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

ComputerMathematik - Einführung in Maple / Teil III

(Grundlagen der Programmierung in Maple)

Winfried Auzinger (SS 2010)

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

1. Auswertungsregeln; weitere nützliche Befehle

*** 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; b:=a;
```

$$a := 1$$

$$b := 1$$

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

$$a := 999$$

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

$$a = 999, b = 1$$

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[0]:=A0NEU; A;
```

$$A_0 := A0NEU$$

$$Array(0..2, \{(0) = A0NEU, (1) = y, (2) = z\}, datatype = anything, storage = rectangular, order = Fortran\_order)$$

```
> B;
```
$$Array(0..2, \{(0)=A0NEU, (1)=y, (2)=z\}, datatype=anything, storage=rectangular, order=Fortran\_order)$$
```
> 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]:=A0NEU;
```
$$A_0 := A0NEU$$
```
> 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.

*** 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;
```
$$2, 1$$

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

```
> x,y:='x','y':
> p := x->add((i+1)*(x-1)^i,i=0..5);
  collect(p(x),x); sort(%);
```
$$p := x \rightarrow add((i+1)(x-1)^i, i=0..5)$$
$$-3+6x^5-25x^4+44x^3-39x^2+18x$$
$$6x^5-25x^4+44x^3-39x^2+18x-3$$
```
> seq(coeff(p(x),x,k),k=0..5);
```
$$-3, 18, -39, 44, -25, 6$$

*** Der Differentiationsoperator D:**

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

```
> x,y:='x','y':
  D(sin);
```

$$\cos$$

```
> D(x->x^2); %(y);
```

$$x \to 2\,x$$

$$2\,y$$

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

$$-\sin(x)$$

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

$$\phi := (x, y) \to x^2\,y^3$$

```
> D[1](phi), D[2](phi); # partielle Ableitungen
```

$$(x, y) \to 2\,x\,y^3,\ (x, y) \to 3\,x^2\,y^2$$

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

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

*** 'Recursive assignment': für Namen nicht möglich!**

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

$$a := a$$

```
> a:=a+1;       # geht natürlich nicht - was soll das überhaupt heißen?
Error, recursive assignment
> a:=1;
  a:=a+1; # OK
```

$$a := 1$$

$$a := 2$$

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

*** 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': # folgendes 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); # das ist eine 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<200 do
>    printf("%d %d \n",j,j^2);
> end do;
```

```
1 1
4 16
7 49
10 100
13 169
```

'Reine' **while**-Schleife (abweisende Schleife):

```
> k:=100000:
  while not is(k,prime) do
     print(k);
     k:=k+1;
  end do:
```

100000

100001

100002

```
> k,ifactor(k);
```

100003, (100003)

Spezielle 'aufzählende' Varianten von **for**:

```
> for i in seq(i,i=3..1,-1) do i end do;
```

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 := 750072072199

x := 454744396973

x := 736602622344

next-Anweisung (zurück zu do und weiter):

```
> do
    x:=rand();
    if is(x,even) then
       next
    else
```

```
        print(x);
        break
    end if
  end do:
```

836404711117

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]);
```

16

3. Prozeduren in näherem 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(); 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]
```

Siehe auch Abschnitt 3.6.

Es gibt 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
```

Globale Variablen verwendet man 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.)

Globale Variablen können auch innerhalb einer Prozedur 'erzeugt' werden.

Einfaches Beispiel als Demo:

```
> restart;
> generate_Array := proc(n)
> global A;
> A:=Array(0..n);
> end proc;
```

$$generate\_Array := \mathbf{proc}(n)\ \mathbf{global}\ A;\ A := Array(0..n)\ \mathbf{end\ proc}$$

Ein kleiner Trick:
Wie können wir den **Namen** dieser globalen Variablen **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_Array := proc(Array_name::string,n)
  assign(convert(Array_name,name),Array(0..n));
  end proc;
```

$$generate\_Array := \mathbf{proc}(Array\_name::string, n)\ assign(convert(Array\_name, name), Array(0..n))\ \mathbf{end\ proc}$$

```
> generate_Array("myArray",5);
> print(myArray);
```

$$Array(0..5, \{\}, datatype = anything, storage = rectangular, order = Fortran\_order)$$

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 funktionert 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("hallo");
> end proc;
```

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

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

1

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

1

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

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

```
> p();
```

```
hallo
```

* Option **remember (!)** : verwendet man manchmal für **Rekursion**

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!

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

```
> fibonacci_v1 := proc(n) # REKURSIVE PROZEDUR!
> if (n<2) then
>    n
> else
>    fibonacci_v1(n-1)+fibonacci_v1(n-2)
> end if
> end proc:
```

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

```
> start:=time():
  fibonacci_v1(35);
  time()-start;
```

9227465

21.3900000000

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_v2:=proc(n) option remember;
> if (n<2) then
>    n
> else
>    fibonacci_v2(n-1)+fibonacci_v2(n-2)
> end if
> end proc;
```

fibonacci_v2 := **proc**(*n*)

option *remember*;

if $n < 2$ **then** *n* **else** *fibonacci_v2*($n - 1$) + *fibonacci_v2*($n - 2$) **end if**

end proc

```
> start:=time():
  fibonacci_v2(500);
  time()-start;
```

1394232245616978801397243828704072839500702565876973072641089629483255716228632906915576588\
76222521294125

0.0000000000

```
> interface(verboseproc=3):
> eval(fibonacci_v2): # hier würde man sieht den Inhalt
                      # der remember-table sehen.
```

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

```
> restart;
```

* Option **trace** :

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

```
> fibonacci_v1 := proc(n) option trace;
> if (n<2) then
>    n
> else
>    fibonacci_v1(n-1)+fibonacci_v1(n-2)
> end if
> end proc:
> fibonacci_v1(3);
{--> enter fibonacci_v1, args = 3
{--> enter fibonacci_v1, args = 2
{--> enter fibonacci_v1, args = 1
                                        1
<-- exit fibonacci_v1 (now in fibonacci_v1) = 1}
{--> enter fibonacci_v1, args = 0
                                        0
<-- exit fibonacci_v1 (now in fibonacci_v1) = 0}
                                        1
<-- exit fibonacci_v1 (now in fibonacci_v1) = 1}
```

```
{--> enter fibonacci_v1, args = 1
```

1

```
<-- exit fibonacci_v1 (now in fibonacci_v1) = 1}
```

2

```
<-- exit fibonacci_v1 (now at top level) = 2}
```

2

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

```
> 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;
```

$$p := \mathbf{proc}(\,)\ 2/3\ \mathbf{end\ proc}$$

```
> Digits;
```

30

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

```
> p:=proc()
> 1/1.00000000000000000000000000000000000001
> end proc;
```

$$p := \mathbf{proc}(\,)\ 1/1.0000000000\ \mathbf{end\ proc}$$

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

1.0000000000

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;
```

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

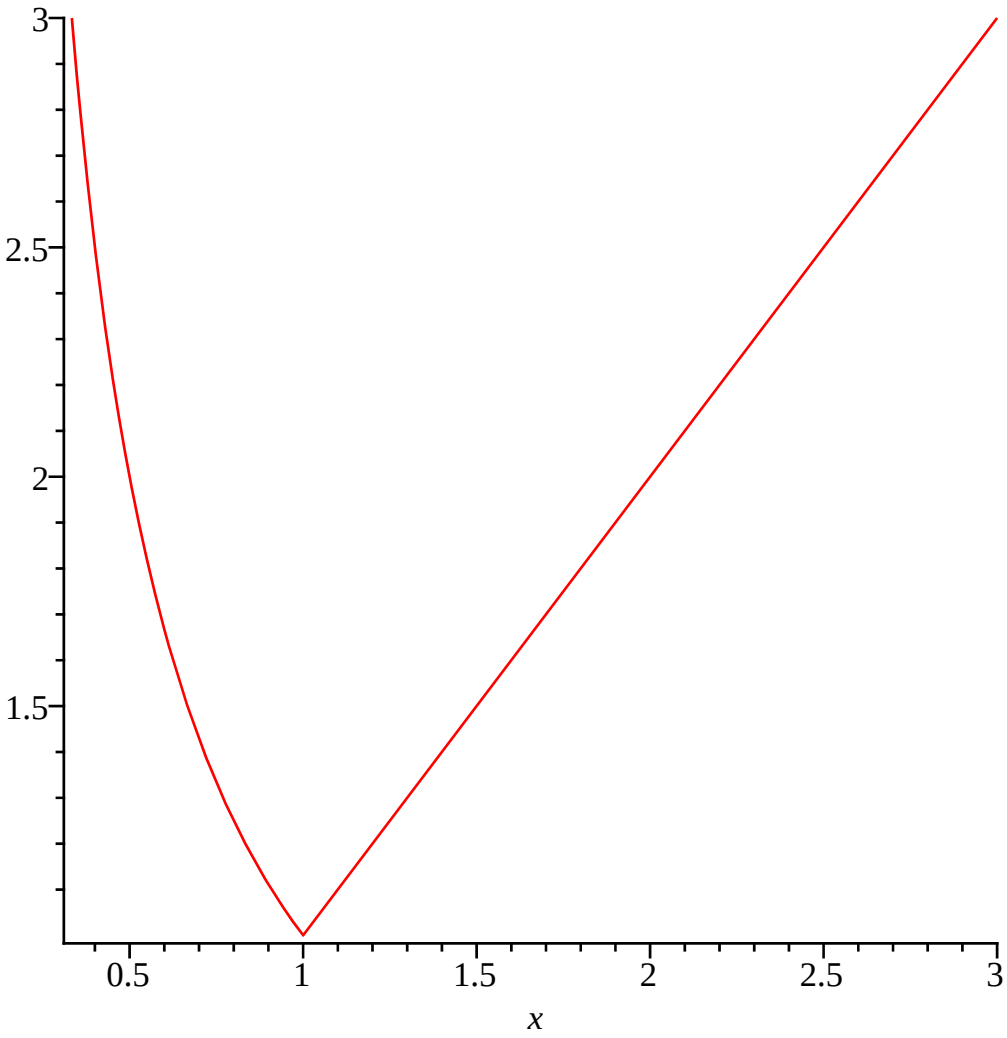
Hier wird versucht, mymax auszuwerten, bevor Werte eingesetzt werden:

```
> 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);
```



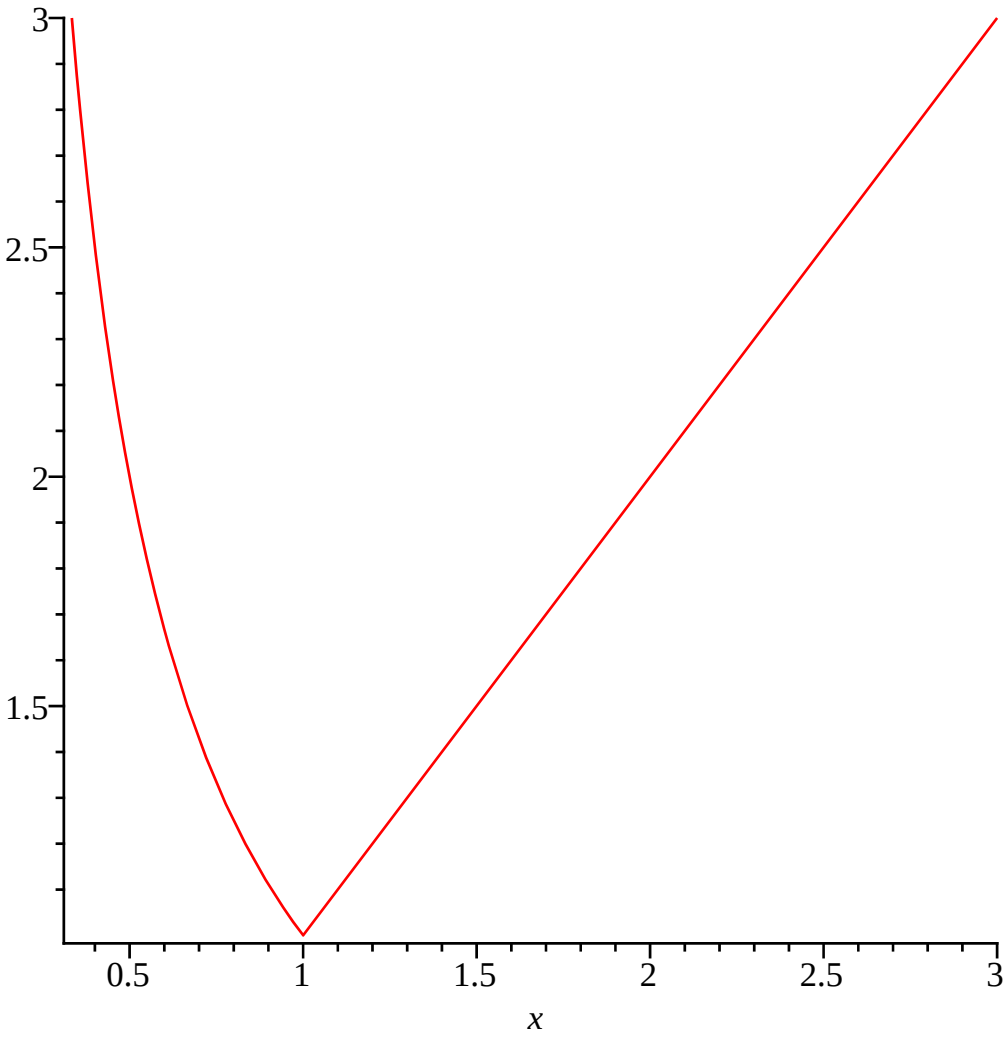
Man kann auch die Prozedur flexibler schreien:
Falls mymax nicht mit numerischen Werten aufgerufen wird,
wird die Auswertung verzögert (nicht immer leicht zu handhaben).
Typisches Problem, das bei Auswertung von 'stückweise' definierten
Funktionen auftritt.

```
> 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);
```

$$p$$

$$\mathbf{proc}(\,)\ 1\ \mathbf{end\ proc}$$

3.5 Alternative Syntaxvarianten für Prozeduren

* Funktionale Notation:

```
> f := (x::integer,y)->if  x<0 then x else y end if;
```

$$f := (x::integer, y) \rightarrow \mathbf{if}\ x < 0\ \mathbf{then}\ x\ \mathbf{else}\ y\ \mathbf{end\ if}$$

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

$$true$$

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

... ein **sehr praktischer** Befehl - wird dazu verwendet, um das (ggf. vereinfachte) Endergebnis einer zuvor erfolgten komplexeren Berechnung als Funktion 'umzudefinieren'.

Alternative: Die 'komplexere Berechnung' müsste jedes mal innerhalb der Funktion erfolgen!

Einfaches Beispiel für Verwendung:

```
> x,y:='x','y';
  a:=1;
  for i from 1 to 3 do
     a := sqrt(a+i*y+x^2):
  end do:
  ergebnis:=factor(diff(a,x));
```

$$x, y := x, y$$

$$a := 1$$

$$ergebnis := \frac{1}{4}\,\frac{x\left(1+2\sqrt{1+y+x^2}+4\sqrt{\sqrt{1+y+x^2}+2y+x^2}\,\sqrt{1+y+x^2}\right)}{\sqrt{\sqrt{\sqrt{1+y+x^2}+2y+x^2}+3y+x^2}\,\sqrt{\sqrt{1+y+x^2}+2y+x^2}\,\sqrt{1+y+x^2}}$$

```
> p:=unapply(ergebnis,x,y);
```

$$p := (x, y) \rightarrow \frac{1}{4}\,\frac{x\left(1+2\sqrt{1+y+x^2}+4\sqrt{\sqrt{1+y+x^2}+2y+x^2}\,\sqrt{1+y+x^2}\right)}{\sqrt{\sqrt{\sqrt{1+y+x^2}+2y+x^2}+3y+x^2}\,\sqrt{\sqrt{1+y+x^2}+2y+x^2}\,\sqrt{1+y+x^2}}$$

```
> p(3,4);
```

$$\frac{3}{56}\,\frac{\left(1+2\sqrt{14}+4\sqrt{\sqrt{14}+17}\,\sqrt{14}\right)\sqrt{14}}{\sqrt{\sqrt{\sqrt{14}+17}+21}\,\sqrt{\sqrt{14}+17}}$$

* Funktionale Komposition:

```
> f:=D(exp+ln);
```

$$f := \exp + \left(z \rightarrow \frac{1}{z} \right)$$

```
> f(1);
```

$$e + 1$$

```
> restart:
```

3.6 Variable Parameterlisten

Mittels des speziellen Namen **_passed** kann man flexible Listen unbenanner Parameter realisieren. **_npassed** liefert die Anzahl der aktuell übergebenen Parameter.

Beispiel:

```
> p:=proc()
>   local i;
>   printf("%d ",_npassed); # Anzahl übergebene Parameter
>   for i from 1 to _npassed do
        printf("%a ",_passed[i]) # %a-Format 'universell'
    end do;
    printf("\n");
> end proc:
> p(); p(a); p(a,b);
0
1 a
2 a b
```

Auch: -- Optionen, optionale Parameter
-- Schlüsselwortparameter
-- etc.

```
> ? using_parameters
```

3.7 Beispiele für Prozeduren

Im folgenden einige weitere Beispiele, die die Verwendung verschiedener Datenstrukturen zeigen bzw. die algorithmisch 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)) # limes!
> end if
> end proc:
```

```
> diffq(x->x^3,x,y);
```

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

```
> diffq(x->x^3,x,x);
```

$$3\,x^2$$

```
> 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,
        merge; # proc merge ist interne Prozedur!
  ###
  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:
  ###
  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([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 ('scope' der variablen etwas unübersichtlich).

Eher nur für einfache Hilfsfunktionen verwenden.

```
> restart;
```

BEISPIEL für eine (nicht rekursive) Prozedur,
die für eine gegebene Funktion f(x,y)
(soll z.B. eine endliche Gruppenoperation repräsentieren) 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:
> f:=(a,b)->a+b mod 3;
                              f := (a, b) → (a + b) mod 3
> Wertetabelle(f,[0,1,2]);
```

```
Array(0..3, 0..3, {(0, 0) = f, (0, 2) = 1, (0, 3) = 2, (1, 2) = 1, (1, 3) = 2, (2, 0) = 1, (2, 1) = 1, (2, 2) = 2, (3,
    0) = 2, (3, 1) = 2, (3, 3) = 1}, datatype = anything, storage = rectangular, order = Fortran_order)
```

Man sieht: 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 0 1 2
0 0 1 2
1 1 2 0
2 2 0 1
```

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

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

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

<-- Beachte hier auch die syntax mit `**generating function**'!

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

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

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,A[1..2,1..3]);
```

$$\left[\begin{array}{c} \textit{1..30 x 1..30 Array} \\ \textit{Data Type: anything} \\ \textit{Storage: rectangular} \\ \textit{Order: Fortran\_order} \end{array}\right], \left[\begin{array}{ccc} 1 & \frac{\ln(5)}{\ln(2)} & \frac{\ln(10)}{\ln(2)} \\ \frac{\ln(5)}{\ln(2)} & 3 & \frac{\ln(13)}{\ln(2)} \end{array}\right]$$

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);
```

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

Anhang: Spezielle Datenstrukturen

Stapelspeicher (**stack**) und Warteschlange (**queue)**
sind vor-implementiert über tables:

Anhang: Spezielle Datenstrukturen
*** STACK** = Linearer Speicher, zugriff nach dem LIFO - Prinzip
(`Last In - First Out'; **Stapelspeicher**)

```
> ? 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 dynamischen Prioritäten
('Haldenspeicher'; interne Implementierung komplizierter).

```
> ? heap
```

=== Ende Maple Teil III ===