

► **ComputerMathematik - Einführung in Maple / Teil II**
(kompetente Verwendung; Ausdrücke und Umformungen;
das wichtigste zu Datenstrukturen, Funktionen und Prozeduren)

INHALT TEIL II :

► **1. Prinzipielles über Auswertungen und Umformungen**

▼ **2. Elemente der Maple-Sprache**

- *2.1 Zeichensatz*
- *2.2 Bezeichner (Namen; alle gültigen Symbole, aus Zeichen zusammengesetzt)*
- *2.3 Spezielle Zeichen und ihre syntaktische Bedeutung, z.B.
: , () ' ' [] { }*
- *2.4 Typen und Operanden*

▼ **3. Maple-Ausdrücke und Anweisungen**

- *3.1 Ausdrücke*
- *3.2 Die wichtigsten Datenstrukturen*
- *3.3 Funktionen, Prozeduren*
- *3.4 Beispiel für eine allgemeinere Prozedur*

▼ **4. Praktischer Umgang mit Datenstrukturen**

- *4.1 Mengen (set)*

- *4.2 Listen (list)*

- *4.3 Tabellen (table)*

- *4.4 Felder (Array)*

ComputerMathematik - Einführung in Maple / Teil II

(kompetente Verwendung; Ausdrücke und Umformungen; das wichtigste zu Datenstrukturen, Funktionen und Prozeduren)

Winfried Auzinger (SS 2011)

Institut für Analysis und Scientific Computing

```
> restart; # restart engine; clear workspace
```

Die Worksheets der Vorlesung sind auf verschiedene Inhalte fokussiert, wobei sich öfters Wiederholungen zu früheren Vorlesungen ergeben (insbesondere zu dem Überblick in Teil I), wenn es vom Aufbau her sinnvoll ist (ein bisschen Redundanz schadet nicht).

Manches wird auch zunächst nur angedeutet und folgt später genauer.

Durcharbeiten der (ggf. individuell modifizierten) Worksheets ist (zumindest für den Neueinsteiger) empfehlenswert. Es handelt sich jedoch um keine komplette 'Referenz'. Zum Nachschlagen verwendet man die Online-Hilfe, und die zur Verfügung gestellte alphabetische Befehlsliste soll die Orientierung erleichtern.

Einige Praxis-Tipps:

- Verwendung der Online-Hilfe ist Teil es routinemäßigen Umganges mit Maple. Vieles erkennt man schnell anhand der dort angegebenen Beispiele.
- Mischen symbolischer und numerischer Operanden in Ausdrücken ist prinzipiell kein Problem. Dasselbe gilt für Funktionen und Prozeduren: Für Parameter können (soweit sinnvoll) sowohl Variablenamen als auch numerische Werte übergeben werden.
- Bei manchen Befehlen ist das genaue Verhalten nicht immer a priori klar, etwa was die genaue Funktionalität oder die Kompatibilität mit diversen Datenstrukturen betrifft. Ein gewisses Aufmaß an 'trial and error' ist daher nicht zu vermeiden. Eine gute Strategie besteht darin, zunächst einfachere Spezialfälle oder kleiner dimensionierte Probleme auszutesten, bevor man sich einer allgemeineren Problemstellung widmet.
- Man beobachtet in der Praxis immer wieder, dass von Maple angezeigte Zwischenergebnisse vom Benutzer manuell zur Weiterverarbeitung abgetippt werden. Das ist ungeschickt, unflexibel und fehleranfällig. Man bemühe sich daher um einen 'automatisierten' Datenfluss, mittels Zuweisung an Variablen und ggfs. Selektion von Teilausdrücken (vgl. z.B. ? op)
- Beim Durchspielen von Rechnungen abhängig von einem Parameter, z.B. n, probiert man oft Spezialfälle separat aus (n=1, n=2, n=3, ...). Spätestens ab (z.B.) n=4 zahlt es sich meistens aus, den Code für allgemeines n zu formulieren und dann (ggf. in einer Schleife über n) abzarbeiten bzw. Prozeduren zu verwenden.

- Bei der Eingabe von Ausdrücken in konventioneller Textform ('Maple notation', wie hier durchwegs) lohnt es sich manchmal, zwecks Kontrolle den Ausdruck zunächst mit '...' zu 'kapseln' (siehe 2.3) und erst nachher auszuwerten (zur Kontrolle), z.B.

```
> summe := 'add((1+i+i^2)/(1+i^2+i^4), i=1..5)';  

$$summe := add\left(\frac{1+i+i^2}{1+i^2+i^4}, i=1..5\right)$$
  
> summe;  

$$\frac{437}{273}$$

```

1. Prinzipielles über Auswertungen und Umformungen

```
> x := 0; # Wertzuweisung mit :=  
x := 0  
> a:=1, b:=2; # das geht nicht!  
Error, := unexpected  
> a,b := 1,2; # das geht  
a, b := 1, 2
```

Da Maple neben numerischen Berechnungen in erster Linie **exakte, symbolische** Formelmanipulationen durchführt, sind einige Grundprinzipien zu beachten.

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

```
> rationale_zahl := 2/6;  
rationale_zahl, evalf(rationale_zahl);  
 $\frac{1}{3}, 0.33333333333333333333333333333333$ 
```

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

```
> sqrt(2), # algebraische Zahl  
sin(3/2); # transzendente Zahl  
 $\sqrt{2}, \sin\left(\frac{3}{2}\right)$ 
```

aber man kann natürlich damit rechnen:

```
> sqrt(2)*sqrt(2), arcsin(sin(3/2));  
 $2, \frac{3}{2}$ 
```

bzw. numerische Approximation anfordern:

```
> evalf(sqrt(2)), sqrt(2.0);
1.41421356237309504880, 1.41421356237309504880
```

```
> evalf(sin(3/2)), sin(1.5); # Bogenmaß!
0.99749498660405443094, 0.99749498660405443094
```

-- Rationale Zahlen werden automatisch gekürzt, aber nicht automatisch als Dezimalbruch dargestellt, auch wenn dieser endlich ist, es sei denn, man verwendet Notation mit Dezimalpunkt:

```
> a:=2/5; b:=2./5; convert(a-b,rational);
a := 2
5
b := 0.40000000000000000000
0
```

```
> ? convert # diverse Konversionen
```

-- Viele Ergebnisse sind in einem 'generischen Sinn' zu interpretieren, d.h. Maple geht davon aus, was 'normalerweise' - außer in Spezialfällen - zutrifft.

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

-- Nicht alle naheliegenden Vereinfachungen erfolgen automatisch; **simplify** versucht hier sein bestes zu geben:

```
> simplify(sin(x)^2+cos(x)^2);
1
```

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

```
> simplify(p);
x^2 - 3 x + 2
```

-- **factor** bzw. **expand** faktorisiert bzw. multipliziert aus:

```
> factor(%);
(x - 1) (x - 2)
```

```
> expand(%);
x^2 - 3 x + 2
```

-- **collect** ordnet nach Potenzen einer Variablen:

```
> collect(%,x);
x^2 - 3 x + 2
```

-- Vorsicht bei Reihenentwicklungen, z.B. geometrische Reihe oder Taylorreihe (**sum** bedeutet 'symbolische' Summation, analog zu **int**, im Gegensatz zu **add** !)

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

```
> taylor(1/(1-x),x=0); # Konvergenzradius ist 1!
1 + x + x^2 + x^3 + x^4 + x^5 + O(x^6)
```

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

```
> u+u; Re(%); # Re=Realteil
2 u
2 ℑ(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
```

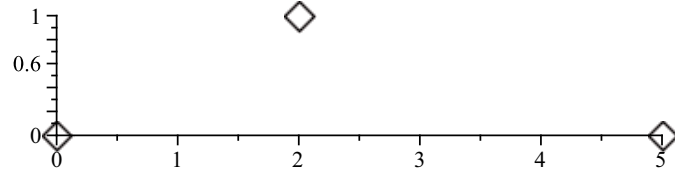
```
> Re(u),Im(u); # Im=Imaginärteil
u~, 0
```

-- Vieles ist in eigene **packages** ausgelagert, z.B. für diverse Plot-Tools:

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

```
[animate, animate3d, animatecurve, arrow, changecoords, complexplot, complexplot3d, conformal,
conformal3d, contourplot, contourplot3d, coordplot, coordplot3d, densityplot, display,
dualaxisplot, fieldplot, fieldplot3d, gradplot, gradplot3d, implicitplot, implicitplot3d, inequal,
interactive, interactiveparams, intersectplot, listcontplot, listcontplot3d, listdensityplot, listplot,
listplot3d, loglogplot, logplot, matrixplot, multiple, odeplot, pareto, plotcompare, pointplot,
pointplot3d, polarplot, polygonplot, polygonplot3d, polyhedra_supported, polyhedraplot,
rootlocus, semilogplot, setcolors, setoptions, setoptions3d, spacecurve, sparsematrixplot,
surfdata, textplot, textplot3d, tubeplot]
```

```
> pointplot ([ [0,0], [2,1], [5,0] ], symbolsize=32, scaling=
constrained);
```



oder: `plots[pointplot](...)`

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
-- z.B.: Dokumentation eines packages:

> ? plots

-- Weitere Beispiele für Auswertungen:

> evalf(Pi*c); # Hier: Ausgabe auf 20 Dezimalstellen ist
eingestellt
# (interne Rechengenauigkeit ist separat
einstellbar)
3.14159265358979323846 c

> evalc(1/(1+I)); # I = imaginäre Einheit
1/2 - 1/2 I
```

2. Elemente der Maple-Sprache

2.1 Zeichensatz

-- Groß/Kleinschreibung ist signifikant:

```
> A1, a1, evalb(A1=a1);
```

A1, a1, false

```
> my_var01; # zulässiger Name (mit _)
my_var01
```

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

```
> endl;
```

_ endl

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

α, Ω

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

π

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

π = π

false

-- **Reservierte** Namen sind Befehle der Maple-Sprache:

```

> 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 := ∞
∞

```

> `gelegentlich mag auch so ein doofer name praktisch sein`
:= 0;
gelegentlich mag auch so ein doofer name praktisch sein := 0

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

```

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

```

```

> %,%%,%%%;
3, 2, 1

```

```

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

```

-- **Beachte:** Ein Name ist nicht dasselbe wie eine Zeichenkette (string):

```

> whattype(`text`), whattype("text");
symbol, string

```

-- **Mehr über Namen, Voreinstellungen, etc.:**

```

> restart;

```

Indizierte Namen (systematisch verwenden für indizierte Objekte!)

```

> a[0], whattype(a[0]); # 0 Index wird tiefgestellt angezeigt
a0, indexed

```

Auch mehrfache und symbolische Indizes erlaubt:

```

> a[0], b[1,1], c[heute] := 0, 1, 2;
a0, b1, 1, cheute := 0, 1, 2

```

Vordefinierte Namen: siehe

```

> ? ininames

```

Zum Beispiel:

```

> I, gamma, Pi; # aber: Euler'sche Zahl e nicht vordefiniert!
I, γ, π

```

```

> exp(I*Pi), e^(I*Pi), evalf(gamma);
-1, eIπ, 0.57721566490153286061

```

-- Vieles kann man auch über **Tools/Options** einstellen.

```

> Digits; # eingestellte Gleitpunkt-Rechengenauigkeit;
30

```

```

> Digits:=3;
Digits := 3

```

```

> evalf(Pi);
3.14000000000000000000

```

```

> Digits:=30;
Digits := 30

```

```

> evalf(Pi); # Ausgabegenauigkeit hier auf 20 Stellen
eingestellt
3.14159265358979323846

```

2.3 Spezielle Zeichen und ihre syntaktische Bedeutung, z.B.

: , () ' ' [] { }

-- Zuweisung mit ; man kann einem beliebigen Objekt einen Namen zuweisen:

```

> dingsbums := a+a+b-int(sin(v),v);
> dingsbums;
2 a + b + cos(v)

```

: statt ; unterdrückt Ausgabe.

```

> (x+y)*z; expand(%); # Klammerung
(x + y) z
zx + zy

```

```

> 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' Ausdrucks (mit Apostrophen)**
entfernt genau ein Paar von Apostrophen.

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

> 'x'; %; %;
'x'
x
1

> s := a,b,c; whattype(%); # eine FOLGE (exprseq;
Basiskonstrukt)
s := a, b, c
exprseq

> s[1];
a

> 1, seq(i^2,i=1..5); # seq-Befehl generiert Folge mittels
`impliziter Schleife'
1, 1, 4, 9, 16, 25
```

[...] und {...} dienen zur Konstruktion von **Listen** und **Mengen**
(siehe Teil I und Abschnitt 4 unten):

-- **Listen** sind statisch allokierte, geordnete lineare Folgen von Objekten in der Form [Folge],
Indizierung beginnt mit 1.

```
> primes:=[2,3,5,7]; primes[1];
primes := [2, 3, 5, 7]
2
```

```
> ? 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}; M[2]; # Vorsicht
M := {2, 3, 5, 7}
3
```

```
> ? set
```

▼ 2.4 Typen und Operanden

Jedes Maple-Objekt gehört einem oder mehreren von (vielen) **Typen** an. Typ-Deklaration ist
jedoch im allgemeinen nicht erforderlich, und mit Neuzuweisung einer Variablen kann sich auch ihr Typ
ändern.

Typ-Überprüfung wird in Prozeduren verwendet, die übergebene Parameter auf ihre Zulässigkeit
prüfen sollen.

-- Beispiele für Typen:

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

> type('x',name), type('x',integer);
true,false

> whattype(1);
integer
```

-- Jedes Maple-Objekt besteht aus einem oder mehreren **Operanden** (interne Baumstruktur).

Zugriff mittels **op(...)**; benötigt man manchmal bei der fortgeschritteneren Programmierung
zur Selektion der intern hierarchisch organisierten Teilausdrücke.

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));
'^', a, 2
```

```

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

```

3. Maple-Ausdrücke und Anweisungen

3.1 Ausdrücke

-- Beispiele für Konstanten:

```

> infinity;
∞
> sin(1);
sin(1)
> 3.14E-3;
0.00314000000000000000
> 4/6; # wird automatisch gekürzt
2/3
> numer(%), denom(%); # Zähler, Nenner
2, 3
> 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(4/3)

```

-- Einsetzen von Werten in einen Ausdruck: subs

```

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

```

-- Verwendung arithmetischer Operatoren:

```

> x^y; # Potenz
x^y
> I^I: simplify(%);
e^(-1/2 π)

```

etc.

-- **Range-Operator:** ('von a bis b')
(wird z.B. für Schleifenkonstrukte verwendet):

```

> 1..5;
1..5
> 'sum(i,i=1..5)'; %;
∑i=15 i
15
... 1..5 ist aber nicht dasselbe wie die Folge 1,2,3,4,5.

```

-- Diverses:

```

> 5!; # Faktorielle
120
> 11 mod 3; # modulo
2

```

-- Vergleichsoperatoren:

```

> 1<2; whattype(%);
1 < 2
'<'
> evalb(1<2); # 'evaluate boolean'
true
> 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(%); # zum Überprüfen verwendet 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; # hier 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 bedeutet unentscheidbar bzw.
undefiniert
FAIL
-- Die Funktion is fragt: Ist das wahr? - Beispiele:
> is(1,positive);
true
> is(I,integer);
false

```

3.2 Die wichtigsten Datenstrukturen

```

-- Folge (exprseq), Liste (list), Menge (set): bereits erwähnt
-- Weiters: Tabelle (table), Feld (Array)

table: Eine Tabelle ist ein assoziatives Feld; dynamische Allokation:
> 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([Januar=31,Februar=28])
> 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.

Die tatsächliche Anzahl der Einträge erhält man hier mittels
> nops(eval(days)); # eval!
2

-- Auch: mehrdimensionale Tabellen:
> sudokufarbe:=table();
sudokufarbe:=table([ ])
> sudokufarbe[eins,zwei]:=grün;
sudokufarbe_eins,zwei:=grün
> eval(sudokufarbe);
table([(eins,zwei)=grün])

-- 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:= [ 0 0 ]

```


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  
zeilenweise 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
```

```
> restart;
```

3.3 Funktionen, Prozeduren

--Achtung: Eine Definition der Form

```
> f(t) := t^2;
```

$$f(t) := t^2$$

definiert **keine Funktion**, sondern eine Variable mit Namen **f(t)**:

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

$$t^2, f(1)$$

-- Für Funktion: 'Arrow'-Notation verwenden:

```
> f:=t->t^2; eval(f); f(1);
```

$$f: t \rightarrow t^2$$
$$t \rightarrow t^2$$
$$1$$

-- Das ist eine 'unspezifizierte' Funktionsauswertung:

```
> f:='f':
```

```
> f(s);
```

$f(s)$

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

$$\frac{d}{ds} f(s)$$

-- Eine Funktion in 2 Variablen:

```
> f := (x,y)->x+x-y;  
f(u,v);
```

$$f: (x,y) \rightarrow 2x - y$$
$$2u - v$$

--Auswertungsregel:

Zuerst wird f symbolisch ausgewertet,
dann werden Argumente ausgewertet und eingesetzt
(call by value)

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

$$2 \sin(1) - \cos(2)$$

... Das könnte man auch als Prozedur schreiben:

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

$$f := \text{proc}(x,y) \ x - y \ \text{end proc}$$

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

$$\text{proc}(x,y) \ x - y \ \text{end proc}$$

procedure

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

$$1 - \text{irgendwas}, \text{irgendwas} - 1$$

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

Error, invalid input: f uses a 2nd argument, y, which is missing

-- 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 notwendig)
```

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

-- **Funktionen und Prozeduren** werden häufig in den Übungen verwendet. Im einfachsten Fall übergibt man einfach ein oder mehrere Parameter, die die Prozedur beim Aufruf abarbeitet. Das innerhalb der Prozedur zuletzt berechnete Objekt wird zurückgegeben, bzw. Verwendung von **return**.

-- **Definition einer Prozedur:**

```
> myint := proc(f,var)
  local C, indefinite_integral; # lokale Variablen
  indefinite_integral := int(f(var),var)+C;
  return Int(f(var),var)=indefinite_integral
end proc;
myint := proc(f,var)
  local C, indefinite_integral;
  indefinite_integral := int(f(var),var) + C; return Int(f(var),var) = indefinite_integral
end proc
```

-- **Aufruf:**

```
> myint(sin,t);  $\int \sin(t) dt = -\cos(t) + C$ 
> myint(t->sin(sqrt(t)),y);  $\int \sin(\sqrt{y}) dy = 2 \sin(\sqrt{y}) - 2\sqrt{y} \cos(\sqrt{y}) + C$ 
```

3.4 Beispiel für eine allgemeinere Prozedur

Wir schreiben eine Prozedur, die nicht sehr viel tut, wo jedoch rein syntaktisch ziemlich viel vorkommt (genauer später)

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

  if (p<s) then # Selektion (if)
    print("a*b ist kleiner als a+b")
```

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

printf("\n");
for i from a*b to 1 by -1 do # Schleife abwärts
  printf("%d ",i) # das ist eine 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 # setze Schleife fort
  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;
```

```
> my_proc(a,b);
Error, invalid input: my_proc expects its 1st argument, a, to be of type posint, but received a
```

Zu beachten: Bei Auftreten eines Syntaxfehlers bei der Definition der Prozedur erscheint eine Fehlermeldung, und Cursor springt zu der Zeile, wo das Problem festgestellt wurde.

Aber: Wo ein **Laufzeitfehler** (siehe oben) genau aufgetreten ist, wird leider nicht automatisch angezeigt.

Zur Fehlersuche in längeren Prozeduren benötigt man den **Debugger** (später).

Bemerkungen:

-- Es gibt formal nur **einen** Rückgabewert; dieser kann aber im Prinzip ein beliebiges Maple-Objekt sein, z.B. auch eine `exprseq` (für Rückgabe mehrerer Werte)

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

-- **Lokale** Variablen sind Prozedur-intern

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

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(my_proc );
# meine super prozedur
my_proc( a::posint, b::posint )

> my_proc(3,4);
"a*b ist größer als a+b"

12 11 10 9 8 7 6 5 4 3 2 1

> my_proc(11,2); # provoziert 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 my_proc) a zu gross!!
> my_proc(10,1);
"a*b ist kleiner als a+b"

10 9 8 7 6 5 4 3 2 1

Error, (in my_proc) numeric exception: division by zero
> restart:
```

4. Praktischer Umgang mit Datenstrukturen

4.1 Mengen (set)

```
> s1 := {seq(-i,i=1..5)};
s2 := {-2,-2,-1};
s1 := {-5, -4, -3, -2, -1}
s2 := {-2, -1}

> s1 union s2, s1 intersect s2, s1 minus s2;
{-5, -4, -3, -2, -1}, {-2, -1}, {-5, -4, -3}

> member(-1,s1);
true

> max(s1); # maximales Element
-1

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

> f:=x->x^2; map(f,s1);
f:=x→x2
{1, 4, 9, 16, 25}

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

-- Eine allgemeinere Variante von map übergibt ein Argument mit 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 alles nach y
{-sin(y), cos(y)}

-- Oder: Verwende Variante des for-Befehls, um eine Operation auf alle
Elemente einer Menge (oder Liste, etc.) anzuwenden:

> for e in s1 do e^2 end do; # praktisch
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
> L:=op(L),6]; # Element dazugeben
      L := [1, 2, 3, 4, 5, 6]

--map, for .. in funktionieren analog wie bei sets:
> map(exp,L);
      [e, e2, e3, e4, e5, e6]

-- 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, 36]
> member(9,L);
      true
> member(9,L,'position'); position;
# Element suchen; 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 (insbesondere Vector, siehe unten).

Aber immerhin: Elementare Vektorrechnung,
d.h. Linearkombination von Listen funktioniert:
> [a,b,c]+2*[x,y,z];
      [2x + a, 2y + b, 2z + c]

Siehe auch
> ? ListTools
```

4.3 Tabellen (table)

Eine Tabelle (assoziatives array) 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 hinzufügen:
> t1[infinity]:=unendlich; eval(t1);
      t1 ∞ := unendlich
      table([1 = cos, 2 = sin, 3 = tan, ∞ = unendlich])
```

4.4 Felder (Array)

(Statische Allokation (d.h. Initialisierung erforderlich); Arrays beruhen auf einer alternativen, speicher-flexibleren Implementierung von tables (sogenannte 'rectangular tables', **rtables**; in etwa vergleichbar den arrays in MATLAB, aber mit flexiblere interner Speicherstruktur).

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)
> a2:=Array(0..2, 0..2);
a2 := Array(0..2, 0..2, { }, datatype = anything, storage = rectangular, order = Fortran_order)
> a2[0,0]:=x; eval(a2);
      a20,0 := x
Array(0..2, 0..2, {(0,0) = x}, datatype = anything, storage = rectangular, order
= Fortran_order)
```

Hier sind die anderen Elemente (noch) undefiniert.

```
> A1:=Array(0..3); # hier: automatisch Initialisierung mit 0
A1 := Array(0..3, { }, datatype = anything, storage = rectangular, order = Fortran_order)
> A1[1];
      0
> eval(A1); # das ist etwas seltsam
      Array(0..3, { }, datatype = anything, storage = rectangular, order = Fortran_order)
> 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)
> A2:=Array([[1,2,3], [4,5,x]]): eval(A2);
```

```
# so sieht es vernünftiger aus
```

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

-- Anmerkung:

Bei Datenstrukturen herrscht in Maple ein gewisser 'Wildwuchs' (z.T. aus historischen Gründen und zwecks Abwärtskompatibilität), mit teilweise ganz spezifischen Zugriffsfunktionen.

Hier wird nicht alles besprochen; manche packages verwenden ihre eigenen Datenstrukturen.

-- Empfehlung:

- Verwende **set** für endliche Mengen
- Verwende **list** für (endliche) Folgen von Objekten
- Verwende **table** bei Bedarf für assoziative Arrays
- Verwende **Array** für 'Vektoren' bzw 'Matrizen', falls man flexible Indizierung wünscht (z.B. von 0 weg)

und

- Verwende **Vector** und **Matrix** für Lineare Algebra im \mathbf{R}^n , \mathbf{C}^n . Dies sind eigentlich spezielle Arrays (Index beginnt mit 1); einfacher im Handling als allgemeine Arrays; viele spezielle Funktionen.

Vector und **Matrix** sind für das Rechnen mit Vektoren und Matrizen (sowohl symbolisch als auch numerisch) relevant. Diese werden später bei der Diskussion des **LinearAlgebra** - Packages genauer besprochen.

=== Ende Maple / Teil II ===