ComputerMathematik - Einführung in Maple / Teil II

Winfried Auzinger (April 2009)

Hinweis:
Teil II ist angelehnt an MIPG, Kapitel 1-4.
Die während der VO entstehende fertige Version dieses Vortrags wird nach der VO auf der CompMath - Homepage zum Download bereitgestellt.

```
> restart;
```

0. Vorbemerkungen

Da Maple neben numerischen Berechnungen auch (in erster Linie wichtig!) SYMBOLISCHE Formelmanipulationen durchführt, sind einige Grundprinzipien zu beachten.

Grundsätzlich wird **exakt** gerechnet; numerische Approximation nach Bedarf

```
> 1/3; evalf(%);
```

$$\frac{1}{3}$$

$$0.33333333333333333$$

Eine reelle Zahl kann i.allg. nicht numerisch exakt angegeben werden,

```
> sqrt(2), sin(1);
```

$$\sqrt{2}, \sin(1)$$

aber man kann damit rechnen:

```
> sqrt(2)*sqrt(2), arcsin(sin(1));
```

$$2, 1$$

Rationale Zahlen werden automatisch gekürzt, aber nicht automatisch als Dezimalbruch dargestellt, auch wenn dieser endlich ist

```
> convert(sqrt(2),float);
```

$$1.4142135623730950 8 8$$

```
> ? convert
```

Viele Ergebnisse sind in einem '**generischen Sinn**' zu interpretieren, d.h. Maple geht davon aus, was 'normalerweise' - außer in Spezialfällen - zutrifft (alles andere wäre Unfug):

```
> y:=1/x;
```

$$y := \frac{1}{x}$$

```
> subs(x=0,y); # (x=0 einsetzen): das geht hier natürlich nicht
Error, numeric exception: division by zero
```

Als Programmierer hat man daher auf relevante Sonderfälle Acht zu geben!

Oder: Test auf Gleichheit

```
> evalb(a=1); # ('evaluate boolean')
```

false

...im Gegensatz zu

```
> a:=3-2; evalb(a=1);
```

$$a := 1$$

true

Vieles wird automatisch vereinfacht, aber nicht alles...

```
> sin(Pi);
```

$$0$$

```
> y:=1-cos(x)^2;
```

$$y := 1 - \cos(x)^2$$

```
> simplify(y);
```

$$\sin(x)^2$$

```
> p:=(x-1)*(x-2);
```

$$p := (x-1)\,(x-2)$$

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

$$(x-1)\,(x-2)$$

```
> expand(%);
```

$$x^2 - 3\,x + 2$$

```
> factor(%);
```

$$(x-1)\,(x-2)$$

Gleichungslösung (solve):
Beispiel für Verhalten abhängig von einer Systemvariablen:

```
> _EnvExplicit;
```

$$\mathit{\_EnvExplicit}$$

```
> solve(x^5-x^4+x^3+33*x^2-x);
```

$$0,\ \mathit{RootOf}\left(\_Z^4 - \_Z^3 + \_Z^2 + 33\,\_Z - 1,\ \mathit{index} = 1\right),\ \mathit{RootOf}\left(\_Z^4 - \_Z^3 + \_Z^2 + 33\,\_Z - 1,\ \mathit{index} = 2\right),$$
$$\mathit{RootOf}\left(\_Z^4 - \_Z^3 + \_Z^2 + 33\,\_Z - 1,\ \mathit{index} = 3\right),\ \mathit{RootOf}\left(\_Z^4 - \_Z^3 + \_Z^2 + 33\,\_Z - 1,\ \mathit{index} = 4\right)$$

```
> _EnvExplicit:=true; # vgl. ? solve
```

$$\mathit{\_EnvExplicit} := \mathit{true}$$

```
> solve(x^5-x^4+x^3+33*x^2-x):
```

Oder gleich numerisch lösen:

```
> fsolve(x^5-x^4+x^3+33*x^2-x);
```

$$-2.825100646625372709\,50,\ 0.000000000000000000,\ 0.030276068839160361\,42$$

Vorsicht bei Reihenentwicklungen, z.B. geometrische Reihe:

```
> sum(x^i,i=0..infinity); # Vorsicht: konvergiert nur für |x|<1!
```

$$-\frac{1}{x-1}$$

```
> taylor(1/(1-x),x=0); # Vorsicht: Konvergenzradius ist 1!
```

$$1 + x + x^2 + x^3 + x^4 + x^5 + \mathrm{O}\left(x^6\right)$$

Generell wird bei 'anonymen' Objekten angenommen, dass es sich um komplexe Zahlen handelt:

```
> u+u; Re(%);
```

$$2\,u$$
$$2\,\Re(u)$$

Man kann aber Eigenschaften definieren, z.B. Reellwertigkeit:

```
> u:='u';assume(u,real); about(u); Re(u),Im(u);
```

$$u := u$$

```
Originally u, renamed u~:
  is assumed to be: real
```

$$u,\ 0$$
With assumptions on u

Vieles ist in eigene **packages** ausgelagert, z.B. [Numerische] Lineare Algebra:

```
> with(LinearAlgebra); # aktiviert package
```

[*&x, Add, Adjoint, BackwardSubstitute, BandMatrix, Basis, BezoutMatrix, BidiagonalForm, BilinearForm, CharacteristicMatrix, CharacteristicPolynomial, Column, ColumnDimension, ColumnOperation, ColumnSpace, CompanionMatrix, ConditionNumber, ConstantMatrix, ConstantVector, Copy, CreatePermutation, CrossProduct, DeleteColumn, DeleteRow, Determinant, Diagonal, DiagonalMatrix, Dimension, Dimensions, DotProduct, EigenConditionNumbers, Eigenvalues, Eigenvectors, Equal, ForwardSubstitute, FrobeniusForm, GaussianElimination, GenerateEquations, GenerateMatrix, Generic, GetResultDataType, GetResultShape, GivensRotationMatrix, GramSchmidt, HankelMatrix, HermiteForm, HermitianTranspose, HessenbergForm, HilbertMatrix, HouseholderMatrix, IdentityMatrix, IntersectionBasis, IsDefinite, IsOrthogonal, IsSimilar, IsUnitary, JordanBlockMatrix, JordanForm, KroneckerProduct, LA_Main, LUDecomposition, LeastSquares, LinearSolve, Map, Map2, MatrixAdd, MatrixExponential, MatrixFunction, MatrixInverse, MatrixMatrixMultiply, MatrixNorm, MatrixPower, MatrixScalarMultiply, MatrixVectorMultiply, MinimalPolynomial, Minor, Modular, Multiply, NoUserValue, Norm, Normalize, NullSpace, OuterProductMatrix, Permanent, Pivot, PopovForm, QRDecomposition, RandomMatrix, RandomVector, Rank, RationalCanonicalForm, ReducedRowEchelonForm, Row, RowDimension, RowOperation, RowSpace, ScalarMatrix, ScalarMultiply, ScalarVector, SchurForm, SingularValues, SmithForm, StronglyConnectedBlocks, SubMatrix, SubVector, SumBasis, SylvesterMatrix, ToeplitzMatrix, Trace, Transpose, TridiagonalForm, UnitVector, VandermondeMatrix, VectorAdd, VectorAngle, VectorMatrixMultiply, VectorNorm, VectorScalarMultiply, ZeroMatrix, ZeroVector, Zip*]

```
> x:=Vector([1,I]); x.x;
```

$$x := \begin{bmatrix} 1 \\ I \end{bmatrix}$$

$$2$$

Es lohnt sich, mal zu schauen, was es alles für spezielle packages gibt.
(siehe UE)

```
> ? index[package]
```

Wichtig: Arbeiten mit online-Hilfe (mit vielen Beispielen!), z.B.
Befehlsliste:

```
> ? command
```

Dokumentation eines packages:

```
> ? codegen
```

1. Basics

```
> restart;
```

1.1 Die Software Maple

... kernel, library, packages (vgl. Teil 0)

1.2 Maple-Befehle - Überblick

```
> a:=2+3; # Wertzuweisung mit :=
```

$$a := 5$$

```
> 2+3: a; whattype(a);
```

$$5$$

$$integer$$

```
> my_string:="Das ist eine Zeichenkette (string)"; whattype
```

```
(my_string);
```

$$my\_string := \text{"Das ist eine Zeichenkette (string)"}$$

$$string$$

```
> a:=1, b:=2; # das geht nicht!
Error, `:=` unexpected
> c:=123/456; # rationale Zahl, wird automatisch gekürzt
```

$$c := \frac{41}{152}$$

```
> whattype(c); is(c,rational);
```

$$fraction$$

$$true$$

```
> evalf(2*c); # Ausgabe 10 Dezimalstellen ist eingestellt
              # (interne Rechengenauigkeit separat einstellbar)
```

$$0.53947368421052631579$$

```
> whattype(%); is(%%,real);
```

$$float$$

$$true$$

```
> diff(x^2+3*x-4/x+1,x); # Ableitung eines Ausdruckes
```

$$2\,x + 3 + \frac{4}{x^2}$$

Übersichtlicher z.B. so:

```
> my_expression:=x^2+3*x-4/x+1; whattype(%);
```

$$my\_expression := x^2 + 3\,x - \frac{4}{x} + 1$$

$$`+`$$

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

$$\frac{d}{dx}\left(x^2 + 3\,x - \frac{4}{x} + 1\right) = 2\,x + 3 + \frac{4}{x^2}$$

Dafür könnte man auch eine **Prozedur** schreiben
(der zuletzt berechnete Ausdruck wird zurückgegeben):

```
> mydiff:=proc(expr,var)
    Diff(expr,var)=diff(expr,var)
  end proc;
```

$$mydiff := \textbf{proc}(expr, var)\ \ Diff(expr, var) = diff(expr, var)\ \ \textbf{end proc}$$

```
> mydiff(my_expression,x);
```

$$\frac{d}{dx}\left(x^2 + 3\,x - \frac{4}{x} + 1\right) = 2\,x + 3 + \frac{4}{x^2}$$

```
> x+1 = Pi; # eine Gleichung (keine Wertzuweisung)
```

$$x + 1 = \pi$$

```
> solve(%,x);
```

$$-1 + \pi$$

```
> solve({x+y=1,2*x+2*y=2},{x,y}); # Gleichungssystem
```

$$\{x = -y + 1, y = y\}$$

oder z.B. so:

```
> eqs:=[x+y=1,2*x+2*y=2];
  vars:=[x,y];
  solve(eqs,vars);
```

$$eqs := [x + y = 1, 2x + 2y = 2]$$
$$vars := [x, y]$$
$$[[x = -y + 1, y = y]]$$

2. Elemente der Maple-Sprache

2.1 Zeichensatz

```
> evalb(A1=a1); # case sensitive!
```

false

```
> _myvar01; # zulässiger Name
```

_myvar01

2.2 Bezeichner (Namen; alle gültigen Symbole, aus Zeichen zusammengesetzt)

```
> ende_01;
```

ende_01

```
> alpha; # wird griechisch angezeigt
```

α

```
> pi; # Vorsicht: pi ist nicht Pi, wird aber gleich angezeigt
```

π

```
> simplify(Pi=pi); evalb(%);
```

$\pi = \pi$

false

Reservierte Namen sind i.w. Befehle der Maple-Sprache (Tabelle 2.2):

```
> end:=1; # unzulässiger Name ('end' ist reserved)
Error, reserved word `end` unexpected
```

Aber: Jede beliebige Zeichenfolge kann man mit `...` zu einem Namen machen:

```
> `end`:=infinity; `end`;
```

end := ∞

∞

```
> `manchmal kann auch so ein dummer name ganz praktisch sein`:=0;
```

manchmal kann auch so ein dummer name ganz praktisch sein := 0

```
> `Ergebnis des 1000. Übungsbespiels, Übungsblatt 100`:=2*Pi;
```

Ergebnis des 1000. Übungsbespiels, Übungsblatt 100 := 2π

```
> myPi:=3.14: # : statt ; unterdrückt Ausgabe
> myPi;
```

3.14000000000000000000

'Ditto'-operator (sparsam verwenden!): %, %%, %%%

```
> a:=1; b:=2; c:=3;
```

a := 1

b := 2

c := 3

```
> %,%%,%%%;
```

$$3, 2, 1$$

```
> a:=1; a; 'a'; %;
```

$$a := 1$$

$$1$$

$$a$$

$$1$$

```
> a:='a': b:='b': c:='c'; # löscht zugewiesene Werte!
>                         # nicht zu verwechseln mit `...`
```

$$c := c$$

Man kann mehrfache Zuweisungen auch in der Form exprseq:=exprseq schreiben
(in Schrift unüblich, aber eigentlich praktisch):

```
> a,b,c:='a','b','c';
```

$$a, b, c := a, b, c$$

```
> a||b, cat(a,b); # Verkettung von Namen zu neuem Namen
```

$$ab, ab$$

```
> c:="a"||"b"; d:=cat("a","b"); # Verkettung von Zeichenketten
```

$$c := \text{"ab"}$$

$$d := \text{"ab"}$$

```
> evalb(c=d); # Test auf Gleichheit (evaluate boolean)
```

$$true$$

Mehr über Namen, Voreinstellungen, etc.:

```
> restart;
```

Indizierte Namen (systematisch verwenden für indizierte Objekte!)

```
> a[0]; whattype(a[0]); # 0 Index wird tiefgestellt angezeigt
```

$$a_0$$

$$indexed$$

```
> a[0]:=0; b[1,1]:=1; c[heute]:=2;
```

$$a_0 := 0$$

$$b_{1,\,1} := 1$$

$$c_{heute} := 2$$

Vordefinierte Namen: siehe

```
> ? ininames
```

Zum Beispiel:

```
> I, Pi; # aber: e nicht vordefiniert!
```

$$I, \pi$$

```
> exp(I*Pi);
```

$$-1$$

```
> e^(I*Pi); simplify(%);
```

$$e^{I\pi}$$

$$e^{I\pi}$$

```
> exp(I*Pi);
```

$$-1$$

Umgebungsvariablen: siehe

```
> ? anames
```

```
> anames('environment');
```
Testzero, UseHardwareFloats, Rounding, %, _ans, %%%, Digits, index/newtable, mod, %%, Order, printlevel, Normalizer, NumericEventHandlers

Z.B,:

```
> Digits; # eingestellte Gleitpunkt-Rechengenauigkeit;
```
30

```
> Digits:=3;
```
Digits := 3

```
> evalf(Pi);
```
3.14000000000000000000

```
> Digits:=30;
```
Digits := 30

```
> evalf(Pi); # Ausgabe bei mir auf 20 Stellen eingestellt
```
3.14159265358979323846

Vordefinierte mathematische und logische Konstanten:

```
> constants;
```
false, γ, ∞, *true*, *Catalan*, *FAIL*, π

```
> evalf(gamma);
```
0.57721566490153286061

```
> gamma:=0.6;
```
```
Error, attempting to assign to `gamma` which is protected
```
```
> unprotect(gamma); gamma:=0.6; # mit Vorsicht verwenden
```
γ := 0.60000000000000000000

2.3 Spezielle Zeichen und ihre Bedeutung, insbesondere

: , ( ) ' ' [ ] { }

```
> u:=v: # : unterdrückt Ausgabe
> (x+y)*z; expand(%); # Klammerung
```
$(x+y)\,z$

$z\,x+z\,y$

```
> a:=1:   a,whattype(a); # a wird ausgewertet
  a:='a': a,whattype(a); # WICHTIG! - Bedeutung der Apostrophe!
```
1, *integer*

a, *symbol*

Mit einer derartigen Zuweisung (name='name')
wird also eine ggfs. zuvor vorgenommene Zuweisung rückgängig gemacht.

REGEL: Auswertung eines 'gekapselten' Ausdruckes (mit Apostrophen)
entfernt genau ein Paar von Apostrophen.

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

```
> ''x''; %; %;
```
'*x*'

x

1

```
> a,b,c; whattype(%); # eine FOLGE (Basiskonstrukt)
```

$$a, b, c$$

$$exprseq$$

```
> 1,seq(i^2,i=1..5); # generiert Folge mittels `impliziter
  Schleife'
```

$$1, 1, 4, 9, 16, 25$$

LISTEN sind statische geordnete lineare Folgen von Objekten in der Form [Folge], also 1D-Felder, Indizierung beginnt mit 1.

Beispiel:

```
> primes:=[2,3,5,7]; whattype(primes); # eine Liste
```

$$primes := [2, 3, 5, 7]$$

$$list$$

```
> primes[2..4]; # Zugriff auf Elemente (Teilliste)
```

$$[3, 5, 7]$$

Element dazugeben: nicht so

```
> primes[5]:=11;
Error, out of bound assignment to a list
```

sondern leider nur so (op ist genereller Befehl zur Selektion von [Teil]Ausdrücken):

```
> primes:=[op(primes),11];
```

$$primes := [2, 3, 5, 7, 11]$$

```
> nops(primes); # Anzahl Elemente
```

$$5$$

```
> primes:=[op(primes[1..2]),blabla,op(primes[3..4])];
```

$$primes := [2, 3, blabla, 5, 7]$$

```
> primes:=[seq(primes[i],i=1..nops(primes)-1)];
```

$$primes := [2, 3, blabla, 5]$$

```
> primes[1..nops(primes)-1];
```

$$[2, 3, blabla]$$

```
> ? list
```

Endliche **MENGEN** (im Sinne der mengentheoretischen Definition) schreibt man in der Form {Folge}, mit allen Eigenschaften von Mengen:

```
> M:={2,2,3,5,7}; whattype(M); # eine Menge
```

$$M := \{2, 3, 5, 7\}$$

$$set$$

```
> N:={seq(i,i=2..10,2)};
```

$$N := \{2, 4, 6, 8, 10\}$$

Vereinigung, Durchschnitt:

```
> M union N, M intersect N;
```

$$\{2, 3, 4, 5, 6, 7, 8, 10\}, \{2\}$$

Folgende Mengen sind identisch:

```
> evalb({a,b}={b,a,b,a,b});
```

$$true$$

```
> ? set
```

2.4 Typen und Operanden

Jedes Maple-Objekt gehört einem oder mehreren TYPEN an.

Beispiele:

```
> whattype('x');
```

symbol

```
> type('x',name), type('x',integer);
```

true, false

```
> whattype(1);
```

integer

```
> type(1,integer), type(1,posint), type(1,negint), type(1,
  rational);
```

true, true, false, true

```
> whattype(1/3);
```

fraction

```
> type(1/3,fraction), type(1/3,rational);
  # (rational bedeutet integer oder fraction)
```

true, true

```
> whattype(q[1]); type(q[1],name);
```

indexed

true

```
> whattype("Zeichenkette");
```

string

```
> whattype(3.14); type(3.14,numeric);
```

float

true

```
> whattype(1,2,3,4); whattype([%]); whattype({%%});
```

exprseq

list

set

Jedes Maple-Objekt besteht aus einem oder mehreren OPERANDEN.
Zugriff mittels op(...); benötigt man manchmal bei der Programmierung
zur Selektion von Teilausdücken.

Beispiele:

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

$a := a$

```
> nops(a);
```

1

```
> op(0,a), op(1,a);  # op(0,...) ergibt Typ
```

symbol, a

```
> op(a);
```

a

```
> b:=a+1;
```

$b := a + 1$

```
> nops(b);
```

2

```
> op(0,b); op(b);
```

`+`

$$a, 1$$

```
> c:=a^2-2*a+1-5+sin(x);
```

$$c := a^2 - 2\,a - 4 + \sin(1)$$

```
> op(c); # geht REKURSIV weiter:
```

$$a^2, -2\,a, -4, \sin(1)$$

```
> op(0,op(1,c)), op(op(1,c));
```

$$\text{`^`}, a, 2$$

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

$$1, 2, 3, 4$$

```
> restart;
```

BEISPIEL:

Eine Prozedur, die die Operanden eines Ausdruckes als Tabelle zurückgibt:

```
> opshow:=proc(expr) # expr=Eingangsparameter
  local i,operand; # lokale Variablen
  for i from 1 to nops(expr) do # for-Schleife
    operand[i]:=op(i,expr)
  end do;
  eval(operand); # generiert Rückgabewert
                 # (= table aus indizierten Werten -
                 #    tables (=assoziative arrays): später
  genauer)
  end proc;
```

opshow := **proc**(*expr*)
local *i*, *operand*;
for *i* **to** *nops*(*expr*) **do** *operand*[*i*] := *op*(*i*, *expr*) **end do**; *eval*(*operand*)
end proc

```
> optab:=opshow(x+y): whattype(optab); type(optab,table);
```

symbol

true

```
> optab, eval(optab);
  # (Tabelleninhalt wird erst mittels eval(...) angezeigt)
```

optab, *table*([1 = *x*, 2 = *y*])

```
> nops(optab), nops(eval(optab));
```

$$1, 2$$

```
> seq(optab[i],i=1..nops(eval(optab))); # implizite Schleife
  # (generiert exprseq bestehend aus optab[i],i=1,...)
```

$$x, y$$

Variante: Wir machen eine Liste der Operanden
in umgekehrter Reihenfolge.

Falls zu viele Operanden, wird "Ausdruck zu lang" gemeldet
und `nichts' (leer, NULL) zurückgegeben

```
> opinv:=proc(expr,max_length)
  local i,length; # nur eine local-Deklaration zulässig
  exprseq:=NULL; # leer (Initialisierung)
  length:=nops(expr);
  if (length>max_length) then
```

```
    printf("Ausdruck zu lang!");
    return NULL; # Rückgabe leer
  else
    for i from nops(expr) to 1 by -1 do # Schleife abwärts
      exprseq := exprseq,op(i,expr)
    end do;
    printf("Fertig!"); # (formatierte) Ausgabe
    return [exprseq]; # Rückgabewert explizit deklariert
  end if;
  end proc: # end proc; würde Prozedur anzeigen (siehe unten)
```

```
Warning, `exprseq` is implicitly declared local to procedure `opinv`
```

Man beachte obige Warnung: eigentlich fehlt local exprseq

```
> print(opinv); # Prozedur anzeigen
```

```
proc(expr, max_length)
    local i, length, exprseq;
    exprseq := NULL;
    length := nops(expr);
    if max_length < length then
        printf("Ausdruck zu lang!"); return NULL
    else
        for i from nops(expr) by -1 to 1 do exprseq := exprseq, op(i, expr) end do;
        printf("Fertig!");
        return [exprseq]
    end if
end proc
```

```
> expr:=x^2+2*x+3; op(expr);
```

$$expr := x^2 + 2\,x + 3$$

$$x^2,\ 2\,x,\ 3$$

```
> opinv(expr,3);
```

```
Fertig!
```

$$[3,\ 2\,x,\ x^2]$$

```
> opinv(NULL,3); # das geht nicht, NULL ist leer, d.h. 'nicht
  existent'!
```

```
Error, invalid input: opinv uses a 2nd argument, max_length, which is
missing
```

```
> q:=opinv(expr+y,3); evalb(q=NULL);
```

```
Ausdruck zu lang!
```

$$q := true$$

3. Maple-Ausdrücke und Anweisungen

... ein systematischer Überblick ...

3.1 Syntax und Semantik

```
> a/b*c; # Vorsicht!
```

$$\frac{a\,c}{b}$$

```
> (a/b)*c; # klarer
```

$$\frac{a\,c}{b}$$

3.2 Ausdrücke

Beispiele für Konstanten:

```
> constants;
```

$$false, \gamma, \infty, true, Catalan, FAIL, \pi$$

```
> infinity;
```

$$\infty$$

```
> sin(1); whattype(%);
```

$$\sin(1)$$

$$function$$

```
> 3.14E-3; whattype(%);
```

$$0.00314000000000000000$$

$$float$$

```
> 4/6; whattype(%); # automatisch gekürzt
```

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

$$fraction$$

```
> numer(2/3), denom(2/3); # Zähler, Nenner
```

$$2, 3$$

```
> evalf(2/3); # Gleitpunkt-Auswertung
```

$$0.66666666666666666667$$

```
> z:=3+4*I; whattype(z); # komplexe Zahl
```

$$z := 3 + 4\,I$$

$$complex(extended\_numeric)$$

```
> Re(z), Im(z), abs(z), argument(z);
```

$$3, 4, 5, \arctan\left(\frac{4}{3}\right)$$

```
> I^2, 1/I;
```

$$-1, -I$$

Beispiele für arithmetische Operatoren:

```
> infinity*infinity; whattype(%);
```

$$\infty$$

$$extended\_numeric$$

```
> 1-infinity, infinity/infinity;
```

$$-\infty, undefined$$

```
> x^y; # Potenz
```

$$x^y$$

```
> I^I; simplify(%);
```

$$I^I$$

$$e^{-\frac{1}{2}\pi}$$

Operatoren für Funktionskomposition:

```
> (sin@cos)(x); # Komposition von Funktionen
```

$$\sin(\cos(x))$$

```
> ((x->x^2)@@3)(x); # funktionale Potenz
```

$$x^8$$

oder:

```
> g:=x->x^2; (g@@3)(y); # (Klammerung notwenig)
```

$$g := x \rightarrow x^2$$

$$y^8$$

Range-Operator: ('a bis b')

```
> 1..5; whattype(%);
  # (wird z.B für Schleifenkonstrukte verwendet)
  # ist aber nicht dasselbe wie die Folge 1,2,3,4,5
```

$$1..5$$

$$..$$

```
> x:=1,2,3,4,5; whattype(%); x[1];
```

$$x := 1, 2, 3, 4, 5$$

$$exprseq$$

$$1$$

das ist dasselbe wie:

```
> y:=seq(i,i=1..5); whattype(%); y[1];
```

$$y := 1, 2, 3, 4, 5$$

$$exprseq$$

$$1$$

Diverses:

```
> 5!; # Faktorielle
```

$$120$$

```
> 11 mod 3; # modulo
```

$$2$$

Vergleichsoperatoren:

```
> 1<2; whattype(%);
```

$$1 < 2$$

$$`<`$$

```
> evalb(1<2); # 'evaluate boolean'
```

$$true$$

```
> evalb(1=2), evalb(1>2); # etc.
```

$$false, false$$

```
> x,y:='x','y'; evalb(x=y); evalb(x<>y); # 'generisch' korrekt!
```

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

$$false$$

$$true$$

Weitere Vergleichsoperatoren: <= >= <>

Gleichungen, linke und rechte Seite einer Gleichung:

```
> x=y; evalb(%); # hier braucht man evalb
```

$$x = y$$

$$false$$

```
> eq := x=y; lhs(eq),rhs(eq);
```

$$eq := x = y$$

$$x, y$$

Logische Operatoren: operieren auf Boole'schen Ausdrücken:

```
> 1>2 or 1<2; # wird automatisch ausgewertet (ohne evalb)
```

$$true$$

```
> a=a and a=c;
```

$$false$$

```
> 1<2 xor 1<3;
```

$$false$$

```
> x:=not(a=b);
```

$$x := true$$

```
> evalb(1+I*2<3); # !!
```

$$FAIL$$

```
> FAIL or true;
```

$$true$$

```
> FAIL and true;
```

$$FAIL$$

Die Funktion is fragt: Ist das wahr? - Beispiele:

```
> is(1,positive);
```

$$true$$

```
> is(I,integer);
```

$$false$$

DATENSTRUKTUREN:

* Folge (exprseq), Liste (list), Menge (set): siehe oben

* Weiters: Tabelle (table), Feld (Array)

Eine Tabelle ist ein **ASSOZIATIVES** Feld:

```
> days:=table([Januar=31]);
```

$$days := table([Januar = 31])$$

```
> days[Januar];
```

$$31$$

```
> days[Februar]:=28;
```

$$days_{Februar} := 28$$

Die Inhalte komplexerer Objekte wie tables werden nur `bei Bedarf',
z.B. mittels eval(...) angezeigt:

```
> days;
  eval(days);
```

$$days$$

$$table([Februar = 28, Januar = 31])$$

```
> length(days); # !!!
```

$$4$$

Der Befehl length ist mit Vorsicht zu genießen;
er zählt (kontext-abhängig) Ziffern, Zeichen, Speicher o.ä.,
aber z.B. nicht die Anzahl von Tabellenelementen.

```
> ? length
```

Die tatsächliche Anzahl der Einträge erhälz man mittels

```
> nops(eval(days)); # eval!
```

$$2$$

* Geht auch mehrdimensional:

```
> sudokufarbe:=table();
```

$$sudokufarbe := table([\,])$$

```
> sudokufarbe[eins,zwei]:=grün;
```

$$sudokufarbe_{eins,\ zwei} := grün$$

```
> eval(sudokufarbe);
```

$$table([\,(eins, zwei) = grün\,])$$

```
> ? table
```

* **Array**: Ähnlich wie table, aber -- ganzzahlige Indizierung
-- statische Speicherallokation

```
> a:=Array(-2..-1,0..1); # hier wird der Indexbereich festgelegt
                         # Standard-Initialisierung: alle Werte 0
```

$$a := Array(-2..-1, 0..1, \{\}, datatype = anything, storage = rectangular, order = Fortran\_order)$$

```
> eval(a);
```

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

```
> print(a);
```

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

```
> a[-1,0];
```

$$0$$

```
> a[-1,0]:=9999999; eval(a);
```

$$a_{-1,\ 0} := 9999999$$

$$Array(-2..-1, 0..1, \{(-1, 0) = 9999999\}, datatype = anything, storage = rectangular, order = Fortran\_order)$$

```
> A:=Array(1..2);A[1],A[2],A[3];
```

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

```
Error, Array index out of range
```

```
> A:=Array(1..2,1..3);
```

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

```
> A:=Array([[1,2,3],[4,5,6]]); # hier werden Werte mittels Listen
  zugewiesen
```

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

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

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

```
> ArrayDims(A);
```

$$1..2, 1..3$$

```
> ? Array
```

FUNKTIONEN:

```
> f(s,t); # eine `unspezifizierte' Funktionsauswertung
```

$$f(s, t)$$

Auswertungsregel:
Zuerst f symbolisch auswerten, dann Argumente s,t auswerten und einsetzen
('call by value')

```
> f(1,2);
```

$$f(1, 2)$$

```
> f:=(x,y)->x+y; # eine konkret spezifizierte Funktion
  f(1,2);
```

$$f := (x, y) \rightarrow x + y$$

$$3$$

```
> whattype(eval(f));
```

procedure

Kann man auch als Prozedur schreiben:

```
> f:=proc(x,y)
> x+y; # besser explizit: return x+y;
> end proc;
```

$$f := \mathbf{proc}(x, y)\ \ x + y\ \mathbf{end\ proc}$$

```
> whattype(f);
```

symbol

```
> whattype(eval(f));
```

procedure

```
> f(1,katerkarlo);
```

$$1 + katerkarlo$$

3.3 Verwendung von Ausdrücken

* Ausdrücke sind als Bäume dargestellt, kann man mit op 'scannen'
  (siehe oben)

* Auswertung von Ausdrücken: Normalerweise 'unmittelbar',
  triviale Vereinfachungen werden sofort ausgeführt;

'Verzögerung' der Auswertung mittels '...'

```
> a:=2; b:=3;
```

$$a := 2$$

$$b := 3$$

```
> a-b;
```

$$-1$$

```
> 'a-b';
```

$$a - b$$

```
> %; # jetzt wird ausgewertet
```

$$-1$$

```
> # Auch eine Funktion kann unausgewertet sein:
> 'f'(1,2); # f on oben
```

$$f(1, 2)$$

```
> %; # jetzt wird ausgewertet
```

$$3$$

* Einsetzen von Werten in Ausdrücke mittels subs:

```
> subs(x=2,x^2+y^2);
```

$$4 + y^2$$

```
> subs(g=f,g(1,2));
```

$$1$$

```
> %;
```

$$1$$

3.4 Anweisungen (--> Programmierung!)

Wir schreiben eine [ziemlich sinnlose] Prozedur,
wo gleich mal rein syntaktisch ziemlich viel vorkommt
(genaueres später)

```
> restart;
> p:=proc(a::posint,b::posint)
  description "eine sinnlose prozedur";
  local i,s,p; # lokale Variablen
  global g; # globale Variablen
  s,p:=a+b,a*b; # Zuweisung

  if (p<s) then # Selektion (if)
    printf("a*b ist kleiner als a+b")
  elif (p=s) then # Vorsicht: elif nicht elseif
    printf("a*b ist gleich a+b")
  else
    printf("a*b ist größer als a+b")
  end if;

  printf("\n");
  for i from a*b to 1 by -1 do # Schleife
    printf("%d ",i) # formatierte ausgabe
  end do;

  i:=0;
  printf("\n");
  while(i<a+b and is(i,prime)) do # while-Schleife
    printf("%d ist < a+b \n",i);
    i:=i+1:
  end do;

  for i from 1 to a*b do
     if is(i,prime) then # überspringe Primzahlen
        next
     end if;
     if is(i^2+1,prime) then
        break # verlasse Schleife
     end if
  end do;
```

```
   if (a=10) then # hier provozieren wir Division durch 0
      return 1/(a-10);
   elif (a>10) then
      error "a zu gross!!"; # für Fehler-Exit
   end if

   end proc:
```

BEACHTE: Bei Syntaxfehler bei der Definition der Prozedur
erscheint Fehlermeldung Meldung, und Cursor springt
zu der Zeile, wo das Problem festgestellt wurde.

Bemerkungen:
* Es gibt formal nur EINEN Funktionswert; dieser kann aber
  im Prinzip ein beliebiges Maple-Objekt sein, auch eine exprseq
  (für Rückgabe mehrerer Werte)

  (Man kann z.B. auch eine Prozedur schreiben, die eine
  Prozedur als Wert zurückliefert)

* LOKALE Variablen sind Prozedur-intern
* GLOBALE Variablen sind innen und außen sichtbar,
  können innerhalb Prozedur auch neu generiert
  und verändert werden.

Achtung: Normalerweise sollte Ein/Ausgabe jedoch über Eingangsparameter
und Funktionswert erfolgen, nicht 'versteckt' über globale Variablen.

Ausnahme: Sehr große Objekte, die nur einmal im Speicher vorhanden sein sollen

```
> Describe(p);

# eine sinnlose prozedur
p( a::posint, b::posint )
```

```
> p(2,7);
a*b ist größer als a+b
14 13 12 11 10 9 8 7 6 5 4 3 2 1
```

```
> p(10,0);
Error, invalid input: p expects its 2nd argument, b, to be of type
posint, but received 0
> p(0,1);
Error, invalid input: p expects its 1st argument, a, to be of type
posint, but received 0
> p(2,9);
a*b ist größer als a+b
18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
```

```
> p(11,2); # provoziert meinen error-exit
a*b ist größer als a+b
22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
```

```
Error, (in p) a zu gross!!
```

Beachte: Bei LAUFZEIT-Fehlermeldungen wird man normalerweise NICHT darauf hingewiesen, an welcher Stelle der Fehler aufgetreten ist!

Daher wichtig für längere Prozeduren: Debugger verwenden (später)

```
> p(10,1); # Fehler in Prozedur - wurde oben
           # absichtlich eingebaut!!
a*b ist kleiner als a+b
10 9 8 7 6 5 4 3 2 1

Error, (in p) numeric exception: division by zero
```

4. Genaueres über Datenstrukturen

... Wir geben jeweils einige genauere Beispiele zur Verwendung von sets, lists, tables etc.
... wird aus Zeitgründen in Vorlesung ggf. nur kurz durchgescannt.

```
> restart;
```

4.1 Mengen (set)

```
> s1:={seq(-i,i=1..5)}; s2:={-2,-2,-1}; whattype(s1);
```

$$s1 := \{-5, -4, -3, -2, -1\}$$

$$s2 := \{-2, -1\}$$

$$set$$

```
> s1 union s2, s1 intersect s2, s1 minus s2;
```

$$\{-5, -4, -3, -2, -1\}, \{-2, -1\}, \{-5, -4, -3\}$$

```
> member(-1,s1);
```

$$true$$

Der Befehl **map** wendet eine Funktion elementweise an (verwandt zu **map** ist **zip**)

```
> f:=x->x^2; map(f,s1);
```

$$f := x \rightarrow x^2$$

$$\{1, 4, 9, 16, 25\}$$

```
> s:={sin(y),cos(y)};
```

$$s := \{\cos(y), \sin(y)\}$$

Eine allgemeinere Variante von **map** übergibt ein Argument an die angewendete Funktion:

```
> map(diff,s); # das ist sinnlos
Error, invalid input: diff expects 2 or more arguments, but received
1
> map(diff,s,y); # differenziere nach y
```

$$\{-\sin(y), \cos(y)\}$$

Ansonsten: Variante des **for**-Befehls, um eine Operation auf alle Elemente einer Menge anzuwenden:

```
> for e in s1 do e^2 end do;
```

$$25$$

$$16$$

$$9$$

4

1

4.2 Listen (list)

```
> L:=[$1..5]; # $1..5 ist Abkürzung für seq(i,i=1..5)
```

$$L := [1, 2, 3, 4, 5]$$

```
> nops(L); op(L);
```

5

1, 2, 3, 4, 5

```
> L[]; # liefert dasselbe wie op(L)
```

1, 2, 3, 4, 5

map, for .. in funktionieren analog wie bei sets:

```
> map(exp,L);
```

$$[e, e^2, e^3, e^4, e^5]$$

es gibt auch **listlist** (= Liste von gleich langen Listen):

```
> LL:=[[1,1],[2,4],[3,9]];
```

$$LL := [[1, 1], [2, 4], [3, 9]]$$

```
> whattype(LL); type(LL,listlist);
```

list

true

Manipulationen mit Listenelementen:

```
> L:=map(x->x^2,L);
```

$$L := [1, 4, 9, 16, 25]$$

```
> member(9,L);
```

true

```
> member(9,L,'position'); position; # Achtung eigenwillige Syntax
```

true

3

```
> L[position]; # Zugriff auf Element
```

9

Manipulation von Listen:

```
> L1:=[1,2]; L2:=[3,4];
```

$$L1 := [1, 2]$$

$$L2 := [3, 4]$$

```
> L:=[op(L1),op(L2)]; # Verknüpfung (Aneinanderhängen)
```

$$L := [1, 2, 3, 4]$$

```
> L:=[V,V,op(L),H,H,H]; # Elemente vorne,hinten anhängen
```

$$L := [V, V, 1, 2, 3, 4, H, H, H]$$

```
> convert(L,set);
```

$$\{1, 2, 3, 4, H, V\}$$

```
> L:=convert(%,list);
```

$$L := [1, 2, 3, 4, H, V]$$

Einfügen zwischendrin ist etwas mühsam:

```
> L:=[op(1..3,L),eingefuegt,op(4..nops(L),L)];
```

$$L := [1, 2, 3, eingefuegt, 4, H, V]$$

Beachte: Listen dienen der `geordneten' Aufbewahrung

durchnumerierter Objekte; sie sind aber zum `Rechnen' im Sinne
von Vektorrechnung nur bedingt geeignet. Dafür gibt
es eigene Datenstrukturen (--> später).

Aber immerhin: Elementare Vektorrechung,
d.h. Linearkombination von Listen funktioniert:

```
> [a,b,c]+2*[x,y,z];
```

$$[2\,x + a,\ 2\,y + b,\ 2\,z + c]$$

Siehe auch

```
> ? ListTools
```

4.3 Tabellen (tables)

Eine Tabelle erstellt man entweder explizit mittels einer
Elementliste (Default für Indizes ist 1,2,...) oder mittels expliziter,
Zuordnung der Elemente zu beliebigen `Indizes'.

```
> t1:=table([cos,sin,tan]);
```

$$t1 := table([1 = \cos, 2 = \sin, 3 = \tan])$$

```
> t1, eval(t1); t1[1]; indices(t1); entries(t1);
```

$$t1,\ table([1 = \cos, 2 = \sin, 3 = \tan])$$

$$\cos$$

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

$$[\cos], [\sin], [\tan]$$

```
> t2:=table([Montag=erster,Dienstag=zweiter]);
```

$$t2 := table([Montag = erster, Dienstag = zweiter])$$

```
> t2, eval(t2); t2[Montag]; indices(t2); entries(t2);
```

$$t2,\ table([Montag = erster, Dienstag = zweiter])$$

$$erster$$

$$[Montag], [Dienstag]$$

$$[erster], [zweiter]$$

tables sind `dynamisch' - man kann beliebig Elemente hinzufuegen:

```
> t1[infinity]:=unendlich; eval(t1);
```

$$t1_{\infty} := unendlich$$

$$table([1 = \cos, 2 = \sin, 3 = \tan, \infty = unendlich])$$

4.4 Arrays

Interne Speicherung: als rtable (eine Variante von table)

Vorteil von Arrays: Man kann beliebige ganzzahlige Indexbereiche
verwenden, insbesondere auch von 0 weg indizieren
(ist in manchen Anwendungen natürlicher als von 1 weg)

```
> a1:=Array(0..2,[1,2,3]);
```

$$a1 := Array(0..2, \{(0) = 1, (1) = 2, (2) = 3\}, datatype = anything, storage = rectangular, order = Fortran\_order)$$

```
> indices(a1); entries(a1);
```

$$[0], [1], [2]$$

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

```
> a2:=Array(0..2,0..2);
```

$$a2 := Array(0..2, 0..2, \{\}, datatype = anything, storage = rectangular, order = Fortran\_order)$$

```
> a2[0,0]:=x; indices(a2), entries(a2);
```

$$a2_{0,\,0} := x$$

$$[0,0],[0,1],[0,2],[1,0],[1,1],[1,2],[2,0],[2,1],[2,2],[0],[0],[0],[0],[0],[0],[0],[0],[x]$$

Hier sind die anderen Elemente (noch) undefiniert.

```
> A1:=Array(0..3);
```

$$A1 := Array(0..3, \{\}, datatype = anything, storage = rectangular, order = Fortran\_order)$$

```
> A1[1]; # hier automatisch Initialisierung mit 0
```

$$0$$

```
> eval(A1); # das ist ewas seltsam
```

$$Array(0..3, \{\}, datatype = anything, storage = rectangular, order = Fortran\_order)$$

```
> indices(A1);
```

$$[0],[1],[2],[3]$$

```
> A2:=Array(0..2,0..2,[[1,2,3],[4,5,x]]); eval(A2);
```

$$A2 := Array(0..2, 0..2, \{(0,0)=1, (0,1)=2, (0,2)=3, (1,0)=4, (1,1)=5, (1,2)=x\}, datatype = anything, storage = rectangular, order = Fortran\_order)$$

$$Array(0..2, 0..2, \{(0,0)=1, (0,1)=2, (0,2)=3, (1,0)=4, (1,1)=5, (1,2)=x\}, datatype = anything, storage = rectangular, order = Fortran\_order)$$

```
> A2:=Array([[1,2,3],[4,5,x]]); eval(A2);
  # so sieht es vernünftiger aus
```

$$A2 := \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & x \end{bmatrix}$$

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

BEMERKUNG: Bei Datenstrukturen herrscht in Maple ein gewisser WILDWUCHS (aus historischen Gründen?), mit teilweise ganz spezifischen Zugriffsfunktionen (siehe untiges Beispiel)

EMPFEHLUNG:

- Verwende **set** für Mengen
- Verwende **list** für (endliche) Folgen von Objekten
- Verwende **table** bei Bedarf für assoziative Arrays
- Verwende **Array** für `Vektoren' bzw `Matrizen', insbesondere wenn man Indizierung mit 0 beginnen will

UND:

- Verwende **Vector** und **Matrix** für Objekte im R^n (Lineare Algebra!!) - diese basieren intern auf einer alternativen, speicher-flexibleren Implementierung von tables (sogenannte 'rectangular tables', **rtables**) --

Vector und **Matrix** sind für das Rechnen mit Vektoren und Matrizen die relevanten Datenstrukturen!

... diese werden später bei der Diskussion des **LinearAlgebra** - Packages genauer besprochen.

5. Beispiel: Inneres Produkt

Wir schreiben eine Prozedur, die zwei eindimensionale Arrays im Sinne des inneren Produktes im R^n multipliziert. Die Summation des inneren Produktes führen wir mit dem (symbolischen) Befehl **sum** durch.

ACHTUNG: **sum** (im Gegensatz zur `arithmetischen' Summation **add**) verhält sich etwas eigenartig; man muss die Summanden meist mittels '...' als unausgewertet `maskieren'.

```
> IP:=proc(A::Array,B::Array)
    options `Copyright W.Auzinger, 2009`;
>   description "Inneres Produkt zweier 1D-Arrays";
>   local dA, dB, i, nA, nB;
>   dA:=ArrayNumDims(A);
>   dB:=ArrayNumDims(B);
>   nA:=ArrayNumElems(A);
>   nB:=ArrayNumElems(B);
>   if dA<>1 then
      error "A must be 1-dimensional"
    end if;
>   if dB<>1 then
      error "B must be 1-dimensional"
    end if;
>   if nA<>nB then error
      "A and B must have the same dimension"
    end if;
>   return sum('A[i]*B[i]',i=ArrayDims(A));
> end proc;
```

IP := **proc**(*A*::*Array*, *B*::*Array*) **description** "Inneres Produkt zweier 1D-Arrays"; ... **end proc**

```
> Describe(IP);

# Inneres Produkt zweier 1D-Arrays
IP( A::Array, B::Array )
```

```
> eval(IP); # Code wird nicht angezeigt (copyright-Klausel!)
```

proc(*A*::*Array*, *B*::*Array*) **description** "Inneres Produkt zweier 1D-Arrays"; ... **end proc**

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

$$A := \left[\begin{array}{ccc} 1 & 2 & 3 \end{array}\right]$$

```
> i:='i': A[i];
Error, bad index into Array
```

ACHTUNG -- Bemerkung zum Verhalten von **sum** (oder vergleichbarer Befehle wie **product**):
sum versucht A[i] auszuwerten, bevor i intern einen Wert zugewiesen bekommt (weil i als globale Variable betrachtet wird)

```
> sum(A[i],i=1..3);
Error, bad index into Array
```

Hier muss man die Auswertung von A[i] `verzögern', damit es nicht sofort beim Aufruf ausgewertet wird. Das Verhalten ist aber manchmal schwer vorherzusehen.

```
> sum('A[i]',i=1..3);
```

```
> ArrayNumDims(A),ArrayNumElems(A),ArrayDims(A);
```

$$1,\ 3,\ 1..3$$

```
> IP(A,A);
```

$$14$$

```
> IP(A,1);
Error, invalid input: IP expects its 2nd argument, B, to be of type
Array, but received 1
> B:=Array([1,2,3,4]);
```

$$B := \left[\begin{array}{cccc} 1 & 2 & 3 & 4 \end{array}\right]$$

```
> IP(A,B);
Error, (in IP) A and B must have the same dimension
> C:=Array([x,y,z]);
```

$$C := \left[\begin{array}{ccc} x & y & z \end{array}\right]$$

```
> IP(A,C);
```

$$x + 2\,y + 3\,z$$

```
> ##################################################################
  ##################################
```