```
> restart;
> ? kernelopts
> ? interface
```

ComputerMathematik - Einführung in Maple / Teil III

Winfried Auzinger (April 2009)

Hinweis:
Teil III ist angelehnt an MIPG, Kapitel 5 und 6.
Die während der VO entstehende fertige Version dieses Vortrags wird nach der VO auf der CompMath - Homepage zum Download bereitgestellt.

Die Inhalte dieser VO (Kontrollstrukturen, Prozeduren) wurden bereits in VO I+II angesprochen - hier geht es um eine detailliertere Beschreibung und genaue Regeln.

1. Ergänzungen und Vorbemerkungen; einige weitere nützliche Befehle und Datenstrukturen

1.1 ACHTUNG: Zuweisung von komplexeren Objekten ([r]tables, Arrays) mit :=

Zuweisung mit := bewirkt normalerweise, dass eine **Kopie** des betreffenden Wertes angelegt wird; die beiden Objekte sind dann unabhängig voneinander.

Beispiel:

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

$$a := 1$$

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

$$b := 1$$

```
> a:=999;
```

$$a := 999$$

```
> a,b; # a wurde verändert, b nicht!
```

$$999, 1$$

Das bedeutet: Die Neuzuweisung eines Wertes an das 'ursprüngliche' Objekt **a** hat keine Rückwirkung auf die Kopie **b** - diese ist unabhängig von a.

ANDERS ist es bei **tables** und sogenannten `rectangular Tables' (**rtables**, insbesondere **Array**, und **Matrix**, **Vector** = Spezialfälle von Array).
(Grund: Speichereffizienz)

Beispiel:

```
> A:=Array(0..2,[x,y,z]);
```

$$A := Array(0..2, \{(0) = x, (1) = y, (2) = z\}, datatype = anything, storage = rectangular, order = Fortran_order)$$

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

$$Array(0..2, \{(0) = x, (1) = y, (2) = z\}, datatype = anything, storage = rectangular, order = Fortran_order)$$

```
> B:=A;
```
$B := Array(0..2, \{(0) = x, (1) = y, (2) = z\}, datatype = anything, storage = rectangular, order = Fortran_order)$

```
> A[1]:=xxxxxxxxxx;
```
$$A_1 := xxxxxxxxxx$$

```
> A;
```
$Array(0..2, \{(0) = x, (1) = xxxxxxxxxx, (2) = z\}, datatype = anything, storage = rectangular, order = Fortran_order)$

```
> B;
```
$Array(0..2, \{(0) = x, (1) = xxxxxxxxxx, (2) = z\}, datatype = anything, storage = rectangular, order = Fortran_order)$

```
> evalb(A=B);
```
$$true$$

Man sieht: B verweist auf A - es wurde keine eigene Kopie engelegt!

Verwendung von eval hilft nicht. Falls gewünscht, muss man explizit eine **Kopie erzwingen:**

```
> ? copy
> A:=Array(0..2,[x,y,z]);
```
$A := Array(0..2, \{(0) = x, (1) = y, (2) = z\}, datatype = anything, storage = rectangular, order = Fortran_order)$

```
> B:=copy(A);
```
$B := Array(0..2, \{(0) = x, (1) = y, (2) = z\}, datatype = anything, storage = rectangular, order = Fortran_order)$

```
> A[0]:=xxxxxxxxxx;
```
$$A_0 := xxxxxxxxxx$$

```
> B;
```
$Array(0..2, \{(0) = x, (1) = y, (2) = z\}, datatype = anything, storage = rectangular, order = Fortran_order)$

```
> evalb(A=B);
```
$$false$$

VORSICHT: Es gibt ähnliche Nebeneffekte bei der Übergabe von tables, Arrays an Prozeduren, d.h. es wird nur ein Pointer auf das Objekt übergeben (`call by reference')

Dies wirkt sich jedoch nur aus, falls das Argument in der Prozedur verändert wird was man in Maple eher nicht tun sollte (Ausgabewerte nur explizit mittels return setzen!). Ausnahme: Speicherplatz muss gespart werden .... aber dann sollte man genau wissen, was man tut.

Weniger fehleranfällig: globale Variablen verwenden (haben allerdings fixe Namen).

1.2 Zuweisung von Werten mittels assign:

```
> a:='a': assign(a=3); a;
  solve({x+1=a,x-y=1},{x,y});
```
$$3$$
$$\{x = 2, y = 1\}$$

```
> assign(%);
> x,y;
```
$$2, 1$$

1.3 **Ersetzen von Variablen oder Teilausdrücken mittels subs bzw. algsubs:**

```
> x:='x':expr:=x^2+1; y:=5;
```
$$expr := x^2 + 1$$
$$y := 5$$

```
> subs(x=y,expr);
```
$$26$$

```
> a:='a':expr:=a^2+b^2+c^2;
```
$$expr := a^2 + 1 + c^2$$

```
> subs(b^2+c^2=1,expr);
```
$$a^2 + 1 + c^2$$

```
> algsubs(b^2+c^2=1,expr);
```
$$a^2 + 1$$

```
> i:=1;
```
$$i := 1$$

1.4 Extraktion von Koeffizienten (z.B. bei Polynomen) mittels coeff:

```
> p:=x->sum((i+1)*(x-1)^i,'i'=0..5);
```
$$p := x \rightarrow \sum_{'i'=0}^{5} (i+1)\,(x-1)^i$$

```
> coeff(p(x),x,0);
```
$$-12$$

```
> coeff(p(x),x,5);
```
$$0$$

```
> expand(p(x));
```
$$12\,x - 12$$

1.5 Der Differentiationsoperator D:

Im Gegensatz zu diff bildet D eine Funktion auf ihre Ableitungs**funktion** ab, d.h.:

```
> x:='x': y:='y':D(sin);
```
$$\cos$$

```
> D(x->x^2);%(y);
```
$$x \rightarrow 2\,x$$
$$2\,y$$

```
> (D@@2)(sin)(x); # 2.Ableitung (funktionale Potenz von D)
```
$$-\sin(x)$$

```
> phi:=(x,y)->x^2*y^3;
```
$$\phi := (x, y) \rightarrow x^2\,y^3$$

```
> D[1](phi), D[2](phi); # partielle Ableitungen
```
$$(x, y) \rightarrow 2\,x\,y^3,\ (x, y) \rightarrow 3\,x^2\,y^2$$

```
> D[1,2](phi); # gemischte 2. partielle Ableitung
```

$(x, y) \to 6\,x\,y^2$

1.6 Weitere vordefinierte Datenstrukturen:

Stapelspeicher (**stack**) und Warteschlange (**queue**),
implementiert über tables

STACK = Linearer Speicher, zugriff nach dem LIFO - Prinzip
(`Last In - First Out')

```
> ? stack
> S:=stack[new](); whattype(eval(S)); is(S,stack);
```

$S := table([0 = 0])$

table

true

```
> stack[push](unterstes_Element,S); stack[push](naechstes_Element,S)
  ;
```

unterstes_Element

naechstes_Element

```
> eval(S); # S[0] ist aktuelle Länge;
```

$table([0 = 2, 1 = unterstes_Element, 2 = naechstes_Element])$

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

2

```
> seq(S[i],i=1..stack[depth](S)); # Folge der Elemente
```

unterstes_Element, naechstes_Element

```
> stack[top](S); # 'peek'
```

naechstes_Element

```
> stack[pop](S); stack[top](S);
```

naechstes_Element

unterstes_Element

```
> stack[pop](S); stack[top](S);
```

unterstes_Element

```
Error, (in stack:-top) empty stack
```

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

true

QUEUE = Linearer Speicher, zugriff nach dem FIFO - Prinzip
(`First In - First Out'; Warteschlange)

```
> ? queue
> Q:=queue[new](); whattype(eval(Q)); is(Q,queue);
```

$Q := table([0 = 0])$

table

true

```
> queue[enqueue](Q,Erster_an_der_Kassa);
```

Erster_an_der_Kassa

```
> queue[enqueue](Q,Zweiter_an_der_Kassa);
```

Zweiter_an_der_Kassa

```
> eval(Q);
```

$$table([0 = 2, 1 = Erster_an_der_Kassa, 2 = Zweiter_an_der_Kassa])$$

```
> queue[length](Q);
```

$$2$$

```
> queue[front](Q);
```

$$Erster_an_der_Kassa$$

```
> abgefertigt:=queue[dequeue](Q);
```

$$abgefertigt := Erster_an_der_Kassa$$

```
> seq(Q[i],i=1..length(Q));
```

$$Zweiter_an_der_Kassa$$

Siehe auch: **HEAP**: Warteschlange mit Prioritäten (interne Implementierung komplizierter).

```
> ? heap
```

2. Ablaufsteuerung (Kontrollstrukturen) im Detail

Anwendung in Prozeduren (aber nicht nur dort);
wir illustrieren die diversen Konstrukte im Detail anhand von Beispielen.

2.1 Bedingte Anweisungen (if)

```
> primecheck:=proc(p::posint)
> local is_prime, is_mersenne;
> is_prime:=is(p,prime);
> is_mersenne:=is(log[2](p+1),posint);
> if is_prime then
>    printf ("%d ist prim\n",p);
>    if is_mersenne then
>       printf ("%d ist auch Mersenne\n",p);
     else
        printf ("%d ist aber nicht Mersenne\n",p);
>    end if
> elif is_mersenne then
>    printf ("%d ist Mersenne, aber nicht prim\n",p);
> else
>    printf ("%d ist weder Mersenne noch prim\n",p);
> end if
> end proc:
```

```
> primecheck(2);
2 ist prim
2 ist aber nicht Mersenne
```

```
> primecheck(3);
3 ist prim
3 ist auch Mersenne
```

```
> primecheck(4);
4 ist weder Mersenne noch prim
```

```
> primecheck(2047);

2047 ist Mersenne, aber nicht prim
```

Anmerkungen:
-- Auch mehrere elif-Zweige möglich
-- 'case'-Konstrukt gibt es nicht
-- FAIL bedeutet weder true noch false! (unentscheidbar)

```
> a:=FAIL;
```

$$a := FAIL$$

```
> if a=true then ja
  elif a=false then nein
  else "unentscheidbar"
  end if;
```

"unentscheidbar"

Kompaktere Form der if-Klausel (mit `if`)

```
> a:=2; `if`(a=1,eins,ungleich_eins);
```

$$a := 2$$

$$ungleich_eins$$

2.2 Wiederholung (Schleifen)

```
> n:='n': # das geht natürlich nicht
  for i from 1 to n do
    printf("%d ",i);
  end do;
Error, final value in for loop must be numeric or character
```

```
> n:=5;
```

$$n := 5$$

```
> for i from 1 to n do
    printf("%d ",i^2); # formatierte Ausgabe
                      # Syntax ähnlich wie in C, Matlab
  end do;
1 4 9 16 25
```

```
> i; # letzter Wert aus Schleife + 1
```

$$6$$

Allgemeinere Varianten:

```
> for i from n to 1 by -2 do
    printf("%d ",i);
  end do;
5 3 1
```

```
> for j from 1 to 100 by 3 while j^2<500 do
>     printf("%d %d \n",j,j^2);
> end do;
1 1
4 16
7 49
10 100
13 169
16 256
19 361
22 484
```

'Reine' while-Schleife (abweisende Schleife):

```
> k:=200:
  while not is(k,prime) do
     print(k);
     k:=k+1;
  end do:
```

200

201

202

203

204

205

206

207

208

209

210

Spezielle 'aufzählende' Varianten von for:

```
> for i in seq(i,i=5..1,-1) do i end do;
```

5

4

3

2

1

```
> for Tier in {Hund,Katze,Maus} do Tier end do;
```

Hund

Katze

Maus

'Endlos'-Schleife, Abbruch mittels break:

```
> do
    x:=rand(); # Zufalls-Integer
    if is(x/3,even) then break end if;
  end do;
```

$x := 395718860534$

$x := 193139816415$

$x := 22424170465$

$x := 800187484459$

$x := 427552056869$

$x := 842622684442$

$x := 412286285840$

$x := 996417214180$

$x := 386408307450$

next-Anweisung (zurück zu do und weiter):

```
> do
```

```
      x:=rand();
      if is(x,even) then
         next
      else
         print(x);
         break
      end if
   end do:
```
694607189265

2.3 *Befehle, die implizit Schleifen beinhalten*

```
> map(x->x^2,[1,2,3,4]);
```
$[1, 4, 9, 16]$

```
> y:='y': map(diff,[sin(y),cos(y),tan(y)],y$2);
```
$\left[-\sin(y), -\cos(y), 2\tan(y)\left(1+\tan(y)^2\right)\right]$

select / remove - Mechanismus:

```
> select(isprime,[$1..10]); # $1..n ist Abkürzung für
                            # seq(i,i=1..n)
```
$[2, 3, 5, 7]$

```
> remove(isprime,[$1..10]);
```
$[1, 4, 6, 8, 9, 10]$

```
> selectremove(isprime,[$1..10]);
```
$[2, 3, 5, 7], [1, 4, 6, 8, 9, 10]$

'Reissverschluss (zip)':

```
> zip((x,y)->x*y,[1,2,3],[alpha,beta,delta]);
```
$[\alpha, 2\beta, 3\delta]$

seq - Konstrukt:

```
> seq(k^2,k=1..10);
```
$1, 4, 9, 16, 25, 36, 49, 64, 81, 100$

add, sum etc. (man beachte den Unterschied!)

```
> add(k^2+z,k=1..10);
```
$385 + 10\,z$

```
> sum('k^2+z','k'=1..10);
```
$385 + 10\,z$

```
> add(1/l,l=1..infinity);
Error, unable to execute add
> sum('1/l',l=1..infinity);
```
∞

```
> mul(i,i=1..5);
```
120

```
> product('k','k'=1..N);
```
$\Gamma(N+1)$

Oder auch so:

```
> add(i,i=[1,3,5,7]);
```

3. Prozeduren im Detail

```
> restart;
> ? procedure
```

3.1 Definition von Prozeduren, Komponenten

```
> p:=proc(FORMALE_PARAMETER)
> local LOKALE_VARIABLEN;
> global GLOBALE_VARIABLEN;
> options OPTIONEN;
> description "Beschreibung";
> # procedure body...
> end proc;
```

p := **proc**(*FORMALE_PARAMETER*)

option *OPTIONEN*;

local *LOKALE_VARIABLEN*;

global *GLOBALE_VARIABLEN*;

description "Beschreibung";

end proc

WICHTIG: Auswertungsregel für aktuelle Parameter:

-- Am BEGINN der Ausführung einer Prozedur werden die aktuellen Parameter ausgewertet und für die formalen Parameter eingesetzt (kopiert - **call by value**), ggfs. mit Überprüfung ihres Typs.

-- Es können auch aktuelle Parameter beim Aufruf hinten weggelassen werden (z.B. nur 2 statt 3); dies ist aber nur korrekt, wenn auf die fehlenden Parameter bei der Ausführung nicht zugegriffen wird. Dies stellt eine (eingeschränkte) Möglichkeit dar, optionale Parameter zu realisieren.

Beispiel:

```
> p:=proc(x,y)
  description "gebe zurück x oder x*y (falls x<0) ";
  if (x>=0) then
     x
  else
     x*y
  end if
  end proc:
> Describe(p);

# gebe zurück x oder x*y (falls x<0)
p( x, y )

> p(1,2);
```

1

```
> p(1); # korrekter Aufruf
```

1

```
> p(-1); # inkorrekt
Error, invalid input: p uses a 2nd argument, y, which is missing
```

```
> p(-1,2); # korrekt
```

-2

```
> p(-1,2,3); # überschüssige Parameter werden ignoriert
```

-2

'Echte' optionale Parameter kann man z.B. realisieren, indem man einen einzigen Parameter als Liste angibt; die aktuelle Anzahl ist dann die aktuelle Länge der übergebenen Liste. Dann funktioniert aber das automatische type-checking nicht, und auch Default-Werte müssen explizit einprogrammiert werden (ein bisschen umständlich)

```
> p:=proc(parameters::list)
> # local l:=nops(parameters); das geht nicht
  local l, lparam;
  l:=nops(parameters);
  lparam:=parameters; # lokale Kopie der Parameter
  printf ("%d Parameter übergeben:\n",l);
  printf ("%a ",parameters);
  if (l>=1 and not type(parameters[1],integer)) then
     error "First parameter must be integer"
  end if;
  if l=1 then lparam:=[op(lparam),default2] end if;
  end proc;
```

$p :=$ **proc**(*parameters*::*list*)

local *l*, *lparam*;

l := *nops*(*parameters*);

lparam := *parameters*;

printf ("%d Parameter übergeben:\n", *l*);

printf ("%a ", *parameters*);

if $1 <= l$ **and not** *type*(*parameters*[1], *integer*) **then error** "First parameter must be integer"

end if;

if $l = 1$ **then** *lparam* := [*op*(*lparam*), *default2*] **end if**

end proc

```
> p(); p(x);
Error, invalid input: p uses a 1st argument, parameters (of type list), which is missing
Error, invalid input: p expects its 1st argument, parameters, to be of type list, but received x
```

```
> p([]);
0 Parameter übergeben:
[]
```

```
> p([1]);
1 Parameter übergeben:
[1]
```

$$[1, default2]$$

```
> p(['x']);
1 Parameter übergeben:
[x]
Error, (in p) First parameter must be integer
```

```
> p([1,2,3,4]);
4 Parameter übergeben:
[1, 2, 3, 4]
```

Seit Maple 10 gibt es auch einen systematischen Mechanismus für optionale Parameter und ihre Default-Werte (später).

Type-Checking (bereits erwähnt) - geht auch für Mengen von Typen:

```
> p:=proc(x::{posint,negint})
  x
  end proc;
```

$$p := \mathbf{proc}(x::\{negint, posint\})\ \ x\ \ \mathbf{end\ proc}$$

```
> p(-1); p(0); p(1);
```

$$-1$$

```
Error, invalid input: p expects its 1st argument, x, to be of type {negint, posint}
, but received 0
```

$$1$$

ZU BEACHTEN:
-- Formale Parameter kann man normalerweise **nicht** als lokale Variablen
 oder zur Rückgabe von Werten verwenden.

```
> p:=proc(x)
> x:=2;
> end proc;
```

$$p := \mathbf{proc}(x)\ \ x := 2\ \ \mathbf{end\ proc}$$

```
> p(2);
Error, (in p) illegal use of a formal parameter
```

Typdeklaration von lokalen Variablen oder des Ergebniswertes: entspricht einer 'assume'-Klausel.

Die Überprüfung ist aber nur aktiv, wenn **kernelopts(assertlevel=2)** aktiviert wurde:

```
> p:=proc()::integer;
  local s::set;
  s:=[infinity,0];
  end proc;
```

$$p := \mathbf{proc}(\ )::integer;\ \ \mathbf{local}\ s::set;\ s := [infinity, 0]\ \ \mathbf{end\ proc}$$

```
> p();
```

$$[\infty, 0]$$

```
> ? kernelopts
```

```
> kernelopts(assertlevel);
```

$$0$$

```
> p();
```

$$[\infty, 0]$$

```
> kernelopts(assertlevel=2);
```

$$0$$

```
> p();
Error, (in p) assertion failed in assignment, expected set, got [infinity, 0]
> kernelopts(assertlevel=0);
```

$$2$$

GLOBALE Variablen können auch erst innerhalb der Prozedur 'erzeugt' werden; man verwendet diese normalerweise für 'große Datencontainer', die z.B. von mehreren Prozeduren gemeinsam verwendet werden sollen. (Vermeidung von unnötigen Kopien im Speicher, die durch call by value angelegt würden)

Beispiel:

```
> restart;
> generate_global_stack_S:=proc()
> global S;
> S:=stack[new]();
> end proc;
```

$$generate_global_stack_S := \textbf{proc}()\ \textbf{global}\ S;\ S := stack[new]()\ \textbf{end proc}$$

```
> generate_global_stack_S():
> type(S,stack), eval(S);
```

$$true, table([0 = 0])$$

```
> push:=proc(element)
> global S;
> stack[push](element,S);
> end proc;
```

$$push := \textbf{proc}(element)\ \textbf{global}\ S;\ stack[push](element, S)\ \textbf{end proc}$$

```
> push(1);
```

$$1$$

```
> eval(S);
```

$$table([0 = 1, 1 = 1])$$

EIN TRICK:
Wie können wir den **Namen** des globalen Stacks **variabel** machen?
Die **global** Deklaration bietet hier keine geeignete Flexibilität.

Aber: Wenn man einen Namen 'bastelt', entsteht damit automatisch eine **globale** Variable, auch innerhalb einer Prozedur

```
> cat(a,b); type(%,name);
```

$$ab$$

$$true$$

```
> cat(a,b):=1;
```

$$ab := 1$$

```
> a||b; # ist äquivalent
```

$$1$$

```
> convert("kloing",name); # oder auch so
```
kloing

Jetzt verwenden wir das in unserer Prozedur. Die globale Variable wird mit dem durch den Parameter stack_name (=string) spezifizierten Namen durch die Zuweisung eines Wertes implizit erzeugt.

Für die Zuweisung muss man hier aber **assign** verwenden (mit := erhält man einen Syntaxfehler).

```
> generate_global_stack:=proc(stack_name::string)
  assign(convert(stack_name,name),stack[new]());
> end proc;
```
generate_global_stack := **proc**(*stack_name*::*string*)

assign(*convert*(*stack_name*, *name*), *stack*[*new*]( ))

end proc

```
> generate_global_stack("mystack");
> type(mystack,stack);eval(mystack);
```
true

table([0 = 0])

```
> stack[push](1,mystack); eval(mystack);
```
1

table([0 = 1, 1 = 1])

EINE WICHTIGE AUSWERTUNGSREGEL !!!! -->

Lokale Variablen in Prozeduren werden wie nicht wie üblich voll ausgewertet, sondern es erfolgt (aus Effizienzgründen) nur eine '1-level' evaluation.
Beispiel:

```
> p:=proc()
> local l1,l2,l3;
> l1:=l2;
> l2:=1;
> l1;
> end proc:
> p();
```
l2

```
> # ABER- außerhalb:
> l1:=l2: l2:=1: l1;
```
1

In der Praxis hat dies wenig Bedeutung, weil ein derartiges \`Rückwärtseinsetzen' ohnehin **schlechter Programmierstil** ist (unübersichtlich, fehleranfällig).

Folgendes funktioniert problemlos
(bei jeder Zuweisung wird **sofort** ausgewertet):

```
> p:=proc()
> local l1,l2,l3,l4;
> l1:=555;
> l2:=l1;
> l3:=l2;
```

```
> l4:=l3;
> end proc:
> p();
```

555

```
> restart;
```

OPTIONEN in Prozeduren: (nur das wesentliche):

* Option **Copyright** :

option als Name, der mit dem Wort Copyright beginnt
(etwas verquere Syntax):

```
> p:=proc()
> options `Copyright my copyright`;
> printf("p");
> end proc;
```

$p :=$ **proc**( ) ... **end proc**

```
> interface(verboseproc);
```

1

```
> interface(verboseproc=2);
```

1

```
> eval(p);
```

proc( ) **option** *Copyright my copyright*; *printf* ("p") **end proc**

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

* Option **remember (!)** :

Generiert automatisch eine sogenannte **REMEMBER TABLE,**
d.h. Maple protokolliert alle Aufrufe (übergebene Parameter + resultierende Werte)
in einer internen Tabelle. Bei wiederholtem Aufruf wird auf die bereits
vorhandenen Tabellenwerte zurückgegriffen!

... Ist zum Beispiel wichtig bei komplexeren Rekursionen!

BEISPIEL: Rekursive Berechnung der Fibonacci-Zahlen (3-Term-Rekursion)

```
> fibonacci:=proc(n) # REKURSIVE PROZEDUR!
> if (n<2) then
>     n
> else
>     fibonacci(n-1)+fibonacci(n-2)
> end if
> end proc:
```

Wie messen jetzt mit der Stoppuhr **time()** die verbrauchte CPU-Zeit (in Sekunden)

```
> start:=time():
  fibonacci(30);
  time()-start;
```

832040

1.859000000000000000000

Das macht man besser mit Iteration! - oder wenn schon rekursiv, dann
mit option remember (erfordert in n linearen zusätzlichen Speicher
für die interne remember-table):

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

fibonacci := **proc**(*n*)

option *remember*;

if $n < 2$ **then** n **else** $fibonacci(n-1) + fibonacci(n-2)$ **end if**

end proc

```
> start:=time():
  fibonacci(30);
  time()-start;
```

832040

0.0000000000000000000

```
> interface(verboseproc=3):
> eval(fibonacci); # jetzt sieht man Inhalt
                   # der remember-table
```

proc(*n*)

option *remember*;

if $n < 2$ **then** n **else** $fibonacci(n-1) + fibonacci(n-2)$ **end if**

end proc#(0) = 0#(1) = 1#(2) = 1#(3) = 2#(5) = 5#(4) = 3#(7) = 13#(6) = 8#(10) = 55#(11) = 89#(8) = 21#(9) = 34#(15) = 610#(14) = 377#(13) = 233#(12) = 144#(21) = 10946#(20) = 6765#(23) = 28657#(22) = 17711#(16) = 987#(17) = 1597#(18) = 2584#(19) = 4181#(30) = 832040#(29) = 514229#(28) = 317811#(26) = 121393#(27) = 196418#(24) = 46368#(25) = 75025

Der Inhalt der remember-Table bleibt permanent gespeichert und wird von weiteren Aufrufen wieder verwendet, es sei denn, man definiert die Prozedur neu.

* Option **trace** :

Protokolliert Aufrufe. Wir betrachten z.B. wieder die ursprüngliche Version von fibonacci:

```
> fibonacci:=proc(n) option trace;
> if (n<2) then
>    n
> else
>    fibonacci(n-1)+fibonacci(n-2)
> end if
> end proc:
> fibonacci(4);
```

```
{--> enter fibonacci, args = 4
{--> enter fibonacci, args = 3
{--> enter fibonacci, args = 2
{--> enter fibonacci, args = 1
```

1

```
<-- exit fibonacci (now in fibonacci) = 1}
{--> enter fibonacci, args = 0
```

```
0
<-- exit fibonacci (now in fibonacci) = 0}
1
<-- exit fibonacci (now in fibonacci) = 1}
{--> enter fibonacci, args = 1
1
<-- exit fibonacci (now in fibonacci) = 1}
2
<-- exit fibonacci (now in fibonacci) = 2}
{--> enter fibonacci, args = 2
{--> enter fibonacci, args = 1
1
<-- exit fibonacci (now in fibonacci) = 1}
{--> enter fibonacci, args = 0
0
<-- exit fibonacci (now in fibonacci) = 0}
1
<-- exit fibonacci (now in fibonacci) = 1}
3
<-- exit fibonacci (now at top level) = 3}
3
```

-- Description: Kann auch mehrzeilig sein,
dazu gibt man eine Folge von Strings an.

```
> restart;
> p:=proc()
> description "description 1.Zeile",
>             "description 2.Zeile";
> end proc:
> Describe(p);

# description 1.Zeile
# description 2.Zeile
p( )
```

3.2 Interpretation und Ausführung von Prozeduren

Grundregel: Die üblichen automatischen Vereinfachungen werden bei der Definition einer Prozedur angewendet.

Beispiel 1:

```
> p:=proc() 4/6 end proc;
```

$p :=$ **proc**( ) $2/3$ **end proc**

```
> Digits;
```

30

Beispiel 2: Vorsicht - float-Konstante wird automatisch gerundet!

```
> p:=proc()
```

```
> 1/1.000000000000000000000000000000000000001
> end proc;
```

$p := \mathbf{proc}(\ )\ 1/1.00000000000000000000\ \mathbf{end\ proc}$

```
> p(); evalb(%=1);
```

$$1.00000000000000000000$$

$$true$$

Rückgabe von Werten: entweder das Resultat des letzten ausgeführten Befehls, oder (sauberer) explizite Angabe mittels **return** (an beliebig vielen Stellen). return beendet den Ablauf der Prozedur.

Im Fall eines Abbruchs mit **error** wird die betreffende Meldung ausgegeben, und der Rückgabewert ist nicht definiert.

Beispiel:

```
> p:=proc(x)
> if (x<1) then
>    return x,x^2 # Ausgabe einer exprseq
> else
>    error "Error!"
> end if
> end proc:
> resultat:=p(1/2);
```

$$resultat := \frac{1}{2}, \frac{1}{4}$$

```
> unassign('resultat');resultat;
```

$$resultat$$

```
> resultat:=p(2);
Error, (in p) Error!
> resultat;
```

$$resultat$$

Es gibt auch einen **try ... catch** Mechanismus zum kontrollierten 'Abfangen' von Fehlern - später.

Ein verbreiteter Fehler: Man verlangt 'vorzeitige' Auswertung von Ausdrücken, wie in folgendem Beispiel:

```
> mymax:=proc(x,y)
>   if (x<y) then y else x end if
> end proc;
```

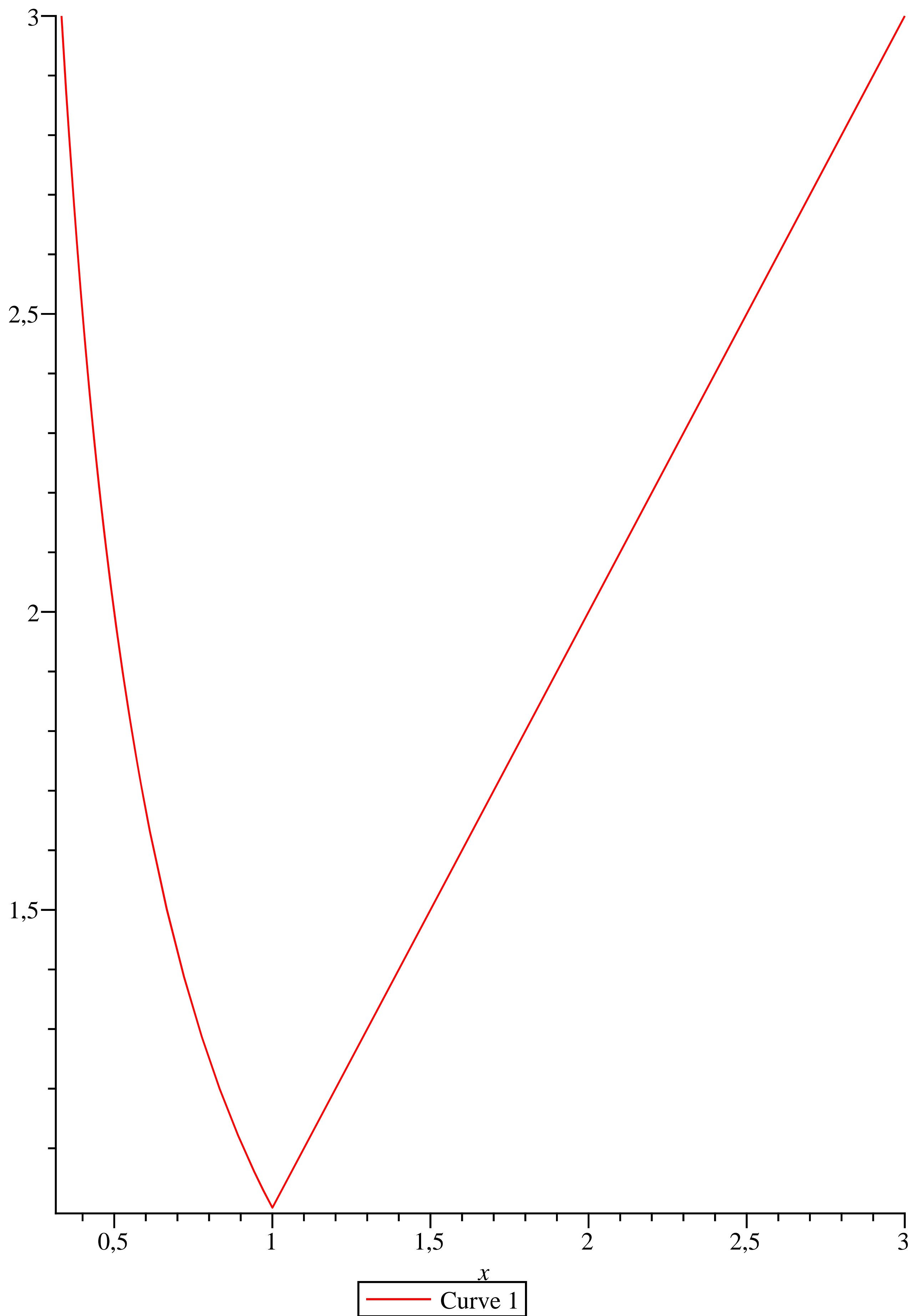
$mymax := \mathbf{proc}(x, y)\ \mathbf{if}\ x < y\ \mathbf{then}\ y\ \mathbf{else}\ x\ \mathbf{end\ if}\ \mathbf{end\ proc}$

Hier wird versucht, mymax auszuwerten, bevor Werte eingesetzt werden:

```
> plot(mymax(x,1/x),x=1/3..3);
Error, (in mymax) cannot determine if this expression is true or false: x < 1/x
```

Mit `Maskierung' (verzögerter Auswertung) funktioniert es:

```
> plot('mymax(x,1/x)',x=1/3..3);
```

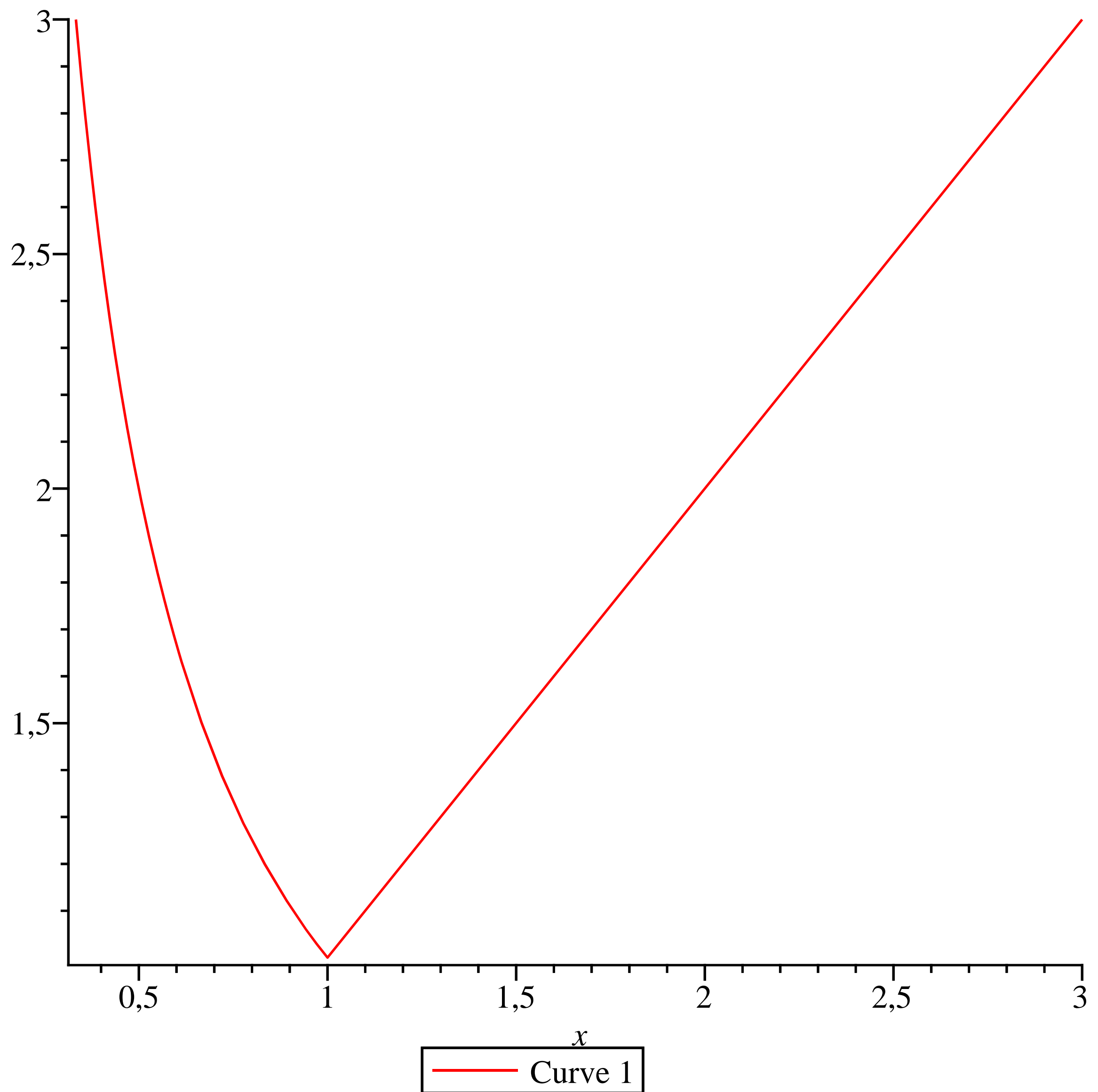


Man kann auch die Prozedur flexibler schreien:
Falls mymax nicht mit numerischen Werten aufgerufen wird,
wird die Auswertung verzögert (nicht immer leicht zu handhaben).
Typisches Problem, das bei Auswertung von 'stückweise' definierten
Funktionen auftritt.

```
> mymax:=proc(x,y)
> if (type(x,numeric) and type(y,numeric)) then
>    if (x<y) then y else x end if
> else
>    'mymax'(x,y)
> end if
> end proc:
> mymax(1,2), mymax(x,y);
```

$$2,\ mymax(x, y)$$

```
> plot(mymax(x,1/x),x=1/3..3);
```



3.3 Dokumentation von Prozeduren, externe Speicherung

Help Pages: Kann man auch selbst generieren, siehe

```
> ? helppages
```

Speicherung / Rücklesen von Prozeduren als **externe Textdateien:**

```
> p:=proc() 1 end proc;
```

$$p := \mathbf{proc}(\ )\ 1\ \ \mathbf{end\ proc}$$

```
> currentdir(); # bzw. setzen mit:
```

"C:\"

```
> currentdir("C:/"): currentdir(); # nicht \ !
```
"C:\"

```
> save p, "pfile.m";
> restart;
> read "pfile.m"; eval(p);
```
proc( ) 1 **end proc**

Damit kann man auch Prozeduren einlesen, die mit einem externen Texteditor erstellt wurden. Nach dem Einlesen mit read kann man sie allerdings nicht innerhalb des aktuellen Worksheets editieren.

3.4 Last name evaluation

Ein weiterer Sonderfall bei Auswertung:

Prozeduren (und tables) werden nicht voll ausgewertet, sondern nur dem Namen nach. Volle Auswertung nur mittels **eval:**

```
> p; eval(p);
```
p

proc( ) 1 **end proc**

3.5 Alternative Syntaxvarianten für Prozeduren

* Funktionale Notation:

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

```
> f:=(x::integer,y)->if  x<0 then x else y end if;
```
$f := (x::integer, y) \rightarrow \textbf{if } x < 0 \textbf{ then } x \textbf{ else } y \textbf{ end if}$

```
> type(f,procedure);
```
true

* **unapply**: wandelt Ausdruck in Funktion um

```
> x,y:='x','y';
  p:=unapply(x^2+y^2,x,y);
```
$x, y := x, y$

$p := (x, y) \rightarrow x^2 + y^2$

* Funktionale Komposition:

```
> f:=D(exp+ln);
```
$f := \exp + \left(z \rightarrow \frac{1}{z}\right)$

```
> f(1);
```
$\mathrm{e} + 1$

3.6 Variable Parameterlisten

Mittels des speziellen Namen **args** kann man flexible Listen unbenanner Parameter realisieren. Beispiel:

```
> p:=proc()
>   local i;
```

```
>   printf("%d / ",nargs); # Anzahl übergebene Parameter
>   for i from 1 to nargs do
        printf("%a ",args[i])
    end do;
> end proc:
> p(); p(a); p(a,b);
0 / 1 / a 2 / a b
```

3.7 Beispiele für Prozeduren

Im folgenden einige weitere Beispiele, die die Verwendung verschiedener Datenstrukturen zeigen bzw. die algoririthmisch oder mathematisch interessant sind:

BEISPIEL: Eine Prozedur, die entweder einen Differenzenquotienten oder einen Differentialquotienten auswertet (je nach Aufruf):

```
> restart;
> diffq:=proc(f::procedure,x::anything,y::anything)
  local h;
> if x<>y then
>   return simplify(f(x)-f(y))/(x-y)
> else
>   return simplify(limit((f(x+h)-f(x))/h,h=0))
> end if
> end proc:
> diffq(x->x^3,x,y);
```

$$\frac{x^3 - y^3}{x - y}$$

```
> diffq(sin,x,x);
```

$$\cos(x)$$

```
> diffq(cos,1,1);
```

$$-\sin(1)$$

BEISPIEL f ür eine rekursive Prozedur: **sortmerge**
eine Variante von **Sort/Merge**
Übernimmt Liste und liefert sortierte Liste, verwendet **merge** zum Mischen

```
> sortmerge:=proc(liste::list)
> local i,l,n,n1,n2,s1,s2;
  n:=nops(liste);
  if (n=0) then return [] end if;
  s1:=[]; s2:=[];
> n1:=trunc(n/2); n2:=n-n1;
> if (n1=1) then
     s1:=[liste[1]]
  elif (n1>1) then
     s1:=sortmerge(liste[1..n1])
  end if;
> if (n2=1) then
     s2:=[liste[n1+1]]
  elif (n2>1) then
```

```
      s2:=sortmerge(liste[n1+1..n])
   end if;
   return merge(s1,s2);
   end proc;
```

sortmerge := **proc**(*liste*::*list*)

local *i*, *l*, *n*, *n1*, *n2*, *s1*, *s2*;

n := *nops*(*liste*);

if $n = 0$ **then return** [ ] **end if**;

s1 := [ ];

s2 := [ ];

$n1 := \mathrm{trunc}(1/2 * n)$;

$n2 := n - n1$;

if $n1 = 1$ **then** $s1 := [liste[1]]$ **elif** $1 < n1$ **then** $s1 := sortmerge(liste[1..n1])$ **end if**;

if $n2 = 1$ **then** $s2 := [liste[n1+1]]$ **elif** $1 < n2$ **then** $s2 := sortmerge(liste[n1+1..n])$ **end if**;

return *merge*(*s1*, *s2*)

end proc

```
> merge:=proc(l1::list,l2::list)
   local i,i1,i2,m,n1,n2;
   i1:=1; i2:=1;
   n1:=nops(l1); n2:=nops(l2);
   m:=[];
   while (i1<=n1 or i2<=n2) do
      if (i1>n1) then
         m:=[op(m),op(l2[i2..n2])];
         break;
      end if;
      if (i2>n2) then
         m:=[op(m),op(l1[i1..n1])];
         break;
      end if;
      if (l1[i1]<l2[i2])then
         m:=[op(m),l1[i1]];
         i1:=i1+1;
      else
         m:=[op(m),l2[i2]];
         i2:=i2+1;
      end if;
   end do;
   return m;
   end proc;
```

merge := **proc**(*l1*::*list*, *l2*::*list*)

local *i*, *i1*, *i2*, *m*, *n1*, *n2*;

i1 := 1;

i2 := 1;

n1 := *nops*(*l1*);

```
    n2 := nops(l2);
    m := [ ];
    while i1 <= n1 or i2 <= n2 do
        if n1 < i1 then m := [op(m), op(l2[i2..n2])]; break end if;
        if n2 < i2 then m := [op(m), op(l1[i1..n1])]; break end if;
        if l1[i1] < l2[i2] then
            m := [op(m), l1[i1]]; i1 := i1 + 1
        else
            m := [op(m), l2[i2]]; i2 := i2 + 1
        end if
    end do;
    return m
end proc
```

```
> sortmerge([2,1,3,5,2,1,9,0,-1,8,-11]);
```

$$[-11, -1, 0, 1, 1, 2, 2, 3, 5, 8, 9]$$

```
> sort([2,1,3,5,2,1,9,0,-1,8,-11]); # Maple-sort
```

$$[-11, -1, 0, 1, 1, 2, 2, 3, 5, 8, 9]$$

ACHTUNG: Interne Prozeduren sind in Maple zwar im Prinzip möglich, aber sehr unüberschaubar zu handhaben.

Besser: Nicht verwenden! (obiges merge ist extern zu sortmerge)

```
> restart;
```

BEISPIEL für eine (nicht rekursive) Prozedur,
die für eine gegebene Funktion f(x,y)
(repräsentiert z.B. eine endliche Gruppenoperation) eine Wertetabelle erstellt
(als Array), wobei die relevanten Argumente in einer Liste übergeben werden

```
> Wertetabelle:=proc(f,x)
> local i,j,n,werte;
> n:=nops(x);
  werte:=Array(0..n,0..n);
  werte[0,0]:='f'; # Name der Funktion!
  if n=0 then return werte end if;
  for i from 1 to n do   # 0-te Zeile bzw. 0-te Spalte:
     werte[i,0]:=x[i];   # Argumentwerte eintragen
     werte[0,i]:=x[i];
  end do;
> for i from 1 to n do # Funktionswerte eintragen
     for j from 1 to n do
        werte[i,j]:=f(x[i],x[j])
     end do
  end do;
  return werte;
> end proc;
```

$$
\begin{aligned}
&\mathit{Wertetabelle} := \mathbf{proc}(f, x)\\
&\quad \mathbf{local}\ i, j, n, \mathit{werte};\\
&\quad n := \mathit{nops}(x);\\
&\quad \mathit{werte} := \mathit{Array}(0..n, 0..n);\\
&\quad \mathit{werte}[0, 0] := 'f';\\
&\quad \mathbf{if}\ n = 0\ \mathbf{then\ return}\ \mathit{werte}\ \mathbf{end\ if};\\
&\quad \mathbf{for}\ i\ \mathbf{to}\ n\ \mathbf{do}\ \mathit{werte}[i, 0] := x[i];\ \mathit{werte}[0, i] := x[i]\ \mathbf{end\ do};\\
&\quad \mathbf{for}\ i\ \mathbf{to}\ n\ \mathbf{do\ for}\ j\ \mathbf{to}\ n\ \mathbf{do}\ \mathit{werte}[i, j] := f(x[i], x[j])\ \mathbf{end\ do\ end\ do};\\
&\quad \mathbf{return}\ \mathit{werte}\\
&\mathbf{end\ proc}
\end{aligned}
$$

```
> f:=(a,b)->trunc(10*sin(a)+cos(b));
```

$$f := (a, b) \rightarrow \mathrm{trunc}(10 \sin(a) + \cos(b))$$

```
> w:=eval(Wertetabelle(f,[eins,zwei]));
```

$w := \mathit{Array}(0..2, 0..2, \{(0, 0) = f, (0, 1) = \mathit{eins}, (0, 2) = \mathit{zwei}, (1, 0) = \mathit{eins}, (1, 1) = \mathrm{trunc}(10 \sin(\mathit{eins}) + \cos(\mathit{eins})), (1, 2) = \mathrm{trunc}(10 \sin(\mathit{eins}) + \cos(\mathit{zwei})), (2, 0) = \mathit{zwei}, (2, 1) = \mathrm{trunc}(10 \sin(\mathit{zwei}) + \cos(\mathit{eins})), (2, 2) = \mathrm{trunc}(10 \sin(\mathit{zwei}) + \cos(\mathit{zwei}))\}, \mathit{datatype} = \mathit{anything}, \mathit{storage} = \mathit{rectangular}, \mathit{order} = \mathit{Fortran_order})$

```
> print(w);
```

$\mathit{Array}(0..2, 0..2, \{(0, 0) = f, (0, 1) = \mathit{eins}, (0, 2) = \mathit{zwei}, (1, 0) = \mathit{eins}, (1, 1) = \mathrm{trunc}(10 \sin(\mathit{eins}) + \cos(\mathit{eins})), (1, 2) = \mathrm{trunc}(10 \sin(\mathit{eins}) + \cos(\mathit{zwei})), (2, 0) = \mathit{zwei}, (2, 1) = \mathrm{trunc}(10 \sin(\mathit{zwei}) + \cos(\mathit{eins})), (2, 2) = \mathrm{trunc}(10 \sin(\mathit{zwei}) + \cos(\mathit{zwei}))\}, \mathit{datatype} = \mathit{anything}, \mathit{storage} = \mathit{rectangular}, \mathit{order} = \mathit{Fortran_order})$

Die Standard-Ausgabe von tables oder arrays ist nicht sehr übersichtlich:

Jetzt machen wir noch eine ordentliche Ausgabeprozedur:

```
> myprint:=proc(werte::Array)
  local i,j,n;
  n:=sqrt(ArrayNumElems(werte));
  for i from 0 to n-1 do
     for j from 0 to n-1 do
         printf("%a ",werte[i,j])
     end do;
     printf("\n");
  end do;
  end proc:
> myprint(w);
f eins zwei
eins trunc(10*sin(eins)+cos(eins)) trunc(10*sin(eins)+cos(zwei))
zwei trunc(10*sin(zwei)+cos(eins)) trunc(10*sin(zwei)+cos(zwei))
```

Oder: Endliche Gruppe:

```
> g:=(x,y)->x+y mod 5;
```

$$g := (x, y) \rightarrow (x + y)\ \mathbf{mod}\ 5$$

```
> Wertetabelle(g,[0,1,2,3,4]):
```

```
> myprint(eval(%));
g 0 1 2 3 4
0 0 1 2 3 4
1 1 2 3 4 0
2 2 3 4 0 1
3 3 4 0 1 2
4 4 0 1 2 3
```

... könnte man noch schöner formatieren

```
> ? printf
```

Anmerkung: Die rtable - basierten Datenstrukturen (rtable, speziell Array, Matrix, Vector) werden normalerweise auch mittels print `schön' ausgegeben:

```
> A:=Array(1..3,1..3,(i,j)->log[2](i^2+j^2));
```

$$A := \begin{bmatrix} 1 & \frac{\ln(5)}{\ln(2)} & \frac{\ln(10)}{\ln(2)} \\ \frac{\ln(5)}{\ln(2)} & 3 & \frac{\ln(13)}{\ln(2)} \\ \frac{\ln(10)}{\ln(2)} & \frac{\ln(13)}{\ln(2)} & \frac{\ln(18)}{\ln(2)} \end{bmatrix}$$

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

$$\begin{bmatrix} 1 & \frac{\ln(5)}{\ln(2)} & \frac{\ln(10)}{\ln(2)} \\ \frac{\ln(5)}{\ln(2)} & 3 & \frac{\ln(13)}{\ln(2)} \\ \frac{\ln(10)}{\ln(2)} & \frac{\ln(13)}{\ln(2)} & \frac{\ln(18)}{\ln(2)} \end{bmatrix}$$

aber z.B.

```
> A:=Array(1..30,1..30,(i,j)->log[2](i^2+j^2));
```

$$A := \begin{bmatrix} \textit{1..30 x 1..30 Array} \\ \textit{Data Type: anything} \\ \textit{Storage: rectangular} \\ \textit{Order: Fortran_order} \end{bmatrix}$$

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

$$\begin{bmatrix} \textit{1..30 x 1..30 Array} \\ \textit{Data Type: anything} \\ \textit{Storage: rectangular} \\ \textit{Order: Fortran_order} \end{bmatrix}$$

```
> print(A[1..3,1..3]);
```

$$\begin{bmatrix} 1 & \frac{\ln(5)}{\ln(2)} & \frac{\ln(10)}{\ln(2)} \\ \frac{\ln(5)}{\ln(2)} & 3 & \frac{\ln(13)}{\ln(2)} \\ \frac{\ln(10)}{\ln(2)} & \frac{\ln(13)}{\ln(2)} & \frac{\ln(18)}{\ln(2)} \end{bmatrix}$$

Abschließendes Beispiel: ***Eine Prozedur, die eine Funktion erzeugt:***
(in nichttrivialen Fällen nicht einfach zu handhaben - Auswertungsregeln!)

```
> p:=proc(f,y::list)
  local i;
  return 'x->sum(f(x-y[i]),i=1..nops(y))';
  end proc;
```

$p := \textbf{proc}(f, y::list)\ \textbf{local}\ i;\ \textbf{return}\ 'x \rightarrow sum(f(x-y[i]), i=1..nops(y))'\ \textbf{end proc}$

```
> p('x->x^2',[a,b,c,d])(z);
```

$$(z-a)^2 + (z-b)^2 + (z-c)^2 + (z-d)^2$$

```
> q:=p(sin,[1,2,3]);
```

$$q := x \rightarrow \sum_{i=1}^{nops([1, 2, 3])} \sin\left(x - [1, 2, 3]_i\right)$$

```
> q(x);
```

$$\sin(x-1) + \sin(x-2) + \sin(x-3)$$

```
########################################################
```