dynamically**. Further entries can be added in a simple way:

```
> month[5]:="May";
```

$$month_5 := \text{"May"}$$

```
> month[4]:="April";
```

$$month_4 := \text{"April"}$$

```
> month; # specifying name of table does not evaluate it
```

$$month$$

```
> eval(month); # evaluate (show contents of table)
```

$$\mathrm{table}([1 = \text{"January"}, 2 = \text{"February"}, 3 = \text{"March"}, 4 = \text{"April"}, 5 = \text{"May"}])$$

```
> print(month);
```

$$\mathrm{table}([1 = \text{"January"}, 2 = \text{"February"}, 3 = \text{"March"}, 4 = \text{"April"}, 5 = \text{"May"}])$$

Specifying the name of the table does not output its contents,
a principle called **last name evaluation**.

This is reasonable for objects of such a type, which may be large.

2 Arrays

An **Array** is a **static**, multidimensional 'rectangular' data structure; think of a vector (1D) or a matrix (2D).
Explicit initialization (storage allocation) is therefore required (default values: 0).

Example:

```
> A := Array(1..5);
  whattype(A);
```

$$A := \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$Array$$

```
> A[1]:=x; A[2]:=x->x^2;
```

$$A_1 := x$$

$$A_2 := x \mapsto x^2$$

```
> print(A);
```

$$\begin{bmatrix} x & x \mapsto x^2 & 0 & 0 & 0 \end{bmatrix}$$

Defaults for options:

-- datatype = anything (any valid type)
-- storage = rectangular ('normal' storage; other options exist!)
-- order = Fortran_order (order in which elements are internally stored):
Example: (1,1),(2,1),(3,1), ... (1,2),(2,2),(3,2) ('colum-major order', as in Matlab)

Normal use: for storing **numerical data.** Datatype may be specified (e.g. integer[4], float[8])
float[8] is standard double precision.

Default initialization with 0.

Example 2-dimensional (also with more general index range!)

```
> A := Array(0..2,0..3,datatype=float[8]);
```

$$A := Array(0..2, 0..3, \varnothing, datatype = float_8)$$

```
> A[0,0] := 1;
```

$$A_{0,\,0} := 1$$

```
> A[0,0], A[0,1];
```

$$1., 0.$$

```
> print(A); # does not look not so nice
```

$$Array(0..2, 0..3, \{(0, 0) = 1.\}, datatype = float_8)$$

Example: with explicit initialization using listlist:

```
> A := Array([[1,2,3,4],[5,6,7,8]]);
```

$$A := \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{bmatrix}$$

Example 3-dimensional (more general index range):

```
> A := Array(-3..3,-3..3,-3..3,datatype=integer[4]);
```

$$A := \begin{bmatrix} \textit{-3..3 x -3..3 x -3..3 Array} \\ \textit{Data Type: } integer_4 \\ \textit{Storage: rectangular} \\ \textit{Order: Fortran\_order} \end{bmatrix}$$

```
> A[3,3,3];
```

$$0$$

Alternative: Generate entries using **indexing function**:

```
> A := Array(0..3,0..4,(i,j)->i*j);
```

$A := Array(0..3, 0..4, \{(1,1)=1, (1,2)=2, (1,3)=3, (1,4)=4, (2,1)=2, (2,2)=4, (2,3)=6, (2,4)=8, (3,1)=3, (3,2)=6, (3,3)=9, (3,4)=12\})$

Test membership (this works for general data structures):

```
> member(1,A), member(-1,A);
```

$$true, false$$

Remarks:

-- Main advantage of arrays: General index range. (Default: start from 1.)

-- Arrays are implemented using the general concept of an **rtable.**
rtable (`rectangular table') is a general, versatile data structure.
In particular, rtables support **special storage modes**, e.g. **symmetric, triangular, sparse, empty**...
(think of matrices!), for efficient and automatic handling of special cases.

-- Storage mode **empty:** Used, e.g., for ZeroMatrix, IdentityMatrix (see Section 3) - these are represented as **symbols**, read-only, nothing is stored.

-- For standard cases vector and matrix, where index starting with 1,
we better use the related, more special data structures **Vector**, **Matrix**
(also derived from rtable).

3 Vectors and Matrices

These may be considered as special types of Arrays.
They can be used for symbolic, numerical, or mixed data.

As for Arrays, *static initialization* with fixed dimension is required, typically with the `constructors' **Vector** and **Matrix** (these have the same names as the corresponding data types).

Vector is one-dimensional, index starts with 1.

```
> v := Vector(5);
  whattype(v);
```

$$v := \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

$$Vector_{column}$$

```
> Vector[column]([3$5]); # same as Vector(...)
```

$$\begin{bmatrix} 3 \\ 3 \\ 3 \\ 3 \\ 3 \end{bmatrix}$$

```
> f:=i->i^2;
  v:=Vector(5,f); # initialization with indexing function
```

$$f := i \mapsto i^2$$

$$v := \begin{bmatrix} 1 \\ 4 \\ 9 \\ 16 \\ 25 \end{bmatrix}$$

Vector element, subvector:

```
> v[1], v[2];   # first, second
```

```
v[-1], v[-2]; # last, second to last
v[1..3]; # subvector
```

$$1, 4$$

$$25, 16$$

$$\begin{bmatrix} 1 \\ 4 \\ 9 \end{bmatrix}$$

```
> Vector[row](4); # a row Vector
```

$$\begin{bmatrix} 0 & 0 & 0 & 0 \end{bmatrix}$$

```
> whattype(%);
```

$$Vector_{row}$$

Inner product of vectors: Use dot: **.**

```
> x:=Vector[row]([1,2,3]);
  y:=Vector([xi,eta,zeta]);
  x.y
```

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

$$y := \begin{bmatrix} \xi \\ \eta \\ \zeta \end{bmatrix}$$

$$\xi + 2\,\eta + 3\,\zeta$$

The dot operator **.** generally stands for Vector-Vector and Vector-Matrix multiplication.
For the case of an inner product of two Vectors, it also accepts two column Vectors as arguments, i.e., transposing the first argument is not necessary (a pragmatic implementation):

```
> x:=Vector([1,2,3]);
  y:=Vector([xi,eta,zeta]);
  x.y
```

$$x := \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

$$y := \begin{bmatrix} \xi \\ \eta \\ \zeta \end{bmatrix}$$

$$\xi + 2\,\eta + 3\,\zeta$$

Matrix is two-dimensional, indices start with 1.

```
> M := Matrix(3,4);
  whattype(M);
```

$$M := \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$Matrix$$

```
> Matrix([[a,2,c,4],[1,b,3,d]]); # specify row-wise
```

$$\begin{bmatrix} a & 2 & c & 4 \\ 1 & b & 3 & d \end{bmatrix}$$

```
> M := Matrix(5,5,(i,j)->1/(i+j-1)); # generate Hilbert Matrix
  using indexing function
```

$$M := \begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} \\ \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} \\ \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} \end{bmatrix}$$

Matrix element, submatrix:

```
> M[1,1], M[1,2], M[1,-1]; # first two elements and last element
  in first row
  M[1,2], M[2,2], M[-1,2]; # first two elements and last element
  in scond column
  M[-1,-1], M[-2,-2]; # last and second to last diagonal element
```

$$1, \frac{1}{2}, \frac{1}{5}$$

$$\frac{1}{2}, \frac{1}{3}, \frac{1}{6}$$

$$\frac{1}{9}, \frac{1}{7}$$

```
> M[1..2,3..4]; # submatrix
```

$$\begin{bmatrix} \frac{1}{3} & \frac{1}{4} \\ \frac{1}{4} & \frac{1}{5} \end{bmatrix}$$

Row, column of a matrix (.. is the analog to : in MATLAB):

```
> M[1,..], M[..,2];
```

$$\begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \end{bmatrix}, \begin{bmatrix} \frac{1}{2} \\ \frac{1}{3} \\ \frac{1}{4} \\ \frac{1}{5} \\ \frac{1}{6} \end{bmatrix}$$

Basic **arithmetic** with vectors and matrices:
for multiplication use . (not *):

```
> a:='a':
  A,x := Matrix(3,3,(i,j)->i+j),Vector(3,i->i^2);
```

$$A, x := \begin{bmatrix} 2 & 3 & 4 \\ 3 & 4 & 5 \\ 4 & 5 & 6 \end{bmatrix}, \begin{bmatrix} 1 \\ 4 \\ 9 \end{bmatrix}$$

```
> 2 * x,   # scalar * Vector
  3 * A,   # scalar * Matrix
  x + x,   # Vector + Vector
  A + A;   # Matrix + Matrix
```

$$\begin{bmatrix} 2 \\ 8 \\ 18 \end{bmatrix}, \begin{bmatrix} 6 & 9 & 12 \\ 9 & 12 & 15 \\ 12 & 15 & 18 \end{bmatrix}, \begin{bmatrix} 2 \\ 8 \\ 18 \end{bmatrix}, \begin{bmatrix} 4 & 6 & 8 \\ 6 & 8 & 10 \\ 8 & 10 & 12 \end{bmatrix}$$

```
> A . x,   # Matrix . Vector
  A . A,   # Matrix . Matrix
  A^3;     # Matrix power
```

$$\begin{bmatrix} 50 \\ 64 \\ 78 \end{bmatrix}, \begin{bmatrix} 29 & 38 & 47 \\ 38 & 50 & 62 \\ 47 & 62 & 77 \end{bmatrix}, \begin{bmatrix} 360 & 474 & 588 \\ 474 & 624 & 774 \\ 588 & 774 & 960 \end{bmatrix}$$

The following commands (and many others) are part of the **LinearAlgebra** package
(to be discussed in more detail later on)

```
> LinearAlgebra[Dimension](M);
```

$$5, 5$$

```
> LinearAlgebra[Row](M,1); # same as M[1,..]
```

$$\begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \end{bmatrix}$$

```
> LinearAlgebra[Column](M,2); # same as M[..,2]
```

$$\begin{bmatrix} \frac{1}{2} \\ \frac{1}{3} \\ \frac{1}{4} \\ \frac{1}{5} \\ \frac{1}{6} \end{bmatrix}$$

Or: activate complete package:

```
> with(LinearAlgebra):
```

Example: Vandermonde matrix

```
> n := 20:
> V := Matrix(n,n,(i,j)->xi[i]^j):
> V;
```

$$\begin{bmatrix} \textit{20 x 20 Matrix} \\ \textit{Data Type: anything} \\ \textit{Storage: rectangular} \\ \textit{Order: Fortran\_order} \end{bmatrix}$$

```
> V[1..4,1..4];
```

$$\begin{bmatrix} \xi_1 & \xi_1^2 & \xi_1^3 & \xi_1^4 \\ \xi_2 & \xi_2^2 & \xi_2^3 & \xi_2^4 \\ \xi_3 & \xi_3^2 & \xi_3^3 & \xi_3^4 \\ \xi_4 & \xi_4^2 & \xi_4^3 & \xi_4^4 \end{bmatrix}$$

Special matrices:

```
> LinearAlgebra[ZeroMatrix](3);
```

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

```
> print(LinearAlgebra[ZeroMatrix](100));
```

$$\begin{bmatrix} \textit{100 x 100 Matrix} \\ \textit{Data Type: anything} \\ \textit{Storage: empty} \\ \textit{Order: Fortran\_order} \end{bmatrix}$$

```
> LinearAlgebra[IdentityMatrix](3);
```

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

```
> print(LinearAlgebra[IdentityMatrix](100));
```

$$\begin{bmatrix} \textit{100 x 100 Matrix} \\ \textit{Data Type: anything} \\ \textit{Storage: empty} \\ \textit{Order: Fortran\_order} \end{bmatrix}$$

4 Storage management

Consider sequence of statements:
(Note that this is usually not good style!)

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

$$a := 1$$

```
> b := a; # current value of a is copied, gives value for b
```

$$b := 1$$

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

$$a := 2$$

```
> b;      # Change of a does not affect b.
```

$$1$$

But: For usually larger objects like [r]tables, Arrays, Vectors, Matrices, assigning via **:= does not copy the object,**
but sets a **pointer** to the original object:

```
> A := Matrix([[1,2],
               [3,4]]);
```

$$A := \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

```
> B := A;
```

$$B := \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

```
> A[1,2] := 1000; 'B'=B; # now also B has changed!
```

$$A_{1,2} := 1000$$

$$B = \begin{bmatrix} 1 & 1000 \\ 3 & 4 \end{bmatrix}$$

To copy such objects, use **copy**:

```
> A;
  B := copy(A);
```

$$\begin{bmatrix} 1 & 1000 \\ 3 & 4 \end{bmatrix}$$

$$B := \begin{bmatrix} 1 & 1000 \\ 3 & 4 \end{bmatrix}$$

```
> A[1,2]:=100000000; # now this does not affect the independent
  copy B
```

$$A_{1,\,2} := 100000000$$

```
> A:='A'; 'B'=B;
```

$$A := A$$

$$B = \begin{bmatrix} 1 & 1000 \\ 3 & 4 \end{bmatrix}$$

5 Stack, queue, heap

Use of stacks, queues and heaps is supported
(predefined functionality, implemented in so-called modules).

Example: use of the stack data structure

```
> S := stack[new]();
  stack[depth](S);
```

$$S := \text{table}([0 = 0])$$

$$0$$

```
> for i from 1 to 5 do
    stack[push](i,S)
  end do;
```

$$1$$
$$2$$
$$3$$
$$4$$
$$5$$

```
> stack[depth](S);
```

$$5$$

```
> for i from 1 to stack[depth](S) do
    stack[pop](S)
  end do;
```

$$5$$
$$4$$
$$3$$
$$2$$
$$1$$

```
> stack[empty](S);
```

$$\mathit{true}$$

6 General form of a procedure; examples

A procedure template:

```
> p := proc(FORMAL_PARAMETERS)
  description "Text";
  option OPTIONS;
  local LOCAL_VARIABLES;
  global GLOBAL_VARIABLES;
  # procedure body
  return ...; # result
  # or:
  error "errormessage"; # error exit
  end proc:
```

Note that the result of a prodedure returned using **return**
may be any valid Maple object or seuqence of Maple objects.

Example: search next prime number after given number n

For demo purpose, this also calls another procedure (advance).

```
> search_prime := proc(n,nmax)
  description "Search prime number >= n";
  option trace; # activate verbose behavior, see output below
  local i:=n;   # initialize local variable
  if is(i,even) then
     i:=i+1
  end if;
  do # general loop
    if isprime(i) then
       return i  # normal return
    end if;
    i:=advance(i);
    if i>nmax then
       error "Limit exceeded" # error exit with your own error
  message
    end if
  end do;
  end proc:
> advance := proc(i)
     return i+2
  end proc:
> Describe(search_prime); # show description

# Search prime number >= n
search_prime( n, nmax )
```

```
> search_prime(74,80);
{--> enter search_prime, args = 74, 80
```

$i := 74$

$i := 75$

$i := 77$

$i := 79$

```
<-- exit search_prime (now at top level) = 79}
```

79

```
> search_prime(74,76);
{--> enter search_prime, args = 74, 76
```

$i := 74$

$i := 75$

$i := 77$

```
<-- ERROR in search_prime (now at top level) = Limit exceeded}
Error, (in search_prime) Limit exceeded
```

... By the way: this is als predefined in Maple:

```
> nextprime(74);
```

79

In the next example, we use a general loop **do ... end do**, with **break** / **next**,
and the random number generator **rand**

Procedure generates lotto quick tipp, but all numbers are required to be divisible by q.

This may result in an infinite loop, especially if q is too large.
Stop with error after nmax rejections of a drawn number.

If second parameter q is not specified (this is admitted), q:=1 is assumed by default:
Default applies if second parameter q is not passed.

(We could also stop immediately if 1<=q<=6 is not satisfied.)

```
> lotto := proc(nmax,q:=1)
  description "generate lotto tip";
  local count:=0,drawn,number,rand1_45:
  randomize(): # initialize rand
  # drawing:
  rand1_45 :=rand(1..45):
  drawn:={}:
  do
    number:=rand1_45():
    if number mod q <> 0 then
       count := count+1:
       if count>=nmax then
          break
       else
          next
       end if
    else
```

```
      drawn := drawn union {number}:
      if numelems(drawn)=6 then
         return convert(drawn,list)
      end if:
    end if
  end do:
  error "# of maximal tries exceeded!"
  end proc:
```

```
> lotto(100);
```

$$[1, 13, 17, 21, 24, 40]$$

```
> lotto(100,2);
```

$$[8, 10, 26, 28, 36, 38]$$

```
> lotto(100,3);
```

$$[3, 21, 27, 30, 33, 42]$$

```
> lotto(1000,10);
Error, (in lotto) # of maximal tries exceeded!
```

Two further examples: use of **option remember**. This is especially useful for recursions where multiple calls occur. A so-called **remember table** is generated, where results already obtained are automatically stored and retrieved.

We measure the performance using the CPU clock (see section 9 below).

```
> fibonacci := proc(n)
  # without option remember
  if n<=2 then
     return 1
  else
     return fibonacci(n-1)+fibonacci(n-2)
  end if
  end proc:
> start:=time():
  fibonacci(30);
  time()-start;
```

$$832040$$
$$0.625$$

```
> fibonacci := proc(n)
  option remember;
  if n<=2 then
     return 1
  else
     return fibonacci(n-1)+fibonacci(n-2)
  end if
  end proc:
> start:=time():
  fibonacci(30);
  time()-start;
```

$$832040$$
$$0.$$

Naturally, this much more simple to realize using an explicit loop.

Example for a 'full' recursion (for each n, all lower values are required):

```
> p := proc(n)
  # without option remember
  local i;
  if n=1 then
     return 1
  else
     return add(n*(n-i)*p(i),i=1..n-1)
  end if
  end proc:
> start:=time():
  p(22);
  time()-start;
```

7659788864114562120948

1.531

```
> p := proc(n)
  option remember; # with option remember
  local i;
  if n=1 then
     return 1
  else
     return add(n*(n-i)*p(i),i=1..n-1)
  end if
  end proc:
> start:=time():
  p(22);
  time()-start;
```

7659788864114562120948

0.

In this example, the storage requirement for remember table is proportional to n.

Converting this to an explicit loop also requires allocating storage:

```
> p := proc(n)
  local i,j,values:=[0$n]:
  values[1]:=1:
  for i from 2 to n do
      print(seq(values[j],j=1..i-1));
      values[i]:=add(i*(i-j)*values[j],j=1..i-1):
  end do:
  return values[n]
  end proc:
> start:=time():
  p(22);
  time()-start;
```

1

1, 2

1, 2, 12

1, 2, 12, 76

1, 2, 12, 76, 550
1, 2, 12, 76, 550, 4506
1, 2, 12, 76, 550, 4506, 41286
1, 2, 12, 76, 550, 4506, 41286, 418648
1, 2, 12, 76, 550, 4506, 41286, 418648, 4656708
1, 2, 12, 76, 550, 4506, 41286, 418648, 4656708, 56392010
1, 2, 12, 76, 550, 4506, 41286, 418648, 4656708, 56392010, 738683000
1, 2, 12, 76, 550, 4506, 41286, 418648, 4656708, 56392010, 738683000, 10408197588
1, 2, 12, 76, 550, 4506, 41286, 418648, 4656708, 56392010, 738683000, 10408197588, 156984674418
1, 2, 12, 76, 550, 4506, 41286, 418648, 4656708, 56392010, 738683000, 10408197588, 156984674418, 2523763381874
1, 2, 12, 76, 550, 4506, 41286, 418648, 4656708, 56392010, 738683000, 10408197588, 156984674418, 2523763381874, 43083378955050
1, 2, 12, 76, 550, 4506, 41286, 418648, 4656708, 56392010, 738683000, 10408197588, 156984674418, 2523763381874, 43083378955050, 778360970710384
1, 2, 12, 76, 550, 4506, 41286, 418648, 4656708, 56392010, 738683000, 10408197588, 156984674418, 2523763381874, 43083378955050, 778360970710384, 14837325735353704
1, 2, 12, 76, 550, 4506, 41286, 418648, 4656708, 56392010, 738683000, 10408197588, 156984674418, 2523763381874, 43083378955050, 778360970710384, 14837325735353704, 297616426348595922
1, 2, 12, 76, 550, 4506, 41286, 418648, 4656708, 56392010, 738683000, 10408197588, 156984674418, 2523763381874, 43083378955050, 778360970710384, 14837325735353704, 297616426348595922, 6266430551668100292
1, 2, 12, 76, 550, 4506, 41286, 418648, 4656708, 56392010, 738683000, 10408197588, 156984674418, 2523763381874, 43083378955050, 778360970710384, 14837325735353704, 297616426348595922, 6266430551668100292, 138190411487129741980
1, 2, 12, 76, 550, 4506, 41286, 418648, 4656708, 56392010, 738683000, 10408197588, 156984674418, 2523763381874, 43083378955050, 778360970710384, 14837325735353704, 297616426348595922, 6266430551668100292, 138190411487129741980, 3185272450532432297310
76597888641145621209482
0.031

Note: Recursive formulation of algorithms is often elegant and convenient, but this is not always of optimal efficiency. In particular, it requires a certain amount of storage, because the data for the recursively pending calls are put on the internal **stack**.

Algorithmically, a recursive procedure performs an loop `in disguise'.

7 Argument evaluation in procedure calls

RULE: 'Call by value', i.e.,

On procedure call, arguments are first evaluated and then passed to the procedure.

Example:

```
> p := proc(a)
    print(a)
  end proc:
> p(a);
```

$$2$$

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

$$a := 1$$

```
> p(a);
```

$$1$$

```
> p('a');
```

$$a$$

> Do **not** try to generate 'hidden' ouput by **redefining arguments** within a procedure. This may work to some extent but is not reliable.

For output, use **return** to return the desired values or objects.

8 Evaluation rules for variables

```
> restart:
```

Usually, variables defined along 'chains' are **fully** evaluated.

Example: (Note: This is **not** good style!)

```
> c:=b;
```

$$c := b$$

```
> b:=a;
```

$$b := a$$

```
> c;
```

$$a$$

Within a procedure, this does not work:

```
> p:=proc()
  local a,b,c:
  c:=b:
  b:=a:
  c;
  end proc:
> p();
```

$$b$$

This rule is called **level-1-evaluation**
(implemented for efficiency reasons).

On the other hand, the following version uses a chain of
level-1-evaluation and works as expected:

```
> p:=proc()
  local a,b,c:
  b:=a:
  c:=b:
  c;
  end proc:
> p();
```

$$a$$

9 CPU stop watch

Using the cpu timer **time()** you can measure computing times, e.g., for a block of statements or function or procedure call:

```
> start_time:=time(); # CPU seconds consumed in this session
  int(exp(-x^n),x); # do something
  time_used := time()-start_time;
```

$$start\_time := 3.406$$

$$\frac{1}{n}\left(\frac{n^2 x^{-n+1}\left(n x^n+n+1\right)\left(x^n\right)^{-\frac{n+1}{2n}} \mathrm{e}^{-\frac{x^n}{2}} \operatorname{WhittakerM}\left(\frac{1}{n}-\frac{n+1}{2n}, \frac{n+1}{2n}+\frac{1}{2}, x^n\right)}{(n+1)(2n+1)} + \frac{n x^{-n+1}(n+1)\left(x^n\right)^{-\frac{n+1}{2n}} \mathrm{e}^{-\frac{x^n}{2}} \operatorname{WhittakerM}\left(\frac{1}{n}-\frac{n+1}{2n}+1, \frac{n+1}{2n}+\frac{1}{2}, x^n\right)}{2n+1}\right)$$

$$time\_used := 0.140$$

```
> start_time:=time(); # CPU seconds consumed in this session
  int(exp(-x^n),x); # do something
  time_used := time()-start_time;
```

$$start\_time := 3.546$$

$$\frac{1}{n}\left(\frac{n^2 x^{-n+1}\left(n x^n+n+1\right)\left(x^n\right)^{-\frac{n+1}{2n}} \mathrm{e}^{-\frac{x^n}{2}} \operatorname{WhittakerM}\left(\frac{1}{n}-\frac{n+1}{2n}, \frac{n+1}{2n}+\frac{1}{2}, x^n\right)}{(n+1)(2n+1)} + \frac{n x^{-n+1}(n+1)\left(x^n\right)^{-\frac{n+1}{2n}} \mathrm{e}^{-\frac{x^n}{2}} \operatorname{WhittakerM}\left(\frac{1}{n}-\frac{n+1}{2n}+1, \frac{n+1}{2n}+\frac{1}{2}, x^n\right)}{2n+1}\right)$$

$$time\_used := 0.$$

Often one observes that doing the same thing again consumes much less CPU time. (Caching of results?)

10 Formatted output; using data files

Remember:

-- print does 'normal' printing of an object;
(very large objects are not printed in detail).

-- lprint uses 'typewriter' printing

Examples:

```
> print(sqrt(x^3));
```

$$\sqrt{x^3}$$

```
> lprint(sqrt(x^3));
(x^3)^(1/2)
> A := Matrix(10,10,(i,j)->i+j):
> print(A);
```

$$\begin{bmatrix} 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\ 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 \\ 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \\ 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\ 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 \\ 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 \\ 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 \\ 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 \\ 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 & 20 \end{bmatrix}$$

```
> lprint(A); # not so nice
Matrix(10, 10, {(1, 1) = 2, (1, 2) = 3, (1, 3) = 4, (1, 4) = 5, (1,
5) = 6, (1, 6) = 7, (1, 7) = 8, (1, 8) = 9, (1, 9) = 10, (1, 10) =
11, (2, 1) = 3, (2, 2) = 4, (2, 3) = 5, (2, 4) = 6, (2, 5) = 7, (2,
6) = 8, (2, 7) = 9, (2, 8) = 10, (2, 9) = 11, (2, 10) = 12, (3, 1)
= 4, (3, 2) = 5, (3, 3) = 6, (3, 4) = 7, (3, 5) = 8, (3, 6) = 9,
(3, 7) = 10, (3, 8) = 11, (3, 9) = 12, (3, 10) = 13, (4, 1) = 5,
(4, 2) = 6, (4, 3) = 7, (4, 4) = 8, (4, 5) = 9, (4, 6) = 10, (4, 7)
= 11, (4, 8) = 12, (4, 9) = 13, (4, 10) = 14, (5, 1) = 6, (5, 2) =
7, (5, 3) = 8, (5, 4) = 9, (5, 5) = 10, (5, 6) = 11, (5, 7) = 12,
(5, 8) = 13, (5, 9) = 14, (5, 10) = 15, (6, 1) = 7, (6, 2) = 8, (6,
3) = 9, (6, 4) = 10, (6, 5) = 11, (6, 6) = 12, (6, 7) = 13, (6, 8)
= 14, (6, 9) = 15, (6, 10) = 16, (7, 1) = 8, (7, 2) = 9, (7, 3) =
10, (7, 4) = 11, (7, 5) = 12, (7, 6) = 13, (7, 7) = 14, (7, 8) =
15, (7, 9) = 16, (7, 10) = 17, (8, 1) = 9, (8, 2) = 10, (8, 3) =
11, (8, 4) = 12, (8, 5) = 13, (8, 6) = 14, (8, 7) = 15, (8, 8) =
```

```
16, (8, 9) = 17, (8, 10) = 18, (9, 1) = 10, (9, 2) = 11, (9, 3) =
12, (9, 4) = 13, (9, 5) = 14, (9, 6) = 15, (9, 7) = 16, (9, 8) =
17, (9, 9) = 18, (9, 10) = 19, (10, 1) = 11, (10, 2) = 12, (10, 3)
= 13, (10, 4) = 14, (10, 5) = 15, (10, 6) = 16, (10, 7) = 17, (10,
8) = 18, (10, 9) = 19, (10, 10) = 20}, datatype = anything, storage
= rectangular, order = Fortran_order, shape = [])
```

```
> A := Matrix(11,11,(i,j)->i+j):
> print(A);
```

$$\left[\begin{array}{c} \textit{11 x 11 Matrix} \\ \textit{Data Type: anything} \\ \textit{Storage: rectangular} \\ \textit{Order: Fortran\_order} \end{array}\right]$$

```
> lprint(A);
Matrix(11, 11, {(1, 1) = 2, (1, 2) = 3, (1, 3) = 4, (1, 4) = 5, (1,
5) = 6, (1, 6) = 7, (1, 7) = 8, (1, 8) = 9, (1, 9) = 10, (1, 10) =
11, (1, 11) = 12, (2, 1) = 3, (2, 2) = 4, (2, 3) = 5, (2, 4) = 6,
(2, 5) = 7, (2, 6) = 8, (2, 7) = 9, (2, 8) = 10, (2, 9) = 11, (2,
10) = 12, (2, 11) = 13, (3, 1) = 4, (3, 2) = 5, (3, 3) = 6, (3, 4)
= 7, (3, 5) = 8, (3, 6) = 9, (3, 7) = 10, (3, 8) = 11, (3, 9) = 12,
(3, 10) = 13, (3, 11) = 14, (4, 1) = 5, (4, 2) = 6, (4, 3) = 7, (4,
4) = 8, (4, 5) = 9, (4, 6) = 10, (4, 7) = 11, (4, 8) = 12, (4, 9) =
13, (4, 10) = 14, (4, 11) = 15, (5, 1) = 6, (5, 2) = 7, (5, 3) = 8,
(5, 4) = 9, (5, 5) = 10, (5, 6) = 11, (5, 7) = 12, (5, 8) = 13, (5,
9) = 14, (5, 10) = 15, (5, 11) = 16, (6, 1) = 7, (6, 2) = 8, (6, 3)
= 9, (6, 4) = 10, (6, 5) = 11, (6, 6) = 12, (6, 7) = 13, (6, 8) =
14, (6, 9) = 15, (6, 10) = 16, (6, 11) = 17, (7, 1) = 8, (7, 2) =
9, (7, 3) = 10, (7, 4) = 11, (7, 5) = 12, (7, 6) = 13, (7, 7) = 14,
(7, 8) = 15, (7, 9) = 16, (7, 10) = 17, (7, 11) = 18, (8, 1) = 9,
(8, 2) = 10, (8, 3) = 11, (8, 4) = 12, (8, 5) = 13, (8, 6) = 14,
(8, 7) = 15, (8, 8) = 16, (8, 9) = 17, (8, 10) = 18, (8, 11) = 19,
(9, 1) = 10, (9, 2) = 11, (9, 3) = 12, (9, 4) = 13, (9, 5) = 14,
(9, 6) = 15, (9, 7) = 16, (9, 8) = 17, (9, 9) = 18, (9, 10) = 19,
(9, 11) = 20, (10, 1) = 11, (10, 2) = 12, (10, 3) = 13, (10, 4) =
14, (10, 5) = 15, (10, 6) = 16, (10, 7) = 17, (10, 8) = 18, (10, 9)
= 19, (10, 10) = 20, (10, 11) = 21, (11, 1) = 12, (11, 2) = 13,
(11, 3) = 14, (11, 4) = 15, (11, 5) = 16, (11, 6) = 17, (11, 7) =
18, (11, 8) = 19, (11, 9) = 20, (11, 10) = 21, (11, 11) = 22},
datatype = anything, storage = rectangular, order = Fortran_order,
shape = [])
```

Formatted output uses a syntax analogous as in C, or Matlab:

```
> printf("This is the integer number %d",100):
This is the integer number 100
> a,b:=3.987,4.098;
```

$$a, b := 3.987, 4.098$$

```
> printf("%5.2f + %5.2f = %5.2f",a,b,a+b):
 3.99 +  4.10 =  8.08
> printf("%5.2e + %5.2e = %5.2e",a,b,a+b):
3.99e+00 + 4.10e+00 = 8.08e+00
```

%a is a general purpose format specifier
(in particular, for printing symbolic objects):

```
> x:=1;
```

$$x := 1$$

```
> printf("%a, %a, %a",x,exp(I*z),int(y^2,y));
1, exp(I*z), 1/3*y^3
```

Example: print a Matrix

```
> A := Matrix(5,5,(i,j)->1/(i+j-1)):
> for i from 1 to 5 do
      printf("%5.2f \n",A[i,1..5])
  end do:
 1.00  0.50  0.33  0.25  0.20
 0.50  0.33  0.25  0.20  0.17
 0.33  0.25  0.20  0.17  0.14
 0.25  0.20  0.17  0.14  0.12
 0.20  0.17  0.14  0.12  0.11
```

sprintf prints to string:

```
> str := sprintf("x = %5.2f",2.2676);
```

str := "x = 2.27"

```
> str;
```

"x = 2.27"

fprintf prints to data file:

Example:

Open (new) data file in my home directory:

```
> currentdir();
```

"C:\Users\wauzinge\Documents\act\L V A\CM\Maple-VO\VO SS 2019"

```
> fd := fopen("my_data_file.dat",WRITE); # fd is file pointer
```

$$fd := 1$$

```
> fprintf(fd,"%12.8f \n",Pi); # returns number of characters
  written
```

14

```
> fclose(fd);
```

Data input (interactive or from data file): See Part V.

== end of Part III ==