Eine Übersicht über die 18 Worksheets dieses Kurses:

session1.mws	(erster Kontakt)
session2.mws	(Mengen, Listen)
session3.mws	(Summen, Produkte, Operanden, Substitution)
session4.mws	(Typen)
session5.mws	(solve, fsolve)
session6.mws	(Dokumentation, Editieren von Worksheets)
session7.mws	(**Aufgabe 1**: Anzahl von Bäumen, Abgabe 23.5.97)
session8.mws	(table, array, matric, vector)
session9.mws	(Prozeduren)
session10.mws	(mehr Prozeduren, **Aufgabe 2**: Additionen, Abgabe 30.5.97)
session11.mws	(Graphik)
session12.mws	(**Aufgabe 3**: Numerik der Poissongleichung, Abgabe 6.6.97)
session13.mws	(Anwendung: Diskrete Fourier-Transformation, Bildverarbeitung)
session14.mws	(**Aufgabe 4**: symbolische Integration, Abgabe 20.6.97)
session15.mws	(Diverses: collect, simplify, factor, ...)
session16.mws	(Die Pakete linalg und grobner)
session17.mws	(**Aufgabe 5**: Runge-Kutta-Verfahren, Abgabe 4.7.97)
session18.mws	(**Aufgabe 6**: Baker-Campbell-Hausdorff-Formel, Abgabe 18.7.97)

Die Schwierigkeitsgrade der Aufgaben schätze ich folgendermassen ein:

Aufgabe 1:	10 Punkte
Aufgabe 2:	3 Punkte
Aufgabe 3:	15 Punkte
Aufgabe 4:	30 Punkte
Aufgabe 5:	15 Punkte
Aufgabe 6:	30 Punkte

Eingereichte akzeptable Lösungen im Wert von etwa 45 Punkten werden mit dem Praktikumsschein zu diesem Kurs belohnt.

Organisatorisches

Ziele des Kurses: Einführung in den Umgang mit Computeralgebra (CA) anhand des Systems MAPLE V4. Es geht weniger um die mathematischen Hintergründe der algebraischen Algorithmen.

Vorlesung: 2-stündige wöchentliche Vorlesung, Do 14–16, im H6. Hier wird das System MAPLE V4 (bzw. wichtige Bausteine, Systemfunktionen, Pakete etc.) vorgestellt und diskutiert werden. Ein entsprechendes Skript wird entstehen und als kopierfähiges Exemplar in der Bibliothek des Mathematischen Seminars zur Verfügung stehen.

Betreute praktische Übungen finden im Raum 107 (Rechner-Pool) des FLAT-Gebäudes Robert-Mayer-Str. 1 statt. Termine:

Mo 10–12 , Di 10–12 , Mi 10–12, Do 10–12, ab Mo 21.4.97

Es wird angestrebt, daß jeder Teilnehmer die Möglichkeit hat, wöchentlich 2 dieser Termine wahrzunehmen. Da nur 10 Maschinen vorhanden sind, wird eine zwingende Zuordnung der Teilnehmer zu diesen Terminen per Aushang an meinem Büro 101, Robert-Mayer-Str. 10, erfolgen. Benötigt werden

User-Login `flcaXX` und voreingestelltes Password,

diese werden in der Vorlesung bzw. den Übungen verteilt werden.

Ablauf der Übungen: Einloggen als `flcaXX`, eröffne ein Terminal. Schaue den Inhalt des Directories `/home/fb12/kurse/flca00` an:

```
ls ../flca00
```

Kopiere, z.B. mittels

```
cp ../flca00/session1.mws session1.mws
```

etc. die unter `/home/fb12/kurse/flca00` bereitgestellten MAPLE-worksheets `session1.mws`, `session2.mws`, ... in den eigenen Home-Bereich. Eröffne eine X-MAPLE-Sitzung

```
maple -x &
```

und öffne das erste Worksheet `session1.mws`, das durchgearbeitet werden soll.

Im Laufe des Semesters werden weitere Worksheets zur Verfügung gestellt werden, die jeweils in den eigenen Bereich kopiert und bearbeitet werden sollen.

Leistungsschein: In die Worksheets sind eine Reihe von Programmier-Aufgaben eingearbeitet (und als solche kenntlich gemacht), die gelöst werden und an die Betreuer zur Korrektur abgeliefert werden sollen (Details: siehe Worksheets). Bei erfolgreicher Bearbeitung dieser Aufgaben wird als Leistungsschein ein *Praktikumsschein der Angewandten Mathematik* erteilt werden.

Installation auf eigenen Rechnern: MAPLE V4 ist für private Rechner von Studierenden der Uni Ffm kostenlos verfügbar, momentan sind Versionen für MacIntosh und PCs unter Windows vorhanden (die Linux-Version wird erwartet):

Mac-Beratung (MacOS)	Raum 109a, Gräfstr. 38, Di-Fr 12-14
HRZ-Beratung (Windows)	Raum 222, FLAT1: Robert-Mayer-Str. 1, Mo-Fr 9:30-15:30

Die Worksheets in `/home/fb12/kurse/flca00` (dies sind ASCII-Dateien) sind auch unter diesen MAPLE V4-Versionen verwendbar.

Computeralgebra und Anwendungen

Skript zum MAPLE-Kurs, JWG-Universität Frankfurt, SS 97

W. Oevel

Inhaltsverzeichnis

1 Einige verfügbare CA-Systeme

1.1 Die wichtigsten Systeme

“1.te Generation”

MACSYMA: der Dinosaurier unter den CA-Systemen: erstes großes general-purpose-System (etwa 30 Jahre alt), auf LISP basierend, war im Vergleich zu anderen Systemen sehr langsam und Speicherplatzhungrig. Hochentwickelte, exzellente Programmbibliotheken, sehr gute symbolische Fähigkeiten. Neuere Versionen sind auf moderner, schneller Hardware durchaus konkurrenzfähig mit den anderen Systemen.

“2.te Generation”

Die “großen” Systeme:

REDUCE: ursprünglich ein Spezialsystem für symbolische Rechnungen in der Hochenergie-Physik, das sich zum viel-benutzten general-purpose-System entwickelte. Auf LISP basierend, aber schneller und effektiver als MACSYMA, dafür deutlich reduzierte Programmbibliotheken. Gewisse Stärken in nicht-kommutativer Algebra (z.B. sind Vertauschungsrelationen von Operatoren einfach zu implementieren).

MATHEMATICA: das wohl bekannteste CA-System (dank einer starken Marketing-Strategie des Entwicklers). Höchstentwickeltes Interface (notebook-Konzept), beste graphische Fähigkeiten. Ausgezeichnetes Pattern-Matching. Gute symbolische Fähigkeiten, umfangreiche Bibliotheken. Sehr schnell bei numerischen Rechnungen. Hat (im Vergleich zu MAPLE) gewisse Schwierigkeiten bei großen, speicherplatzintensiven Rechnungen.

MAPLE: In vielen Bereichen schnellstes der großen general-purpose-Systeme. Sehr effektiver Umgang mit großen Datenmengen. Umfangreiche, hochentwickelte Programmbibliotheken, starke symbolische Fähigkeiten, gewisse Schwächen in nichtkommutativer Algebra. Effizienzprobleme bei numerischen Rechnungen. Das Interface hat sich per Worksheet-Konzept dem MATHEMATICA-Standard angenähert, gute graphische Fähigkeiten.

Einige kleinere Systeme:

MuMATH: symbolisches Paket ohne Programmiersprache, im Vergleich zu den “großen” Systemen nur rudimentäre symbolische Fähigkeiten.

DERIVE: Weiterentwicklung von MuMATH, einfache rudimentäre Programmiersprache, recht schnell. Kandidat für den Einsatz von CA in Schulen.

RIEMANN: recht effektives kleineres System für kleinere Maschinen, Programmbibliotheken und symbolische Fähigkeiten längst nicht so umfangreich wie die der "großen" Systeme.

Einige Spezialsysteme:

GAP, CAYLEY: Gruppentheorie

CoCoA: kommutative Algebra

Macaulay: algebraische Geometrie

DELiA, Lie: Lie-Gruppen-Analyse von Differentialgleichungen

usw.

"3.te Generation"

AXIOM: moderner Kern mit domain-Konzept, eigene Routinen können kompiliert werden, Anschluß an die numerische NAG-Bibliotheken. Spärliche Programmbibliotheken, schwerfällig.

MuPAD: moderner für Parallelverarbeitung ausgelegter Kern mit domain-Konzept, eigener kompilierter C-Code kann eingebunden werden. Programmbibliotheken in Entwicklung. Einziges großes general-purpose System, das kostenlos zur Verfügung steht.

Ein aktueller Vergleich der Windows95-Versionen von MACSYMA, MATHEMATICA (3.0), MAPLE (V4), AXIOM, MuPAD und DERIVE mit Laufzeitvergleichen für gewisse Testprobleme sind in dem Artikel

Barry Simon: *Symbolic Math Powerhouses Revisited*
Zeitschrift: Desktop Engineering, Vol. 2 (7), March 1997
Seiten 12–25 + 52,53

zu finden (siehe beigefügte Kopie).

1.2 MAPLE: Anmerkungen

Entwicklung der University of Waterloo (Canada) seit 1980, mittlerweile kommerziell. Erste praktisch brauchbare Releases (III) seit etwa 1985. Sprung von Release IV auf V mit wesentlichen Änderungen in 1990. Letzter Release V.4 in 1996.

Kern: sehr kleiner C-Kern (<300 kBytes), der den Interpreter, die Speicherverwaltung und eine Reihe schneller, oft gebrauchter Routinen enthält.

Iris: verwaltet Input (parser) und Output.

Bibliotheken: etwa 15 MBytes von in der MAPLE-Programmiersprache geschriebenen Routinen. Werden bei Bedarf entweder automatisch oder durch den Benutzer geladen.

1.3 Literatur

Die Standard-Referenz-Manuals zu MAPLE V4 sind [HHR 96, Red 96, Mon 96]:

Zur Einarbeitung:

[HHR 96] K.M. Heal, M.L. Hansen, K.M. Richard, *MAPLE V Learning Guide*, Springer, 1996.

Auflistung der MAPLE-Funktionen ("mathematisch" geordnet):

[Red 96] D. Redfern, *The MAPLE Handbook*, Springer, 1996.

Sehr ausführliche Einführung in MAPLEs Programmiersprache:

[Mon 96] M.B. Monagan et al., *MAPLE V Programming Guide*, Springer, 1996.

Zur Einarbeitung (nur bis Release V.2, aber ausführlicher als [HHR 96]):

[Cha 92] B.W. Char et. al., *First Leaves: A Tutorial Introduction to Maple V*, Springer, 1992.

Sehr Empfehlenswerte, ausführliche und gut lesbare Beschreibungen von MAPLE V:

[Hec 96] A. Heck, *Introduction to MAPLE*, 2nd edition, Springer, New York, 1996.

[Kof 96] M. Kofler, *MAPLE V Release 4*, Addison-Wesley, 1996.

Die wesentliche Referenz zu algorithmischen Hintergründen:

[GCL 92] K. O. Geddes, S.R. Czapor, G. Labahn, *Algorithms for Computer Algebra*, Kluwer, Dordrecht, 1992.

Weitere Referenz zu den Algorithmen:

[DST 88] J.H. Davenport, Y. Siret, E. Tournier, *Computer Algebra*, Academic Press, London, 1988.

Ein (nicht sehr aktueller) Vergleich der wichtigsten Systeme:

[DWH 91] D. Harper, C. Wooff, D. Hodgkinson, *A Guide to Computer Algebra Systems*, Wiley, Chichester, 1991.

2 MAPLE-Statements

Die Basis-Syntax für MAPLE-Eingaben ist

$$statement\,;\;\; < return >$$

bzw.

$$statement_1\,;\; statement_2\,;\;\; \ldots\,;\; statement_n\,;\;\; < return >$$

Nach dem $< return >$ werden die Statements ausgeführt ("evaluiert") und ausgegeben. Alternativer syntaktischer Trenner zwischen Statements ist : mit der Bedeutung: "evaluiere, aber gebe das Ergebnis nicht aus".

Beispiele:

`a:=1;`	$\longrightarrow$	$a := 1$
`b:=a+1:`	$\longrightarrow$	
`c:=b+17; e:=2*c: sin(e);`	$\longrightarrow$	$c := 19$
	$\longrightarrow$	
	$\longrightarrow$	$sin(38)$

Zulässige Statements	Beispiele
Neustart der Sitzung	`restart;`
Zuweisungen (assignments)	`a:=1; xy3:=x+f(y); x[4]:=b;`
Aufrufe von Variablen	`a;`
bzw. Ausdrücken	`x+y*10∧c;`
bzw. Funktionen	`f(x); help(seq);`

Schleifen	`for i from 12 to 20 do`
	`a[i]:=20*sqrt(i); print(i,a[i]);`
	`od;`
	`while i<10 do i:=i+1; od;`

if-Verzweigungen	`if x>0 then x∧2; else -x∧2; fi;`
Prozedur-Deklaration	`proc(x) x∧2; end;`
Eingabe aus Dateien	`read Dateiname;`
Ausgabe in Dateien	`save a,b,c, Dateiname;`
Kommentare	`a:=1; # b:=2;` `a,b;` ⟶ `1,b`
Schleifenabbruch	`break;`
Help-Aufruf	`?seq`
empty	`a; ; ; b;`
Ende einer Sitzung	`quit` (oder `done` oder `stop`)

3 Aufbau von MAPLE-Ausdrücken

MAPLE-Objekte werden als "Ausdrücke" (expressions) bezeichnet. Sie sind praktisch alle aus "Atomen" aufgebaut: dies sind Namen (strings) und ganze Zahlen, welche mittels Operatoren und Funktionsaufrufen zu komplizierteren mathematischen Strukturen zusammengesetzt werden.

3.1 Operatoren

Die MAPLE-Operatoren zum Aufbau von Ausdrücken sind:

`*,/,+,-`		`a*b; a/(b+c-d);`
`^`, `**`	(Potenz)	`a^b; x**y;`
`!`	(Fakultät)	`10!; x!;`

`,`	(Folgenverkettung)	`seq1:=a,b,c; seq2:=c,d;`
		`seq3:=seq1,seq2;`
		$\longrightarrow$ $seq3 := a, b, c, c, d$

`$`	(Folgengenerator)	`c:=a$b; b:=3 ; c;`
		$\longrightarrow$ $c := a\$b$
		$\longrightarrow$ $b := 3$
		$\longrightarrow$ a, a, a
		`i**2 $ i=3..5 ;`
		$\longrightarrow$ $9, 16, 25$

`.`	(Stringverkettung, Konkatenation)	`a.b` $\longrightarrow$ ab `b:=3; a.b;` $\longrightarrow$ $a3$
`..`	(Range)	`c:= 0..2; int(f(x), x=c );` `a:=array(5..10);`
`=`	(Gleichung)	`eq := x+y = 1 ;`
`<`, `<=`, `>`, `>=`, `<>`	(Ungleichungen)	`bedingung:= x>y;`
`‘ ‘`	(Stringbegrenzer)	`a:=‘ a strange name ‘;` `‘11+1‘ := 13;`
`’ ’`	(verzögerte Evaluierung)	`a:=1;c:='2*a';` $\longrightarrow$ $c := 2\,a$ `a:=1;` $\longrightarrow$ $a := 1$ `a:='a';` $\longrightarrow$ $a := a$ `a;` $\longrightarrow$ a
`[ ]`	(Listenbegrenzer)	`Liste:=[a,b,c,d];`
`{ }`	(Mengenklammern)	`Menge:={a,b,b};` $\longrightarrow$ $Menge := \{a, b\}$ `empty:={ };`
union minus intersect	(Mengenoperationen)	`{a,b}union{b,c};` $\longrightarrow$ $\{a, b, c\}$
and, or, not	(Boolsche Operatoren)	`condition:= x>0 and y<0 ;`
`->`	(Abbildung)	`g:= x -> x^2;` $\longrightarrow$ $g := x \; {-}{>} \; x^2$ `f:=g;` $\longrightarrow$ $f := g$ `f(x);` $\longrightarrow$ x^2;
`@`	(Komposition)	`h:= f@g; h(x);` $\longrightarrow$ $f(g(x))$
`@@`	(Iteration)	`g:=f@@3;` $(= f \circ f \circ f)$ `g(x);` $\longrightarrow$ $f(f(f(x)))$

$\boxed{\text{Weitere Ausdrücke lassen sich durch Funktionsaufrufe erzeugen.}}$

Die meisten Operatoren sind verkappte Funktionsaufrufe, z.B.,

$$\mathtt{a}+\mathtt{b}+\mathtt{c} \mathrel{\widehat{=}} `+`(\mathtt{a},\mathtt{b},\mathtt{c})\ , \quad \mathtt{a!} \mathrel{\widehat{=}} \mathtt{factorial}(\mathtt{a})\ ,$$

$$\mathtt{i}^2\$\mathtt{i} = 1..3 \mathrel{\widehat{=}} \$(\mathtt{i}^2, \mathtt{i} = 1..3)\ .$$

Einige Verknüpfungen, die mathematisch keinen Sinn machen, sind nicht zulässig, z.B.,

$$1\ \mathtt{union}\ \{\mathtt{a},\mathtt{b}\} \quad \longrightarrow \quad Error\ , \quad 3\ \mathtt{and}\ \mathtt{x} \quad \longrightarrow \quad Error\ .$$

Ein symbolischer Ausdruck `a and b;` (mit Namen a,b) ist jedoch zulässig (kann aber nicht als boolscher Ausdruck ausgewertet werden, solange nicht für a und b boolsche Ausdrücke eingesetzt werden).

Anmerkungen zur Stringerzeugung: *Wenn a ein Name ist, so werden die Stringbegrenzer von* `‘a‘` *abgestreift, der verbleibende Name a wird dann evaluiert:*

`a:=1;`	$\longrightarrow$	$a := 1$
`b:=‘a‘;`	$\longrightarrow$	$b := 1$
`b-1;`	$\longrightarrow$	0 ;
`g:=x+y: ‘g‘+x;`	$\longrightarrow$	$2x + y$

Aber:

`a:=1: b:=‘a+1‘-1;`	$\longrightarrow$	$b := a + 1\ -\ 1$
`‘x+y‘+x;`	$\longrightarrow$	$x + y\ +\ x$

Zur Stringkonkatenation: in `x.i;` *wird i evaluiert, aber* `x` *nicht:*

`x:=y: i:=1: x.i;` $\longrightarrow$ $x1$

Soll auch der linke Ausdruck evaluiert werden, so benutze den leeren String `‘‘`*:*

`x:=y: i:=1: ‘‘.x.i;` $\longrightarrow$ $y1$

Die Operatoren haben unterschiedliche **Bindungsstärken**:

`a.b(x);`	bedeutet	`(a.b)(x) ( = ab(x) )`
`a∧b(x)`	bedeutet	`a∧(b(x))`
`a*b∧c`	bedeutet	`a*(b∧c)`
`a+b*c`	bedeutet	`a+(b*c)`
`a+b(i)$i=2..3`	bedeutet	`(a+b(i))$i= 2..3` $\longrightarrow$ $a+b(2), a+b(3)$
`a=b,c`	bedeutet	`(a=b) , c`
`a,b$3`	bedeutet	`a,(b$3)` $\longrightarrow$ a,b,b,b
`[a,b]`	bedeutet	`[(a,b)]`
`{[a,b]}`	bedeutet	`{([(a,b)])}`
usw,		

also mit $\prec$ (= "schwächer bindend"):

$$\left.\begin{array}{c} \{\,\} \\ [\,] \\ '\,' \\ `\,` \end{array}\right\} \prec \;,\; \prec \;\$\; \prec \;=\; \prec \;+\; \prec \;*\; \prec \;\wedge\; \prec \textit{Funktionsaufruf} \prec \;.$$

Es können beliebig Klammern zur Erhöhung der Bindung gesetzt werden:

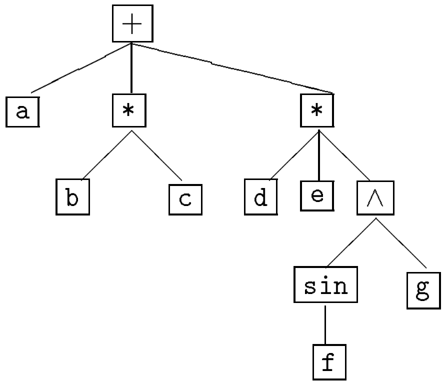
`i:=2: x.i∧2 , x.(i∧2);`	$\longrightarrow$	x_2^2 , x_4
`a*f(x),(a*f)(x);`	$\longrightarrow$	$a\,f(x)$, $a(x)\,f(x)$

3.2 Darstellungs-Bäume

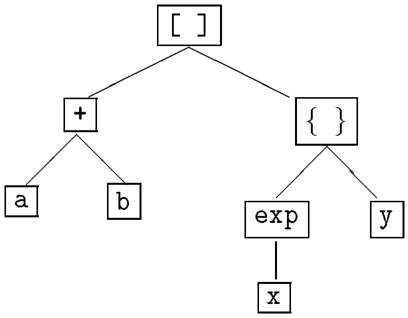
Eine Modell-Vorstellung, die die interne Darstellung eines MAPLE-Ausdrucks annähert, ist die des **Darstellungs-Baums** (expression tree). Die Operatoren (bzw. Funktionsnamen) werden als Knoten interpretiert, die Argumente bilden davon verzweigende Unterbäume. Der Operator mit der geringsten Bindungsstärke definiert die Wurzel.

Beispiele:

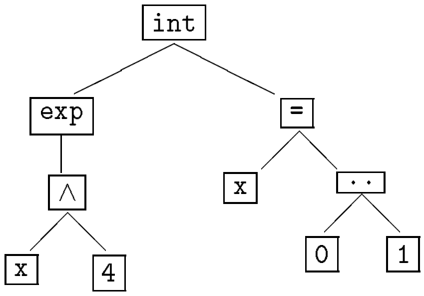
`a + b*c + d*e*sin(f)∧g;`



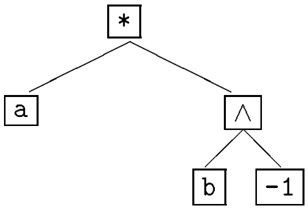
`[ a+b , {exp(x),y} ];`



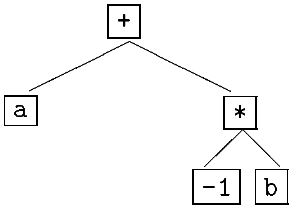
`int(exp(x∧4),x=0..1);`



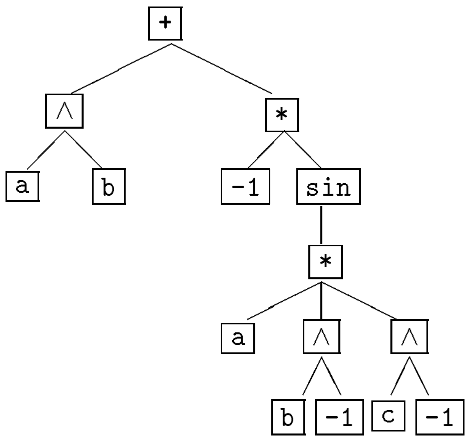
`a/b;`



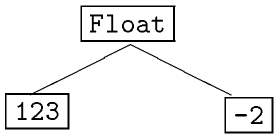
`a-b;`



`a∧b-sin(a/b/c);`



`1.23;` $(= 123 \times 10^{-2})$



Beispiel: `Float(1,-1)+Float(2,-3);` $\longrightarrow$ .102

Die Endknoten der Darstellungsbäume sind die MAPLE-“Atome”: sie sind Strings oder Integer (positive und negative).

3.3 Operanden

Die Teilbäume eines Darstellungsbaums heißen die **Operanden** des Ausdrucks. Die Systemfunktionen zur Bestimmung der Operanden eines Ausdrucks *expr* sind:

`nops(expr);`	(number of operands) Anzahl der Operanden
`op(i,expr);`	i.ter Operanden (i=1,...,nops(expr))
`op(i..j,expr);`	Folge der Operanden i,...,j: op(i,expr),...,op(j,expr)
`op(expr);`	Folge aller Operanden: op(1,expr),..., op(nops(expr),expr)

Beispiel:

`nops(a+b+sin(x));`	$\longrightarrow$	3
`op(3,a+b+sin(x));`	$\longrightarrow$	$sin(x)$
`op(1..2,a+b+sin(x));`	$\longrightarrow$	a, b
`op(a+b+sin(x));`	$\longrightarrow$	$a, b, sin(x)$

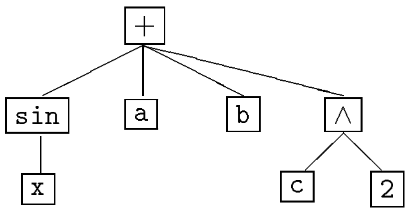
Zusätzlich existiert `op(0,expr);` Dies ist der Typ-Name von *expr*, z.B.,

`op(0,a+b*c );`	$\longrightarrow$	‘+‘
`op(0,a∧(b+c) );`	$\longrightarrow$	‘∧‘
`op(0,{a,b} );`	$\longrightarrow$	*set*

Falls *expr* ein <u>unevaluierter</u> Funktionsaufruf ist, so liefert `op(0,expr);` den Funktionsnamen:

`op(0, sin(x) );`	$\longrightarrow$	*sin*
`op(0, int(exp(x∧4),x) );`	$\longrightarrow$	*int*
`op(0, diff(y(x),x) );`	$\longrightarrow$	*diff*

Beispiel: `expr:=sin(x)+a+b+c∧2;`



ist vom Typ '+'

`op(0,expr);` $\longrightarrow$ '+'

und hat die Operanden

`op(1,expr);` $\longrightarrow$ $sin(x)$
`op(2,expr);` $\longrightarrow$ a
`op(3,expr);` $\longrightarrow$ b
`op(4,expr);` $\longrightarrow$ c^2

Achtung: Teilsummen wie $a+b$ sind nicht Unterbäume des Darstellungsbaums von *expr* und damit keine Operanden! Dies wird relevant in der Erklärung der Funktionsweise der überaus wichtigen Kernfunktion `subs` ("substituiere"), die Teilbäume eines Ausdrucks gegen andere Ausdrücke austauscht.

Beispiel:

`subs(x=y+exp(x),sin(x)+(y+x)∧2);` $\longrightarrow$ $sin(y+e^x)+\left(2\,y+e^x\right)^2$
`subs(1+exp(x)=cos(x),a+sin(1+exp(x)));` $\longrightarrow$ $a+sin(cos(x))$

Rekursiv wird durch `subs(old=new,expr);` der Darstellungsbaum von *expr* durchsucht und jeder Teilausdruck (Teilbaum), der *old* entspricht, wird durch *new* ersetzt. Dies erklärt

`subs( x+y=1 , x+y+sin(x+y) );` $\longrightarrow$ $x+y+sin(1)$

In der Summe *x+y+sin()* ist *x+y* kein Teilbaum und wird daher nicht ersetzt. Es gibt aber die Bibliotheksfunktion *algsubs* (algebraische –statt syntaktische– Substitution), die Teilsummen ersetzen kann.

Die Systemfunktion *op* operiert auf allen MAPLE-Ausdrücken mit Ausnahmen von Folgen (Typ "expression seq"). Hiermit kann jeder Ausdruck bis in seine Atome zerlegt werden.

Beispiel:

```
expr:= sin(x+sin(a+b)):
```

`op(1,expr);`	$\longrightarrow$	$x + sin(a+b)$
`op(1,op(1,expr));`	$\longrightarrow$	x
`op(2,op(1,expr));`	$\longrightarrow$	$sin(a+b)$
`op(1,op(2,op(1,expr)));`	$\longrightarrow$	$a+b$
`op(2,op(1,op(3,op(1,expr))));`	$\longrightarrow$	b

Weiterhin gilt

`op(-1,expr) = op(nops(expr),expr)`	= letzter Operand
`op(-2,expr) = op(nops(expr)-1,expr)`	= vorletzter Operand

usw.

Für Mengen, Listen existiert als Alternative zu *op* der **Auswahloperator**, ein nachgestelltes [ ]. Er funktioniert auch für Folgen.

Beispiel:

```
Menge:={a,b,c}; Liste:=[a,b,c]; Folge:=a,b,c;
```

`Menge[2],Liste[2],Folge[2];`	$\longrightarrow$	b , b , b
`Menge[1..2],Liste[1..2],Folge[1..2];`	$\longrightarrow$	$\{a,b\}$, $[a,b]$, a,b
`Menge[-1],Liste[-2],Folge[-3];`	$\longrightarrow$	c , b , a

Achtung: Es gibt keine Kontrolle der Reihenfolge der Operanden in Summen, Produkten, Mengen:

`sin(x+y*z)+cos(z*y+x);`	$\longrightarrow$	$sin(x + y\,z) + cos(x + y\,z)$
`a:=[x,y]; b:={y,x}; c:=[y,x];`	$\longrightarrow$	$a := [x,y]; b := \{x,y\}; c := [y,x]$

Der Grund ist, daß Ausdrücke in jeder MAPLE-Sitzung intern nur einmal gespeichert werden (der Vergleich von Daten ist durch diesen Mechanismus in MAPLE sehr schnell). Mathematisch identische Summen, Produkte, Mengen werden so angeordnet, daß sie durch dieselben internen Daten gespeichert werden können. Die Anordnung in Summen, Produkten, Mengen wird damit durch den zeitlichen Ablauf der MAPLE-Sitzung bestimmt.

Achtung: MAPLEs Reihenentwickler *taylor* und *series* erzeugen spezielle Datentypen:

`expr:= a7*x^7+a10*x^10+a11*x^11;`		
`t:=taylor(expr,x=0,11);`	$\longrightarrow t := a_7\,x^7 + a_{10}\,x^{10} + O(x^{11})$	
`op(0,t);`	$\longrightarrow x$	(Potenzreihe in x)
`op(1,t);`	$\longrightarrow a_7$	(führender Koeffizient)
`op(2,t);`	$\longrightarrow 7$	(führende Ordnung)
`op(3,t);`	$\longrightarrow a_{10}$	(2.ter Koeffizient)
`op(4,t);`	$\longrightarrow 10$	(2.te Ordnung)
`op(5,t);`	$\longrightarrow O(1)$	(Koeffizient des Abbruchterms)
`op(6,t);`	$\longrightarrow 11$	(Ordnung des Abbruchterms)

Die Ausgabe kann wie eine Summe aussehen, ist aber stets eine Reihe:

`t:=taylor(a7*x^7+a8*x^8,x=0,9);`	$\longrightarrow t := a_7\,x^7 + a_8\,x^8$	
`op(0,t);`	$\longrightarrow x$	(Potenzreihe in x)
`op(1,t);`	$\longrightarrow a_7$	(führender Koeffizient)
`op(2,t);`	$\longrightarrow 7$	(führende Ordnung)
`op(3,t);`	$\longrightarrow a_8$	(2.ter Koeffizient)
`op(4,t);`	$\longrightarrow 8$	(2.te Ordnung)
`op(5,t);`	$\longrightarrow$ *Error*	
`t- a7*x^7-a8*x^8;`	$\longrightarrow (a_7\,x^7 + a_8\,x^8) - a_7\,x^7 - a_8\,x^8$	

Achtung: MAPLEs multivariate Taylor-Entwicklung *mtaylor* (zu laden mittels `readlib(mtaylor);`) erzeugt Polynome (d.h. Summen), keine Reihen!

3.4 Typen

MAPLE-Ausdrücke werden ihrer mathematischen Bedeutung nach in Typen eingeteilt. Der Typ einer Variable braucht jedoch nicht deklariert zu werden, jeder Variable kann ein Ausdruck beliebigen Typs zugewiesen werden (optional kann der Typ von Eingabeparametern an Prozeduren deklariert werden). Ein Ausdruck kann mehrere Typen haben. Zuständig für die Typ-Kontrolle sind die MAPLE-Funktionen *whattype, type , typematch.* Hierbei liefert *whattype* den von MAPLE bevorzugten Typ-Namen (einen String):

```
whattype(a);                       ⟶ string
whattype(a+b);                     ⟶ '+'
whattype({a,b});                   ⟶ set
whattype([a]);                     ⟶ list
whattype(1/2);                     ⟶ fraction
whattype(1.72);                    ⟶ float
whattype(taylor(exp(x),x=0,2));    ⟶ series
whattype(f(x));                    ⟶ function
whattype(x->x∧2);                  ⟶ procedure
```

Der Aufruf *type(expr,Typname)* liefert *true/false*:

```
type(a+b,string);                      ⟶ false
type(f(x),function);                   ⟶ true
type(taylor(exp(x),x=0,2),taylor);     ⟶ true
```

Mehrere der etwa 100 Typ-Namen (siehe *help(type)*) können passen:

```
type(a+b,'+');          ⟶ true
type(a+b,polynom);      ⟶ true
type(a+b,algebraic);    ⟶ true
```

Der Typ *numeric* ist die Zusammenfassung der Typen *integer, fraction, float*. Der Typ *algebraic* ist die Zusammenfassung von *numeric, string, '+', '*', '∧', function, series, ...*. Man kann solche Zusammenfassungen selbst in der Form *type(expr,{Typ1,Typ2,...})* realisieren, so ist z.B.

```
type(expr,{integer,fraction,float});
```

äquivalent zu `type(expr,numeric);`

Es gibt "Oberflächentypen" (surface types), die nur von der Wurzel des Darstellungsbaums eines Ausdrucks bestimmt werden. Die "vernesteten Typen" (structured types) werden durch den gesamten Baum bestimmt. Beispiel: der Typ *constant* repräsentiert die komplexen Zahlen (plus ein paar weitere Ausdrücke).

```
type(1+sqrt(2)*I,constant);   ⟶ true
type(exp(2*Pi),constant);     ⟶ true
type(exp(x),constant);        ⟶ false
type(infinity,constant);      ⟶ true
type(false,constant);         ⟶ true
```

Leider gibt es Inkonsequenzen:

```
type(int(exp(x∧4),x=0..1),constant); ⟶ false
```

Man kann selbst leicht "vernestete Typentests" realisieren:

```
type(x=5.0, string=float);                  ⟶ true
type(x=y, string=integer);                  ⟶ false
type({1,2,3}, set(integer));                ⟶ true
type([x=1,y={a,b}], list(name=anything));   ⟶ true
```

Beispiel: der System-Integrierer *int* nimmt als 2.tes Argument entweder einen Namen (→ Stammfunktion) oder eine Gleichung wie z.B. *x=0..1* (→ bestimmtes Integral). Die entsprechende Implementierung startet im Prinzip so:

```
int:=proc(expr,var);
  if not type(var,{string,string=range}) then ERROR(...)
  ...
end;
```

Zu den Möglichkeiten siehe *help(type[structured])*.

3.5 Indizierte Typen: *table, array, matrix, vector*

Es existiert der "indizierte Typ" *table* mit dem Spezialfall *array* und den Spezialfällen *matrix, vector* von *array*. Die allgemeine Struktur ist *table*: dies sind Objekte, die beliebige MAPLE-Ausdrücke als Indizes zulassen:

Beispiel:

```
T[Dienstag]:=Uebung:
T[Donnerstag]:=Vorlesung:
T[1+sin(x)]:=2+sin(x);
print(T);               ⟶ table([
                             1 + sin(x) = 2 + sin(x)
                             Donnerstag = Vorlesung
                             Dienstag = Uebung
                             ])
T[1+sin(x)];            ⟶ 2 + cos(x)
T[sin(x)+cos(0)];       ⟶ 2 + cos(x)
T[2+sin(x)];            ⟶ T_{2+sin(x)}
```

Tables werden entweder explizit mit der *table*-Funktion erzeugt:

```
Ableitungen:=table([exp=exp,sin=cos,cos=-sin,ln=1/id]);
```

`Ableitungen[ln];` $\longrightarrow \frac{1}{id}$

`Ableitungen[BesselJ];` $\longrightarrow Ableitungen_{BesselJ}$

oder implizit:

```
T[index]:=expr;
```

trägt einen neuen Wert in eine existierende Table `T` ein (oder überschreibt einen alten Wert). Wenn `T` nicht vorher als Table existierte, so wird eine Table-Struktur mit diesem Eintrag erzeugt ("implizite Table-Erzeugung"):

`T:=evaln(T); T[1]:=x;`

`print(T);` $\longrightarrow$ *table*([
$1 = x$
])

`T[2]:=y; print(T);` $\longrightarrow$ *table*([
$1 = x$
$2 = y$
])

$\boxed{\text{Table-Operationen sind im MAPLE-Kern sehr schnell implementiert.}}$

(*table* ist eine sehr effiziente, flexible, empfehlenswerte Datenstruktur!)

Beispiel: Sei *Liste:=[a,b,a,sin(x),cos(x),sin(x)]*. Bestimme, welche unterschiedlichen Einträge in dieser Liste existieren und wie oft sie auftauchen:

`Zaehler:=table(sparse);` $\longrightarrow$ *table(sparse,[ ]);* *(Initialisierung)*
`Zaehler[x+y];` $\longrightarrow$ *0;* *(wegen der sparse-Initialisierung)*

```
for i from 1 to nops(Liste) do
  Zaehler[op(i,Liste)]:=Zaehler[op(i,Liste)]+1:
od:
```

`print(Zaehler);` $\longrightarrow$ *table(sparse,[*
sin(x)=2
cos(x)=1
a=2
b=1
])

`op(2,eval(Zaehler));` $\longrightarrow [sin(x) = 2, cos(x) = 1, a = 2, b = 1]$
`indices(Zaehler);` $\longrightarrow$ *[sin(x)],[cos(x)],[a],[b]*
`entries(Zaehler);` $\longrightarrow$ *[2],[1],[2],[1]*

Achtung: Tables werden in MAPLE nicht unmittelbar evaluiert!! Der Aufruf `T;` einer Table `T` liefert nicht den Tabelleninhalt, sondern nur den Namen `T` ! Benutze `print(T);` zur Ansicht des Inhalts oder auch `eval(T);` (evaluiere) oder auch `copy(T);` (evaluierte Kopie). Vergleiche mit dem Kapitel "Evaluierung".

Der Typ *array* bezeichnet spezielle Tables, in denen nur Indizes der Form *integer* oder $integer_1, integer_2, \ldots$ zugelassen sind (sie entsprechen damit dem *array*-Typ anderer Sprachen). Bei jedem Aufruf wird die Zulässigkeit der Indizes überprüft. Dieser Datentyp muß mittels der Funktion *array* erzeugt werden:

```
M:= array(-1..1,2..3,sparse);  ⟶ M:=array(sparse,
                                      -1 .. 2 , 2 .. 3 , [ ]);
type(M,array);                 ⟶ true
type(M,table);                 ⟶ true
M[0,3]:=x:
M[2,3]:=x:                     ⟶ Error
M:=array(1..n,1..2);           ⟶ Error
```

Die Typen *vector* und *matrix* werden mit *array(1..n)* bzw. *array(1..n,1..m)* (mit konkreten Werten für n, m) erzeugt. Ihre Besonderheiten sind, daß *print* sie als Matrix bzw. (Zeilen-)Vektor ausgibt und Matrixmultiplikation mit &* und *evalm* ("evaluate as a matrix") möglich ist:

```
M:=array(1..2,1..3):
N:=array(1..3,1..2):
v:=array(1..2):
type(v,vector);         ⟶ true
type(v,array);          ⟶ true
type(v,table);          ⟶ true
```

`print(M);` $\longrightarrow \begin{bmatrix} M_{1,1} & M_{1,2} \\ M_{2,1} & M_{2,2} \end{bmatrix}$

```
evalm( 2* M &* N &* v ):
evalm( N &* M &* v ):   ⟶ Error
```

Alternativ zur Erzeugung mittels *array* können Matrizen und Vektoren auch mittels *matrix* bzw. *vector* erzeugt werden, wobei eine Abbildung zur Initialisierung der Komponenten übergeben werden kann:

`M:=matrix(2,3,(i,j)->(i+j-1)∧(-1));` $\longrightarrow M := \begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} \end{bmatrix}$

`v:=vector(2,i->i∧2);` $\longrightarrow v := [1, 4]$

Achtung: Vektoren sehen aus wie Listen, sind aber keine!

Achtung: Vektoren vom Typ *vector* werden zwar von *print* als Zeilen ausgegeben, werden von *evalm* aber kontextabhängig als Zeilen oder als Spalten angesehen:

```
M:=array(1..2,1..2):
v:=array(1..2):
evalm(M &* v);        (liefert Vektor = Matrix &* Spalte )
evalm(v &* M);        (liefert Vektor = Zeile &* Matrix )
```

Achtung: Mit `v:=evaln(v);v[1]:=1;v[2]:=2;` wird nicht der Tabellen-Subtyp *vector* erzeugt, sondern der allgemeine *table*-Typ.

Achtung: Funktionen wie *det* (Determinante) oder *inverse* für quadratische Matrizen werden nicht automatisch geladen, sondern müssen über das Paket *linalg* (Lineare Algebra) angefordert werden:

`A:=array(1..2,1..2); det(A);` $\longrightarrow$ *det(A)*
`with(linalg):`
`det(A);` $\longrightarrow A_{1,1}A_{2,2} - A_{1,2}A_{2,1}$

4 Evaluierung

Beim Aufruf von Variablen bzw. beim Bilden von Ausdrücken wird <u>in der Regel</u> "vollständig evaluiert", d.h. für alle Operanden wird rekursiv ihr momentaner Wert eingesetzt:

`a:=1;` $\longrightarrow$ *a:=1*
`A:=a+b+c+f(a);` $\longrightarrow$ *A:=1+b+c+f(1)*
`a:=2;` $\longrightarrow$ *a:=2*
`A;` $\longrightarrow$ *1+b+c+f(1)*

Die Evaluierungstiefe kann mit `eval(expr,Tiefe);` gesteuert werden:

```
a:=b1: b1:=b2: b2:=b3:
a;                     ⟶ b3   (vollständige Evaluierung)
eval(a,1);             ⟶ b1
eval(a,2);             ⟶ b2
eval(a,3);             ⟶ b3
eval(a);               ⟶ b3   (vollständige Evaluierung)
```

Verwandt sind die "Verzögerungszeichen":

```
a:=b:
'a';          ⟶ a
c:='''a+b'''; ⟶ c:="a+b"
d:=c;         ⟶ d:= 'a+b'
e:=d;         ⟶ e:=  a+b
f:=e;         ⟶ f:=  2 b
```

Beispiel:

```
f:= x -> if x>=0 then 1 else -1 fi:
f(x);                          ⟶ Error
plot(f(x),x=-1..1);            ⟶ Error
plot('f(x)',x=-1..1);          ⟶ (Graphik)
```

Achtung: Es gibt Ausnahmen vom Konzept der vollständigen Evaluierung! Variable, denen ein Wert mit indiziertem Typ (*table, array, matrix, vector*) oder eine Abbildung (Typ *procedure*) zugewiesen wurden, werden aus Effizienzgründen bei einem einfachen Aufruf nicht evaluiert, sondern es wird nur ihr Name angefordert ("call by name").

Beispiel:

```
T[1]:=x;     ⟶ T1:=x      (implizite Table-Erzeugung)
T;           ⟶ T
print(T);    ⟶ table([1=x])
V:=T;        ⟶ V:=T       (call by name)
print(V);    ⟶ table([1=x])
T:=sin(x);   ⟶ T:=sin(x)
V;           ⟶ sin(x)
```

Mit *eval* wird Evaluierung erzwungen:

`T[1]:=x;`	$\longrightarrow$	T_1:=x
`eval(T);`	$\longrightarrow$	*table([1=x])*
`V:=eval(T);`	$\longrightarrow$	*V:=table([1=x])*
`T:=sin(x);`	$\longrightarrow$	*T:=sin(x)*
`print(V);`	$\longrightarrow$	*table([1=x])*
`cos`	$\longrightarrow$	*cos*
`eval(cos);`	$\longrightarrow$	*proc(x) ... end*

Alternativ kann mit `Kopie:=copy(Original);` evaluiert und kopiert werden.

5 Input und Output

5.1 Input aus Dateien

Alternativ zur interaktiven Eingabe kann aus Dateien gelesen werden:

`read Dateiname;`	(Eingabe aus Textdateien)
`read 'Dateiname.m';`	(Eingabe aus Dateien in MAPLEs Zwischencode)

`for i from 1 to 10 do`
`  read Datei.i;`
`od;`
	(die Statements in den Dateien *Datei1,..,Datei10* werden eingelesen und ausgeführt)

Die Funktion *readdata* erlaubt <u>formatierte</u> Eingabe von numerischen Daten aus einer Datei.

5.2 Interaktiver Output

Erinnerung:

`statement ;`		Evaluierung und Ausgabe
`statement :`		Evaluierung, aber keine Ausgabe
`for i from 1 to 10 do`		
`  i∧2;`		die Zwischenergebnis der Schleife
`od :`	$\longleftarrow$	werden <u>nicht</u> angezeigt.

Die globale Variable *printlevel* (Default-Wert 1) bestimmt, bis zu welchem "Level" Ausgabe erfolgt. Die interaktive Ebene der Sitzung ist Level 0. Eintritt in eine Schleife oder in ein if-Statement erhöht den Level jeweils um 1, Eintritt in eine Prozedur um 5.

`printlevel:= -1;`	außer *print*-Befehlen wird nichts angezeigt
`printlevel:=1000;`	praktisch alle Zwischenergebnisse (z.B. bei vernesteten Funktionsaufrufen) werden angezeigt.

Es gibt zur Anzeige *print* (der "pretty-Printer", 2-dimensionale Ausgabe) und *lprint* (der "line-Printer", dem Eingabeformat entsprechende 1-dimensionale Ausgabe):

`lprint(2*sin(x));`	$\longrightarrow$ `2*sin(x)`
`lprint(diff(f(x,y),x,y));`	$\longrightarrow$ `diff(diff(f(x,y),x),y)`
`print(ifactor(6));`	$\longrightarrow$ *(2) (3)*
`lprint(ifactor(6));`	$\longrightarrow$ ``` ``(2)*``(3) ```

Weiterhin existiert *printf* zur formatierten Ausgabe nach C-Syntax.

Die Funktion *interface* erlaubt die Einstellung von Eingabe/Ausgabe-Optionen:

`interface(verboseproc=3);`	der MAPLE-Quell-Code von Systemfunktionen kann nun mit `eval(Name);` eingesehen werden
`eval(cos);`	*proc(x) local n,t; option ...* *...(Implementierung des cosinus)...* *end*

Nach `infolevel[Name]:= 1;` erzeugt die System-Funktion *Name* bei Problemen Warnungen:

`solve(0=1,x);`	$\longrightarrow$
`infolevel[solve]:=1:`	
`solve(0=1,x);`	$\longrightarrow$ *Warning: no solutions found*

5.3 Output in Dateien

Die Ausgabe kann in Text-Dateien umgelenkt werden:

`writeto(Dateiname);`	die Ausgabe der folgenden Statements wird in die Datei *Dateiname* geschrieben
`appendto(Dateiname);`	die Ausgabe der folgenden Statements wird an die Datei *Dateiname* angehängt

```
for i from 1 to 10 do
    writeto(Datei.i);
    statements;
od;
```

`writeto(terminal);`	Rückkehr zur Sitzung

Zugewiesene Variable können mit

`save a,b,c, Dateiname;`	speichere *a,b,c*
`save Dateiname;`	speichere die gesamte Sitzung

in eine Textdatei *Dateiname* geschrieben werden. Diese kann mit `read Dateiname;` unmittelbar in eine neue Sitzung eingelesen werden, wodurch die gespeicherten Variablen ihre Werte wieder annehmen. Mit

```
save RiesenAusdruck, `Name.m`;
```

wird die Datei *Name.m* in einem Zwischen-Code erzeugt, der schneller wiedereinlesbar ist. Dann werden die gemachten Definitionen beim Wiedereinlesen aber nicht angezeigt. Mit `anames();` ("assigned names") erhält man die Folge aller zugewiesenen Variablennamen.

5.4 Speichern von Graphiken in Dateien

Mit *interface* (oder auch *plotsetup*) kann die graphische Ausgabe von *plot* und *plot3d* in Dateien umgelenkt werden und dort in gängigen Graphikformaten (*postscript, jpeg, pcx, ..*) gespeichert werden:

```
interface(plotdevice=postscript,
          plotoptions=`portrait,noborder`);

for i to 10 do
  interface(plotoutput=Bild.i.`.ps`);
  plot(sin(x),x=0..1,color=black);
od;
```

`interface(plotdevice=default);` (Rückkehr ins Worksheet)

erzeugt die druckfähigen Text-Dateien *Bild1.ps,...,Bild10.ps* mit Postscript-Text.

Achtung: Die Plots sollten mit der Option *color=black* aufgerufen werden, sonst werden Farben bei Schwarz-Weiß-Druck sehr unbefriedigend umgesetzt.

6 Schleifen

Die allgemeine Form ist

```
for i from Start by Inkrement to Ende while Bedingung do
  statements;
  if weitereBedingung then break fi;
  statements;
od;
```

Hierbei müssen *Start, Inkrement, Ende* vom Typ *numeric* (also *integer, fraction* oder *float*) sein, *Bedingung* muss zu *true/false* evaluierbar sein. Das *break*-Statement unterbricht die Schleife.

Beispiel:

```
i:=x+y:

for i from 2/10 by 2.7 to evalf(Pi) while i∧2<100 do
  i;           ⟶ 1/5
od;            ⟶ 2.900000000

i;             ⟶ 5.600000000
```

Alle Teile sind optional:

weglassen	entspricht
`from Start`	*Start=1*
`by Inkrement`	*Inkrement=1*
`to Ende`	*Ende*$=+\infty$ ($-\infty$ bei negativem Inkrement)
`while Bedingung`	*Bedingung = true*

Beispiele:

```
for i do
  i;
od;      ⟶ 1
         ⟶ 2
           (Endlosschleife)
```

```
x:=2;
do
  x:=2*x;
  if x>=70 then break; fi;
od;
```

oder auch

```
while Bedingung do
  statetments;
od;
```

Die Schleifenvariable kann innerhalb der Schleife verändert werden:

```
for i to 10 do  ⟶ i:=1
  i:=i∧2;       ⟶ i:=4
od;             ⟶ i:=25
```

Start, Inkrement, Ende sind mit dem Aufruf der Schleife fixiert:

```
n:=2:
for i to n do
  n:=10;       ⟶ n:=10
od;            ⟶ n:=10
```

Eine andere (wichtige) Schleifenform ist

```
for x in expr do
  statements;
od;
```

Dies ist eine Abkürzung für

```
for i to nops(expr) do
  x:= op(i,expr);
  statements;
od;
```

bzw.

```
for i to nops([expr]) do
  x:= op(i,[expr]);
  statements;
od;
```

wenn *expr* eine Folge ist.

Beispiele:

```
for x in f(a+b,c,d) do   ⟶ a+b
  x;                     ⟶ c
od;                      ⟶ d
```

```
for x in Folge do ... od;
for x in Liste do ... od;
for x in Menge do ... od;
```

7 If-Statements

Die allgemeine Form ist

```
if Bedingung
  then statements;
  else statements;   (optional)
fi;
```

oder auch (*elif* $\widehat{=}$ *else if*)

```
if     Bedingung then statements;     ⟵ "1.ter Spezialfall"
  elif Bedingung then statements;     ⟵ "2.ter Spezialfall"
  elif ...
  ...
  else statements;                    ⟵ "Standardfall"
fi;
```

Dies ist die MAPLE-Version eines *case*-Statements.

Anmerkung: mit *and* erzeugte Bedingungen werden nacheinander nur so weit wie nötig überprüft:

```
if x<>0 and x∧(-2)>10 then ...
```

erzeugt kein Problem für $x = 0$, da dann $x^{-2} > 10$ nicht mehr überprüft wird.

8 Prozeduren

Die beste und ausführlichste Referenz zu MAPLE-Prozeduren ist der *MAPLE V Programming Guide* [Mon 96].

8.1 Syntax

8.1.1 Einfache Funktionen: ->

Die einfachste Form einer Prozedur-Definition ist

(var$_1$*,var*$_2$*, ...)* `->` *expr;*

In dieser Form ist nur ein einziges Statement zur Bildung des Funktionswertes *expr* erlaubt. Beispiele:

```
x -> x∧2:
sqr:= ";              ⟶ sqr:= x -> x^2
sqr(a);               ⟶ a^2
sqr;                  ⟶ sqr        (call by name !!!)
eval(sqr);            ⟶ x -> x^2
whattype(sqr);        ⟶ procedure

Betrag:= x -> if x>=0 then x else -x fi:
Betrag(-2);           ⟶ 2
```

8.1.2 Komplexere Prozeduren: *proc() .. end*

Die allgemeine Form einer Prozedurdefinition ist

proc*(var*$_1$*,var*$_2$*, ...)*	(kein ; zwischen **proc( )** und **local** !!)
local *loc*$_1$*,loc*$_2$*, ... ;*	(lokale Variablen, optional)
global *glob*$_1$*,glob*$_2$*, ... ;*	(globale Variablen, optional)
options *opt*$_1$*,opt*$_2$*, ... ;*	(Optionen, optional)
description *Text ;*	(String zur Dokumentation, optional)
statements;	
end;	

Beispiele:

```
sqr:= proc(x); x∧2; end;
sqr(y);                       ⟶ y^2
sqr(a+1);                     ⟶ (a+1)^2

f:= proc(x,n)
  local i;
    sum(x∧i,i=0..n);
end;
f(y,3);                       ⟶ 1 + y + y^2 + y^3
```

Prozeduren sind Funktionen, sie liefern <u>stets</u> einen Wert zurück, der jedoch $NULL$ (die leere Folge) sein kann. Der Wert ist entweder der letzte in der Prozedur ausgewertete Ausdruck oder ein durch die $RETURN$-Funktion zurückgelieferter Wert, wobei ein $RETURN$-Aufruf die Prozedur beendet:

```
myevalf:=proc(a);
  if type(a,numeric) then evalf(a); else a; fi;
end;
```

ist identisch mit

```
proc(a); if type(a,numeric) then RETURN(evalf(a)); fi; a; end;
```

aber nicht identisch mit

```
proc(a); if type(a,numeric) then evalf(a); fi; a; end;
```

`myevalf(x+y);` $\longrightarrow x+y$
`myevalf(1/3);` $\longrightarrow 0.333...$
`myevalf(x+1/3);` $\longrightarrow x+\frac{1}{3}$

8.2 Manipulation von Eingangsparametern

Eingangsparameter, wenn sie als Variable übergeben wurden, können von der Prozedur verändert werden:

```
f:=proc(x,y); x:=9; y:=2*x; y+2; end;
```

`x:=evaln(x); y:=evaln(y);`
`z:= f(x,y);` $\longrightarrow z := 20$
`x,y;` $\longrightarrow 9, 18$

Aber:

`f(1,abc);` $\longrightarrow$ *Error*

`g:=proc(x) x[1]:=3; end;` (implizite *table*-Erzeugung oder neuer *table*-Eintrag)
`x:=evaln(x); g(x): print(x);` $\longrightarrow$ *table([1=3])*
`x[2]:=5: g(x): print(x);` $\longrightarrow$ *table([1=3,2=5])*

Zusammenfassung: Damit existieren 3 Mechanismen, um in einer Prozedur berechnete Werte zurückzuliefern:

- Rückgabe als Funktionswert der Prozedur,
- Manipulation globaler Variabler,
- Rückgabe durch Eingabeparameter.

8.3 Lokal oder global?

Werden Variable benutzt, die nicht als *local* oder *global* deklariert wurden, so gilt:

- Variable, denen Werte zugewiesen werden, werden automatisch (mit ausgegebenen Warnungen) als lokal angenommen, ebenso Laufvariablen in Schleifen, *sum, seq* etc. Eine Ausnahme ist die Variable *Digits*, die stets als global angenommen wird.

- Variable, auf die nur zugegriffen wird, werden automatisch (ohne ausgegebene Warnungen) als global angenommen. Damit können z.B. die Namen externer Prozeduren verwendet werden, ohne diese als *global* deklarieren zu müssen.

Achtung: Die globale Variable *Digits* wird unverändert von einer Prozedur zurückgegeben, selbst wenn sie explizit als *global* deklariert wird.

Achtung: Lokale Variable können eine Prozedur verlassen !! Beispiel:

```
f:=proc(x) local t; x*t end:
y:=f(2);
z:=f(3);
t:=1:
z-y;
```

$\longrightarrow y := 2t$
$\longrightarrow z := 3t$
$\longrightarrow z := 3t - 2t$

In der Ausgabe wird als Referenz auf die lokale Variable der Name benutzt, mit dem die lokale Variable deklariert war. Das Beispiel zeigt, daß jedoch jeder Aufruf zu einer anderen lokalen Variablen führt, deren Namen zwar alle gleich aussehen, die intern aber unterschieden werden. Auch bei globaler Benutzung des Namens werden die globalen Variablen nicht mit den lokalen identifiziert. Ein weiteres Beispiel:

`f:=proc(x) local t; t[x]:=1; t end:` (implizite Table-Erzeugung)
`y:=f(2);` $\longrightarrow$ *y:=t*
`eval(y);` $\longrightarrow$ *table([2=1])*
`z:=f(3);` $\longrightarrow$ *z:=t*
`eval(z);` $\longrightarrow$ *table([3=1])*
`eval(y);` $\longrightarrow$ *table([2=1])*

Will man die Referenz der zurückgelieferten Tabelle auf den lokalen Variablennamen vermeiden, so geht dies durch Rückgabe der evaluierten lokalen Variablen:

```
f:=proc(x) local t; t[x]:=1; eval(t) end:
```

Beide Version sind letztlich identisch und unterscheiden sich nur in der Art und Weise, wie die Namen der globalen Tabellen ausgegeben werden: im ersten Fall werden y, z durch (unbekannte interne, gleichaussehende aber vom Kern unterschiedene) Namen t ersetzt. Bei Rückgabe mittels *eval* entsteht kein Effizienzverlust, da *eval* nicht alle Tabelleneinträge sammelt und zurückgibt, sondern lediglich einen internen Zeiger auf die existierende Tabellenstruktur umsetzt. Bei Rückgabe mittels *copy* ist dies anders: alle Tabelleneinträge werden gesammelt und zu einer neuen Tabelle zusammengefaßt, was bei großen Tabellen zeitaufwendig ist.

8.4 Prozeduren mit beliebig vielen Argumenten: *args, nargs*

Prozeduren können ohne Angabe von Argumenten definiert werden:

```
f:=proc() print(‘hallo !‘); end;
f;                                  ⟶ f    (call by name !!!)
f();                                ⟶ hallo !
f(a,b,c);                           ⟶ hallo !
```

Man kann (auch bei der Prozedur-Deklaration ohne Angabe von Argumenten) auf die wirklichen Argumente beim Aufruf mittels

args	= die Folge der Argumente
args[i]	= das i.te Argument
nargs	= *nops([args])* = die Anzahl der Argumente

zugreifen:

```
first_and_last:=proc();
  if nargs<>0 then args[1],args[nargs]; else NULL; fi;
end;

f:=first_and_last:
f(x,10,a+b,sin(a/b));    ⟶ x, sin(a/b)
f(a);                    ⟶ a, a
f( );                    ⟶
```

8.5 Der Prozedurname: *procname*

Der Name der Prozedur kann innerhalb der Prozedur verwendet werden, ohne ihn zu kennen: die spezielle System-Variable *procname* ist der Platzhalter. Damit kann eine Prozedur sich rekursiv selbst aufrufen, auch wenn sie noch keinen Namen trägt:

```
proc() procname; end; f:=''; g:="";
f(x,y,z); ⟶ f
g( );     ⟶ g
```

8.6 Unevaluierte Rückgabe

Prozeduren können sich selbst (oder andere Funktionsaufrufe) unevaluiert zurückgeben:

```
Evalf:=proc(x)
  if type(x,constant) then evalf(x); else 'Evalf(x)'; fi;
end;
```

```
Evalf(1/3);     ⟶ .333333
Evalf(x+1/3);   ⟶ Evalf(x+1/3)
```

Eleganter ist

```
Evalf:=proc(x)
  if type(x,constant) then evalf(x); else 'procname(args)'; fi;
end;
```

Falls die Funktion unevaluiert mit anderen Argumente als *args* zurückgegeben werden sollen, so kann dies mit

RETURN('procname'(NeueArgumente));

geschehen, wobei die Verzögerung des Prozedurnamens verhindert, daß die Prozedur erneut durchlaufen wird.

8.7 Fehlermeldungen: *ERROR*

Die Funktion *ERROR* (gedacht für Fehlermeldungen) druckt ihre Argumente und bricht die Prozedur mit dem Wert *NULL* ab, z.B.,

```
mybinomial:=proc(n,m);
  if nargs<>2 then
    ERROR(`ich erwarte 2 Argumente und habe `,
        nargs,` Argumente erhalten`);
  fi;
  n!/m!/(n-m)!;
end;
```

8.8 Typen-Deklaration

Besser wäre es, die Argumente n, m im obigen *mybinomial* auf ihren mathematisch sinnvollen Typ *integer* bzw. *nonnegint* (nicht-negative ganze Zahl) zu überprüfen. MAPLE erlaubt dazu als Abkürzung den Operator :: ("Typ-Deklaration") in einer Prozedur-Definition:

```
mybinomial:=proc(n::nonnegint,m::nonnegint);
  if nargs<>2 then
    ERROR(`ich erwarte 2 Argumente und habe `,
        nargs,` Argumente erhalten`);
  fi;
  n!/m!/(n-m)!;
end;
```

`mybinomial(3,-1);` $\longrightarrow$ *Error*

Anmerkung: deklariert man einen Eingabeparameter als *uneval*, so wird er unevaluiert an die Prozedur übergeben:

```
QuadriereIndex:=proc(x::uneval);
  cat( op(1,x), op(2,x)^2 );
end:
```

```
i:=3: a3:=x:
QuadriereIndex(a.i);
```
$\longrightarrow$ *a9*

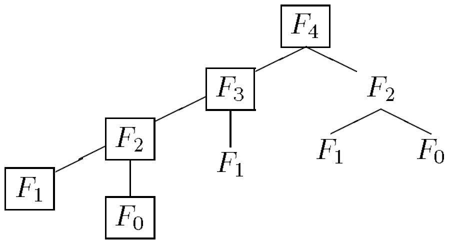
8.9 Option *remember*

Unter den Optionen, mit denen Prozeduren ausgestattet werden können, sind *option trace* (ein Hilfsmittel zum Debuggen, siehe Kapitel 8.14) und *option remember* interessant.

Nach Aufruf einer mit *option remember* deklarierten Prozedur wird das Ergebnis in einer *remember table* gespeichert. Bei erneutem Aufruf mit denselben Argumenten wird das Ergebnis direkt aus der *remember table* entnommen, ohne daß die Prozedur erneut durchlaufen wird.

Beispiel: die Folge der Fibonacci-Zahlen ist rekursiv definiert durch

$$F_n = F_{n-1} + F_{n-2} \ , \quad F_0 = 1, F_1 = 1 \ .$$



Rekursive Aufrufe

Ohne *option remember*:
    fib1(n) wird $\approx 1.45 \times (1.618..)^n - 1$ mal durchlaufen,
mit *option remember*:
    fib2(n) wird $n + 1$ mal durchlaufen.

```
fib1:=proc(n::nonnegint) global zaehler1;
  zaehler1:=zaehler1+1;
  if n=0 or n=1 then 1 else fib1(n-1)+fib1(n-2); fi;
end;

fib2:=proc(n::nonnegint) global zaehler2;
option remember;
  zaehler2:=zaehler2+1;
  if n=0 or n=1 then 1 else fib2(n-1)+fib2(n-2); fi;
end;
```

`zaehler1:=0; time(fib1(25));`	$\longrightarrow$ *29.230*	(Sekunden)
`zaehler1;`	$\longrightarrow$ *242785*	(Durchläufe)
`zaehler2:=0; time(fib2(25));`	$\longrightarrow$ *0.000*	(Sekunden)
`zaehler2;`	$\longrightarrow$ *26*	(Durchläufe)

Anmerkung: Der Interpreter verarbeitet rekursive Prozeduren (mindestens) genauso schnell, wie eine äquivalente Schleife. Der Aufruf *fib2(n)* ist nicht langsamer als die iterative Berechnung

```
F[0]:=0; F[1]:=0:
for i from 2 to n do F[n]:=F[n-1]+F[n-2] od:
ergebnis:=F[n];
```

Allerdings braucht die Berechnung mittels rekursiver Prozeduren erheblich mehr Speicherplatz, was sich dann auch dramatisch auf die Rechenzeit auswirkt, sobald der Hauptspeicher des Rechners aufgefüllt ist.

Achtung: Der *remember*-Mechanismus erkennt nur schon früher verarbeitete Eingangsparameter, aber nicht die Werte der verwendeten globalen Variablen! Nach Änderung globaler Werte werden i.a. erinnerte falsche Prozedurwerte zurückgeliefert!

Achtung: Funktionen (=Prozeduren), die sich oft unevaluiert zurückgeben, sollten in der Regel mit *option remember* implementiert werden, damit die unevaluierten Ausdrücke weiterverarbeitet werden können, ohne jedesmal die Prozedur erneut durchlaufen zu müssen.

Mit

`readlib(forget); forget(Name);`

wird die *remember table* der Prozedur *Name* gelöscht. Mit

`Name(Argument):=Ausdruck;`

wird ein zusätzlicher Eintrag in die *remember table* von *Name* erzeugt (bzw. eine *remember table* angelegt, wenn nicht *option remember* in der Deklaration von *Name* benutzt worden war).

Beispiel:

`f:= x -> sin(x)/x: f(0);` $\longrightarrow$ *Error*

Mit `f(0):=1;` wird die Definitionslücke geschlossen.

Achtung:

`f:=evaln(f);`
`f(x):= x∧2;` (dies erzeugt <u>nicht</u> die Abbildung x-> x^2)
`eval(f);` $\longrightarrow$ *proc( ) option remember; 'procname(args)' end;*
`f(x);` $\longrightarrow$ x^2
`f(y);` $\longrightarrow$ *f(y)*

8.10 Die Operanden eine Prozedur

Ein zu einer Prozedur evaluierbarer Name wird durch einen einfachen Aufruf nur als Name aufgerufen (call by name):

`f:=proc(x) option remember; x∧2; end;`

`f;` $\longrightarrow$ *f*
`eval(f);` $\longrightarrow$ *proc(x) option remember;* x^2 *end;*
`f(2);` $\longrightarrow$ *4*

Die Prozedur *eval(f)* (nicht ihr Name *f*) hat 6 Operanden:

`op(0,f)`	$\longrightarrow$ *string*	
`op(0,eval(f))`	$\longrightarrow$ *procedure*	(der Typenname)
`op(1,eval(f))`	$\longrightarrow$ x	(Folge der Argumente)
`op(2,eval(f))`	$\longrightarrow$	(Folge der lokalen Variablen)
`op(3,eval(f))`	$\longrightarrow$ *remember*	(Folge der Optionen)
`op(4,eval(f))`	$\longrightarrow$ *table([2=4])*	(die *remember table*)
`op(6,eval(f))`	$\longrightarrow$	(die globalen Variablen)

Diese Operanden können dynamisch verändert werden. Das folgende Beispiel ersetzt in einer Prozedur die *remember table* durch *NULL*, womit diese gelöscht wird:

```
Loesche_Remember_Table:=proc(f)
  if type(f,procedure) then f:=subsop(4=NULL,eval(f)); fi;
  NULL;
end;
```

```
f(x):=1:
Loesche_Remember_Table(f);
f(x);
```
$\longrightarrow$ *f(x)*

Anmerkung: einfacher geht das natürlich mit der Systemfunktion *forget*.

Anmerkung: Nicht nur die Operanden von *eval(f)*, auch der Prozedurkörper ist zugänglich und dynamisch manipulierbar, siehe

```
help(procbody);  help(procmake);
```

8.11 Operatorschreibweise

Der Arithmetikoperator + ist eigentlich eine Funktion, deren Name der String '+' ist:

'+'(a,b); $\longrightarrow$ $a+b$

Als Operator ist die Eingabe $a+b$ erlaubt. Diese Form der Ein- und Ausgabe kann auch für eigene Funktionen (=Prozeduren) erzeugt werden: Funktionen, deren Name mit dem Symbol & beginnen, werden als "Operatoren" angesehen. Interessant ist der Fall binärer Operatoren (eine Funktion mit 2 Argumenten). Der Aufruf des folgenden trivialen Operators $\&m$ liefert sich unevaluiert zurück, erlaubt dabei aber die elegante Ein- und Ausgabe:

```
`&m`:=proc(x,y); 'x &m y'; end;
```

`&m(x,y);` $\longrightarrow x \mathbin{\&m} y$
`x &m y;` $\longrightarrow x \mathbin{\&m} y$
`y &m x;` $\longrightarrow y \mathbin{\&m} x$

Dies ist ein Hilfsmittel zur Definition eines eigenen (eventuell nicht-kommutativen) Produktes mit angenehmer Ein- und Ausgabe. Eine Komposition

`x &m y &m z;` $\longrightarrow x \mathbin{\&m} y \mathbin{\&m} z$

ist äquivalent zu `&m(&m(x,y),z)`, womit

$$x \mathbin{\&m} y \mathbin{\&m} z \;=\; (x \mathbin{\&m} y) \mathbin{\&m} z$$

gilt:

`x &m y &m z - (x &m y) &m z;` $\longrightarrow 0$

Achtung: in der Ausgabe von "Mehrfachprodukten" werden Klammerungen fortgelassen, so daß

`(x &m y) &m z;` $\longrightarrow x \mathbin{\&m} y \mathbin{\&m} z$
`x &m (y &m z);` $\longrightarrow x \mathbin{\&m} y \mathbin{\&m} z$

entsteht. Intern stellen sich diese Ausdrücke jedoch als

$$\begin{aligned} x \mathbin{\&m} y \mathbin{\&m} z &= (x \mathbin{\&m} y) \mathbin{\&m} z &= \&m(\&m(x,y),z) \\ &\neq x \mathbin{\&m} (y \mathbin{\&m} z) &= \&m(x,\&m(y,z)) \end{aligned}$$

dar. In der Tat ergibt sich

`(x &m y) &m z - x &m (y &m z);`
$\longrightarrow (x \mathbin{\&m} y \mathbin{\&m} z) - (x \mathbin{\&m} y \mathbin{\&m} z) \quad (\neq 0)$

Soll das "Produkt" $\&m$ assoziativ sein (wie es die Ausgabe suggeriert), so muss dies in der Definition von $\&m$ explizit implementiert werden[1]. Dies kann leicht dadurch geschehen, daß in einem vernesteten Aufruf $\&m(\&m(x,y),z)$ alle inneren "Multiplikationen" systematisch auf den rechten (alternativ: den linken) Eingang verschoben werden:

[1] MAPLEs *define*-Möglichkeit für Operatoren scheint mir nicht sehr brauchbar.

```
`&m`:=proc(x,y)
  if op(0,x)=`&m` then RETURN(&m(op(1,x),&m(op(2,x),y)));fi;
  'x &m y';
end;
```

`(x &m y ) &m z - x &m (y &m z);` $\longrightarrow 0$

Die folgende Multiplikation $\&m$ genügt einem Distributivgesetz

$$(a+b)\ \&m\ (c+d) \ = \ (a\ \&m\ c) + (a\ \&m\ d) + (b\ \&m\ c) + (b\ \&m\ d)\ ,$$

welches mit Hilfe der Systemfunktionen *map, map2* implementiert wird, die unmittelbar

`map(f,x1+x2+...,y);` $\longrightarrow f(x1,y)+f(x2,y)+...$
`map2(f,x,y1+y2+...);` $\longrightarrow f(x,y1)+f(x,y2)+...$

liefern:

```
`&m`:=proc(x,y);
  if type(x,`+`) then RETURN(map(`&m`,x,y)); fi;
  if type(y,`+`) then RETURN(map2(`&m`,x,y)); fi;
  'x &m y';
end;
```

`(a+2*b) &m (c+d);`
$\longrightarrow$ *(a &m c) + (a &m d) + (2 b &m c) + (2 b &m d)*

Achtung: Die Operatoren $\&*$ und $\&+$ werden von der Systemfunktion *evalm* ("evaluate as a matrix") als Matrixaddition bzw. -multiplikation verstanden und verarbeitet. Hierbei funktioniert auch der Aufruf `evalm(a+b);` als Matrixaddition, während `evalm(Skalar*Matrix);` als Multiplikation einer Matrix mit einem skalaren Parameter ausgewertet wird.

8.12 Konvertiere Ausdrücke in Funktionen: *unapply*

Sei *expr* ein Ausdruck, der Namen $x_1, x_2, ..$ enthält. Mit

`f:=unapply(expr,`x_1`,`x_2`,...);` $\longrightarrow$ f:= (x_1,x_2,...) `->` expr

wird eine Funktion erzeugt. Beispiel:

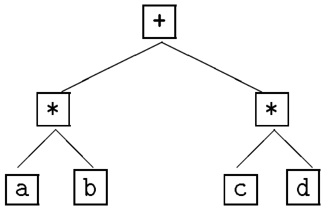
`f:=unapply( x∧2+y∧2+z∧2 ,x,y);` $\longrightarrow f := (x,y)$`->` $x^2+y^2+z^2$
`f(3,4);` $\longrightarrow 25+z^2$

8.13 Einige einfache Beispiele

8.13.1 Beispiel: *Kanten*

Aufgabe: implementiere eine Funktion *Kanten :=proc(expr) .. end*, welche die Anzahl der Kanten im Darstellungsbaum von *expr* zählt, z.B.

`a*b+c*d;`



$\longrightarrow$ 6 Kanten

Erste Lösung: (gebe das Ergebnis an einen globalen *Zaehler*)

```
Kanten:=proc(expr) local term; global Zaehler;
  Zaehler:=Zaehler+nops(expr);
  for term in expr do
    if not type(term,{string,integer}) then Kanten(term); fi;
  od;
  NULL;
end;
```

Achtung: hier darf *option remember* nicht verwendet werden, da *Zaehler* nur beim wirklichen Durchlaufen der Prozedur hochgezählt wird.

`Zaehler:=0: Kanten(a*b+c*d): Zaehler;` $\longrightarrow$ *6*

Zweite Lösung: (gebe das Ergebnis als Wert zurück)

```
Kanten:=proc(expr) local i; option remember;
  if type(expr,{string,integer}) then RETURN(0) fi;
  nops(expr)+sum('Kanten(op(i,expr))',i=1..nops(expr));
end;
```

Achtung: ohne die verzögerte Evaluierung *'Kanten(op(i,expr))'* in *sum* würde *sum* versuchen, dieses Argument als symbolischen Ausdruck von *i* zu berechnen und dabei in eine Endlos-Schleife geraten.

`Kanten(a*b+c*d);` $\longrightarrow$ *6*

8.13.2 Beispiel: *diffOrdnung*

Aufgabe: Sei *expr* ein Ausdruck der Form

$$\left(\frac{d^k}{dx^k}\ u(x)\right)^m$$

mit einer unbestimmten Funktion $u(x)$. In MAPLE:

```
diff(..(diff(u(x),x),..,x)∧m.
```

Implementiere eine Funktion *diffOrdnung:=proc(expr) .. end*, welche die Anzahl k der Ableitungen zurückliefert:

```
diffOrdnung:= proc(expr) option remember;
  if type(expr,'∧') then RETURN(diffOrdnung(op(1,expr)));fi;
  if type(expr,function) then
    if op(0,expr)=diff
      then RETURN(1+diffOrdnung(op(1,expr)));
      else RETURN(0);
    fi;
  fi;
  ERROR('der Ausdruck ist nicht von der erwarteten Form');
end;
```

8.13.3 Beispiel: *Max*

Aufgabe: Implementiere eine Funktion *Max:=proc() ... end*, welche das Maximum seiner (beliebig vielen) Argumente zurückliefert. Hierbei soll das Maximum derjenigen Eingänge gebildet werden, die vom Typ *numeric* sind. Das Maximum anderer Ausdrücke soll unevaluiert zurückgegeben werden. Dabei soll die Regel

$$Max(a, Max(b, c)) = Max(a, b, c)$$

implementiert werden.

```
Max:=proc() local Eingaenge,num,sym,numMax,i;
  if nargs=1 then RETURN(args[1]); fi;                          #0)
  Eingaenge:=subs(Max=proc() args end,{args});                  #1)
  num:= map(x->if type(x,numeric) then x else NULL fi,''); #2)
  sym:= '''' minus '';                                          #3)
  if nops(num)>0 then                                           #4)
    numMax:=num[1];                                             #4)
    for i from 2 to nops(num) do                                #4)
        if op(i,num)>numMax then numMax:=op(i,num);fi;          #4)
    od;                                                         #4)
  fi;                                                           #4)
  if   sym={} then RETURN(numMax)                               #5)
  elif num={} then RETURN('Max'(op(Eingaenge)));                #5)
  else             RETURN('Max'(numMax,op(sym)));               #5)
  fi;                                                           #5)
end;
```

```
Max(2,3,4);                                  ⟶ 4
Max(x);                                      ⟶ x
Max(x,y);                                    ⟶ Max(x,y)
Max(2,17/8,Max(15.2,x),Max(x,y,16/8),15.2);  ⟶ Max(15.2,x,y)
```

Erläuterung der angegebenen Implementierung: Bei nur einem Argument ist dieses das Ergebnis (#0)). In #1) werden die inneren Maxima eliminiert, indem *Max* durch eine Prozedur ersetzt wird, welche nur die Folge ihrer Argumente zurückliefert, z.B.,

$$Max(x, Max(x, y), Max(z, 2)) \longrightarrow \; args = x, Max(x, y), Max(z, 2)$$

$$\longrightarrow \; Eingaenge := \{x, x, y, z, 2\} \quad (= \{x, y, z, 2\}) \; .$$

In #2) wird eine Prozedur auf die Elemente von *Eingaenge* angewendet, die alle nicht-numerischen Einträge auf *NULL* setzt. Dies ergibt die Menge der numerischen Argumente *num*. In #3) wird die Menge der symbolischen Eingänge *sym* gebildet, die durch Abziehen der numerischen Eingänge *num* von allen Eingängen *Eingaenge* entsteht. In #4) wird das Maximum der numerischen Eingänge berechnet. In #5) wird eine sinnvolle Ausgabeform erzeugt.

8.13.4 Beispiel: Differentiation

Aufgabe: Es soll ein symbolischer Differenzierer implementiert werden, der die Ableitung algebraischer Ausdrücke bestimmt, die in beliebiger

Weise durch Addition, Multiplikation, Potenzieren und Anwendungen einiger mathematischer Funktion (*exp, ln, sin, arcsin, ...*) aus beliebigen Namen erzeugt wurden, wobei nach einem dieser Namen differenziert werden soll. Der Prozedurname sei *Diff*. Da dieser Name den inerten System-Differenzierer bezeichnet und dementsprechend geschützt ist, muß er zunächst durch `unprotect(Diff);` freigegeben werden[2].

Die Eigenschaften, welche die Differentiation algebraisch auf der Menge solcher Ausdrücke festlegen, sind:

`#1)` $\dfrac{df}{dx} = 0$, wenn f nicht von x abhängt,

`#2)` $\dfrac{dx}{dx} = 1$,

`#3)` $\dfrac{d\,(f+g)}{dx} = \dfrac{df}{dx} + \dfrac{dg}{dx}$ (Linearität),

`#4)` $\dfrac{d\,ab}{dx} = \dfrac{da}{dx}\,b + a\,\dfrac{db}{dx}$ (Produktregel),

`#5)` $\dfrac{d\,a^b}{dx} = a^{b-1}\,\dfrac{da}{dx} + a^b\,ln(a)\,\dfrac{db}{dx}$ (aus $a^b = exp(b\,ln(a))$),

`#6)` $\dfrac{d}{dx}\,F(y(x)) = F'(y)\,\dfrac{dy}{dx}$ (Kettenregel).

Hierbei sind in `#6)` für einige Funktionen F die Ableitungen bekannt.

`#7)` Bei nicht spezifizierten Funktionen F soll *Diff* unevaluiert zurückgegeben werden.

Diese Eigenschaften sind in dieser Reihenfolge in der Prozedur *Diff* (Tafel 1) implementiert. Dabei überprüft in `#1)` die MAPLE-Funktion *depends*, ob der zu differenzierende Ausdruck von x abhängt. Die Linearität `#3)` ist mittels der MAPLE-Funktion *map* implementiert, die unmittelbar

```
map(Funktion, a₁+a₂+··· , x )
        ⟶   Funktion(a₁,x)+Funktion(a₂,x)+···
```

liefert.

[2] I.a. sollte man geschützte Namen nicht überschreiben, da so das System leicht unbrauchbar gemacht werden kann. Die einzige Rolle des inerten System-*Diff* ist aber, vom Pretty-Printer erkannt und schön dargestellt zu werden, so daß in diesem Fall keine Probleme entstehen.

```
unprotect(Diff);                                              #0)

Diff:=proc(f,x)
local a,b,F,y;
 if not depends(f,x) then RETURN(0);                     fi; #1)
 if f=x              then RETURN(1);                     fi; #2)
 if type(f,`+`)      then RETURN(map(Diff,f,x));         fi; #3)
 if type(f,`*`)      then a:=op(1,f);b:=f/a;
                       RETURN(Diff(a,x)*b+a*Diff(b,x));fi; #4)
 if type(f,`**`) then a:=op(1,f);b:=op(2,f);
        RETURN(f*ln(a)*Diff(b,x)+a**(b-1)*b*Diff(a,x));fi; #5)
 if type(f,function) then
   if nops(f)>1 then ERROR(`Differentiation von Funktionen
     mit mehreren Eingaengen ist nicht implementiert`);fi;
   F:=op(0,f);y:=op(1,f);                                     #6)
   if   F=exp    then RETURN( exp(y)        *Diff(y,x));
   elif F=ln     then RETURN( 1/y           *Diff(y,x));
   elif F=sin    then RETURN( cos(y)        *Diff(y,x));
   elif F=arcsin then RETURN( (1-y^2)^(-1/2)*Diff(y,x));
   elif F=cos    then RETURN(-sin(y)        *Diff(y,x));
   elif F=arccos then RETURN(-(1-y^2)^(-1/2)*Diff(y,x));
   elif F=tan    then RETURN( (1+tan(y)^2)  *Diff(y,x));
   elif F=arctan then RETURN(  1/(1+y^2)    *Diff(y,x));
   fi;
 fi;
 'Diff(f,x)';                                                 #7)
end:
```

Tafel 1: Ein symbolischer Differenzierer

Offensichtlich kann in der angegebenen Implementierung die Menge der Funktionen F, die differenziert werden kann, leicht erweitert werden. Der Einbau von Funktionen F mit mehreren Eingängen ist leicht realisierbar usw. Speziell kann das Differenzieren von Ausdrücken der Form *int(..)* eingebaut werden.
Die Effizienz des (im Kern implementierten) Systemdifferenzierers *diff* ist mit eigenen Implementierungen natürlich nicht zu erreichen.

8.13.5 Beispiel: Symbolische Integration von Differentialausdrücken

Aufgabe: Sei

$$F(u_0, \ldots, u_{k+1}) = \frac{d}{dx}\, G(u_0, \ldots, u_k) \ , \quad u_i = \frac{d^i}{dx^i}\, u(x) \ .$$

Finde $G(u_0, \ldots, u_k)$ bei gegebenem $F = F(u_0, \ldots, u_{k+1})$, d.h., implementiere einen Integrierer $Int(F, x)$, der $G(u_0, \ldots, u_k)$ liefert.

Herleitung des Algorithmus:

Fakt 1:

$$\frac{d}{dx}\, G(u_0, \ldots, u_k) = \underbrace{\frac{\partial G}{\partial u_0}\, u_1 + \cdots + \frac{\partial G}{\partial u_{k-1}}\, u_k}_{h(u_0,\ldots,u_k)} \ + \ \underbrace{\frac{\partial G}{\partial u_k}}_{f(u_0,\ldots,u_k)} \ u_{k+1}$$

Fakt 2 (Folgerung): $F(u_0, \ldots, u_{k+1})$ kann nur dann $\frac{d}{dx} G(u_0, \ldots, u_k)$ sein, wenn es von der Form

$$F(u_0, \ldots, u_{k+1}) = h(u_0, \ldots, u_k) + f(u_0, \ldots, u_k)\ u_{k+1}$$

ist (der höchste Ableitungsterm muß linear auftauchen).

Fakt 3: Zu $F(u_0, \ldots, u_{k+1}) = h(u_0, \ldots, u_k) + f(u_0, \ldots, u_k)\ u_{k+1}$ finde durch Integration nach u_k eine Funktion $g(u_0, \ldots, u_k)$ mit

$$\frac{\partial}{\partial u_k} g(u_0, \ldots, u_k) = f(u_0, \ldots, u_k) \ .$$

Dann hängt $c(u_0, \ldots, u_k) := G(u_0, \ldots, u_k) - g(u_0, \ldots, u_k)$ gar nicht von u_k ab:

$$\frac{\partial c}{\partial u_k} = \frac{\partial G}{\partial u_k} - \frac{\partial g}{\partial u_k} = f(u_0, \ldots, u_k) - f(u_0, \ldots, u_k) = 0 \ .$$

Interpretation: Das gesuchte G ist durch g bis auf

$$G(u_0, \ldots, u_k) = g(u_0, \ldots, u_k) + c(u_0, \ldots, u_{k-1})$$

bestimmt.

Fakt 4: $F - \frac{dg}{dx}$ hängt nicht mehr von u_{k+1} ab:

$$F - \frac{dg}{dx} = \frac{d}{dx}(G-g) = \frac{d}{dx}\, c(u_0, \ldots, u_{k-1}) = \frac{\partial c}{\partial u_0}\, u_1 + \cdots + \frac{\partial c}{\partial u_{k-1}}\, u_k \ .$$

Hieraus ergibt sich der folgende rekursive Algorithmus:

1) Finde die Zerlegung

$$F(u_0, \ldots, u_{k+1}) = h(u_0, \ldots, u_k) + f(u_0, \ldots, u_k)\ u_{k+1} \ .$$

2) Bestimme $g(u_0, \ldots, u_k)$ durch Integration von $f(u_0, \ldots, u_k)$ nach u_k:

$$\frac{\partial}{\partial u_k} g(u_0, \ldots, u_k) = f(u_0, \ldots, u_k) \ .$$

3) Rekursiver Aufruf:

$$Int(F, x) = g + Int(F - \frac{dg}{dx}, x) \ .$$

Der verbleibende Integrand $F^{(neu)} = F - \frac{dg}{dx}$ hängt nur noch von $u_0, \ldots, u_k$ ab und besitzt wieder eine Stammfunktion:

$$F - \frac{dg}{dx} = \frac{d}{dx}(G-g) \ .$$

Ende: Sei

$$F(u_0, u_1) = h(u_0) + f(u_0) u_1 = \frac{d}{dx} G(u_0) = \frac{\partial G}{\partial u_0}\, u_1 \ ,$$

d.h., notwendigerweise gilt $h(u_0) = 0$. In Schritt 2) wird g mit $\frac{\partial g}{\partial u_0} = f(u_0) = \frac{\partial G}{\partial u_0}$ gefunden, d.h., $g(u_0) = G(u_0) + const$. Der letzte rekursive Aufruf

$$Int(F, x) = g + Int(F - \frac{dg}{dx}, x) = g + Int(0, x)$$

führt zum erfolgreichen Abbruch des Algorithmus.

Spezialfall: Sei F ein Polynom:

$$F(u_0, \ldots, u_{k+1}) = h(u_0, \ldots, u_k) + A(u_0, \ldots, u_{k-1})\; u_k^p\, u_{k+1}\ .$$

Dann ist $g(u_0, \ldots, u_k) = \frac{1}{p+1}\, A u_k^{p+1}$ und der rekursive Aufruf 3) lautet:

$$\begin{aligned} Int(F, x) &= \frac{A u_k^{p+1}}{p+1} + Int(F - \frac{A_x u_k^{p+1}}{p+1} - A u_k^p u_{k+1}, x) \\ &= \frac{A u_k^{p+1}}{p+1} + Int(h(u_0, \ldots, u_k) - \frac{A_x u_k^{p+1}}{p+1}\ , x)\ , \end{aligned}$$

wobei diese Rekursion nach Ausnutzung der Linearität von Int monomweise (d.h., mit $h(u_0, \ldots, u_k) = 0$) implementiert werden kann. Vergleiche mit der Anleitung zur Aufgabe 4 im Worksheet *session4.mws*.

Achtung: für beliebige (nicht-polynomiale) Integranden F:

- Man darf die Linearität von Int nicht implementieren, da sonst in 2) Schwierigkeiten beim Suchen der (darstellbaren) Stammfunktion $g = \int f du_k$ auftreten können, falls die einzelnen Summanden einer Summe nicht getrennt darstellbare Stammfunktionen besitzen. Bei polynomialem F ist dies kein Problem.
- Es kann passieren, daß $\frac{dg}{dx} - F$ nicht automatisch so weit vereinfacht wird, daß die Abhängigkeit von u_k herausfällt, z.B.

$$F - \frac{dg}{dx} = \tilde{c}(u_0, \ldots, u_{k-1}) + \underbrace{\frac{2\, u_k}{1 - u_{k-1}^2}}_{\text{aus } F} \quad \underbrace{- \frac{u_k}{1 - u_{k-1}} - \frac{u_k}{1 + u_{k-1}}}_{\text{aus } \frac{dg}{dx}}$$

Implementierung (Grundidee: Björn Brill):

```
# Die folgende Prozedur durchsucht einen beliebigen Ausdruck
# nach Teilausdruecken der Form diff(u(x),x$k) und bestimmt
# die Anzahl der Ableitungen k:

diffOrdnung:=proc(expr,x::name) local i; option remember;
  if type(expr,{string,integer}) then RETURN(0);fi;
  if op(0,expr)=diff and op(2,expr)=x
     then RETURN(1+procname(op(1,expr),x));
  fi;
  RETURN(max(seq(diffOrdnung(op(i,expr),x),i=1..nops(expr))));
end:

# Die folgende Prozedur sucht in einem beliebigen Ausdruck den
# Teilausdruck diff(u(x),x$k), der den Wert diffOrdnung(expr,x)
# erzeugt:

hoechsterTerm:=proc(expr,x::name) local term; option remember;
  for term in expr do
    if diffOrdnung(term,x)=diffOrdnung(expr,x)
       then RETURN(hoechsterTerm(term,x));
    fi;
  od;
  if op(0,expr)=diff and op(2,expr)=x then RETURN(expr) fi;
  ERROR(`should not arrive here`);
end:

# Der Name Int ist geschuetzt und muss freigegeben werden:

unprotect(Int):
```

```
# Die folgende Integration findet Stammfunktionen eines
# Differentialausdrucks F(u0,...,u.k,u.(k+1)) in einer
# Funktion u0=u(x) in einer Variablen x :

Int:=proc(expr,x::name) local F,Hoechster,lambda,g;
   option remember;
   F:=expand(expr):
   if F=0 then RETURN(0); fi;
   if diffOrdnung(F,x)=0 then RETURN('procname(args)');fi;
   Hoechster:=hoechsterTerm(F,x);
# Sei Hoechster=u.(k+1).  Zerlege
#   F = h(u0,..,u.k)+f(u0,..,u.k)*u.(k+1) +R(u0,..,u.k,u.(k+1))
# wobei der Rest R(u0,...,u.(k+1)) nichtlinear in u.(k+1) ist.
# Dieser Teil hat definitiv keine Stammfunktion und wird
# abgespalten ("Teillinearitaet" von Int):
   if not type(F,linear(Hoechster)) then
     if type(F,`+`)                          # F=h+f*u.(k+1)+R
        then remove(type,",linear(Hoechster));# h+R
             remove(has,F,Hoechster);         # h
             RETURN('procname'(""-",x)+procname(F-""+",x));
        else RETURN('procname'(F,x));
     fi;
   fi;
# Nun gilt F = h(u0,..,u.k) + f(u0,..,u.k)*u.(k+1).
# Bestimme f(u0,..,u.k) und durch Integration g(u0,..,u.k)
# mit diff(g,u.k)=f:
   coeff(F,Hoechster,1);           # dies ist f(u0,..,u.k)
   subs(op(1,Hoechster)=lambda,"); # u.k=op(1,Hoechster)
   g:=int(",lambda);               # Integration nach u.k
# Wenn der Systemintegrierer keine Stammfunktion findet:
   if has(g,int) then RETURN('procname(args)');fi;
   g:=simplify(subs(lambda=op(1,Hoechster),g));
# Nach Konstruktion gilt
#    diff(g,x) = diff(g,u0)*u1 +...+ diff(g,u.k)*u.(k+1)
# sodass F-diff(g,x) gar nicht mehr Hoechster=u.(k+1) enthaelt.
# Allerdings werden nicht alle Ausdruecke automatisch so weit
# vereinfacht, dass diese Abhaengigkeit herausfaellt. Setze
# daher einfach Hoechster=0 ein, um simplify zu unterstuetzen:
   subs(Hoechster=0,F-diff(g,x));
   simplify(");
# Offensichtlich gilt Int(F,x) = g + Int( F-diff(g,x) ,x),
# wobei der verbleibende Integrand F-diff(g,x) eine niedrigere
# diffOrdnung als der urspruengliche Integrand F hat.
   g+procname(",x);
end:
```

Beispiel:

```
alias(u0=u(x),seq(u.i=diff(u(x),x$i),i=1..10):

u3/(1-u5^2)+ln(u3)/(1+u4^3)*exp(cos(u2)^2+1/(1+u6)*sin(u5^2)^2);
F:=expand(diff(",x));
Int(F+u4/(1+u3^2),x);
                      2          2                 2 2
              cos(u2)  + cos(u2)  u6 + 1 - cos(u5 )
   ln(u3) exp(-------------------------------------)
                             1 + u6                         u3
   ---------------------------------------------------- - -------
                             3                             2
                       1 + u4                            u5  - 1

   + arctan(u3)
```

8.14 Debugging

MAPLEs Hilfsmittel zum Debuggen:

a) Heraufsetzen von *printlevel*.

b) Mit `trace(Prozedurname);` oder `debug(Prozedurname);` wird nachträglich *option trace* in die Prozedur eingebaut. Ein Aufruf der Prozedur zeigt dann alle Zwischenergebnisse an.

c) Man kann "watchpoints" setzen (siehe *help(debugger)*).

d) Man kann nach `readlib(maplemint);` diverse Informationen wie z.B. nie benutzte lokale Variablen etc. mittels `maplemint(Prozedurname);` erhalten.

e) Man kann auf UNIX-Ebene mit `mint Dateiname` Informationen über die in der Datei *Dateiname* enthaltenen MAPLE-Befehle erhalten (Syntax-Fehler etc.).

8.15 Effizienztest: *time, profile*

Seien

```
fib1:=proc(n)
  if n=0 or n=1 then 1 else fib1(n-1)+fib1(n-2); fi;
end;

fib2:=proc(n) option remember;
  if n=0 or n=1 then 1 else fib2(n-1)+fib2(n-2); fi;
end;
```

Mit `time(fib1(10));` wird die verbrauchte Zeit zur Evaluation von *fib1(10)* (in Sekunden) angegeben. Die bislang in der Sitzung verbrauchte Zeit wird durch `time();` geliefert (siehe auch `help(showtime);`).

Mit der *profile*-Funktion erhält man Information über die Anzahl der Funktionsaufrufe, Rekursionstiefe, Laufzeiten etc.:

```
readlib(profile):
profile(fib1,fib2):
fib1(10):fib2(10):
showprofile();
```

```
function  depth  calls  time   time%   bytes   bytes%
------------------------------------------------------
fib1         10    177  .400   93.02  152072    90.01
fib2         10     19  .030    6.97   16864     9.98
------------------------------------------------------
total:       20    196  .430  100.00  168936   100.00
```

Achtung: die absoluten Laufzeiten täuschen, da das Profilieren Prozeduren verlangsamt. Die relativen Angaben sind aber interessant um festzustellen, in welchen Routinen die meiste Zeit verbraucht wird.

Achtung: Mittels *procname* implementierte rekursive Aufrufe werden von *profile* nicht erkannt, die *depth*-Werte sind falsch.

9 Einige Systemfunktionen

Eine Auflistung aller Bibliotheks-Funktionen ist mittels

```
help(index[function]);
```

zu erreichen. Viele weitere Funktionen (Prozeduren, Algorithmen) existieren in den Anwendungs-Paketen:

```
help(index[packages]);
```

Eine Liste der implementierten mathematischen Speziellen Funktionen erhält man mit

`help(inifcns);` ("initially known mathematical functions")

"Implementiert" heißt dabei, daß diese Funktionen von *diff, evalf, ...* verarbeitet werden.

9.1 Bibliotheksfunktionen (ein unvollständiger Überblick)

Zunächst eine unvollständige Auflistung einiger interessanter Systemfunktionen[3].

* `abs`	Betrag
* `add`,`sum`	Summengeneratoren
`AiryAi`	Airy-Funktionen
* `alias`	Definition von Abkürzungen
* `allvalues`	alle Werte eines *RootOf*-Ausdrucks
* `anames`	Auflistung aller gesetzten Variablen
* `appendto`, `writeto`,`writedata`	Ausgabe in Dateien
`arcsin,..`	Inverse trigonometrischer Funktionen
`arcsinh,..`	Inverse der Hyperbel-Funktionen
* `array`	erzeuge eine *array*-Variable
* `assign`	konvertiere Gleichungen in Zuweisungen
`asympt`	asymptotische Entwicklung, z.B. Stirling-Formel: `asympt(n!,n);` $\longrightarrow \left(\frac{n}{e}\right)^n \sqrt{2\pi\, n}\ (1+\frac{1}{2n}+\cdots)$
`bernoulli`	Bernoulli-Zahlen und -Polynome
* `BesselI`, `BesselJ,..`	Bessel-Funktionen
`Beta`	$B(x,y) = (\Gamma(x)+\Gamma(y))/\Gamma(x+y)$
`bspline`	B-Splines
`C`	erzeuge C-Code
* `cat`	String-Konkatenation
`chebyshev`	Entwicklung nach Chebyshev-Polynomen
`chrem`	Algorithmus des Chinesischen Restsatzes

[3] Mit * gekennzeichnete Funktionen tauchen im Skript oder in den Worksheets auf. Kernfunktionen sind unterstrichen.

Ci	$Ci(x) = \gamma + ln(x) + \int_0^x \frac{cos(t)-1}{t}\ dt$
* <u>coeff</u>	extrahiert die Koeffizienten eines Polynoms
* <u>coeffs</u>	die Koeffizienten multivariater Polynome
* collect	faßt Koeffizienten in Summen zusammen
* combine	faßt Terme zusammen, z.B. `combine(exp(a)*exp(b));` $\longrightarrow e^{a+b}$ `combine(2*sin(a)*sin(b));` $\longrightarrow cos(a-b) - cos(a+b)$

* <u>convert</u>	Konvertierer mit vielen Möglichkeiten, z.B.
	`convert(2/(x∧2-1),parfrac,x);` $\longrightarrow \frac{1}{x-1} - \frac{1}{x+1}$
	`convert(10,base,2);`$\longrightarrow [0,1,0,1]$
	`convert(Table,Array);`
	`convert(Reihe,Polynom);`

* copy	evaluierte Kopie von Tables, Arrays, ...
* D	Differentialoperator, z.B.: `D(sin);` $\longrightarrow$ *cos* `D(x->x∧2);` $\longrightarrow$ x `->` $2\,x$
* debug,trace	Debugging-Hilfsmittel
* depends,<u>has</u>	Test auf Abhängigkeit: `depends(expr,x);` $\longrightarrow$ *true/false*
* <u>denom</u>,<u>numer</u>	"denominator, numerator", Nenner und Zähler
<u>degree</u>	Polynomgrad
* det	im *linalg*-Paket: Determinante
* <u>diff</u>,Diff	Differentiation von Ausdrücken, *Diff* ist inert
<u>divide</u>	Polynomdivision
dsolve	Löser für gewöhnliche Differentialgleichungen, Optionen z.B., *explicit, Laplace, series, numeric, ...*
Ei	$Ei(x) = \int_1^\infty e^{-xt}/t\ dt$
* eigenvals	im *linalg*-Paket: Matrix-Eigenwerte
* eigenvects	im *linalg*-Paket: Eigenwerte- und Vektoren
Eigenvals	Eigenwerte einer numerischen Matrix, ist inert, wird aber von *evalf* verstanden: `evalf(Eigenvals(A));` $\longrightarrow$ *numerische Eigenwerte*
erf	Fehlerfunktion $erf(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\ dt$
* <u>eval</u>	erzwungene Evaluierung (z.B. von Table-Namen)
<u>evalb</u>	Boolscher Evaluierer, z.B.: `evalb(x=x);` $\longrightarrow$ *true*
* evalc	evaluiere als komplexe Zahl, z.B.: `evalc(exp(I));` $\longrightarrow cos(1) + I\,sin(1)$

* evalf	universeller Floating-Point-Evaluierer
* evalhf	Hardware-gestützter Float-Evaluierer (schneller als *evalf*, aber nur Hardware-Genauigkeit)
* evalm	Evaluiere als Matrix
* evaln	Evaluiere als Name
evalr,shake	Evaluiere mit Intervallarithmetik
* exp	exp-Funktion
* expand	Expansion, z.B. `expand(sin(2*x));` $\longrightarrow 2\,sin(x)\,cos(x)$
extrema	Extremwerte (unter Nebenbedingungen)
* factor, Factor	Faktorisierer *Factor* ist inert, wird von *mod* verarbeitet
* FFT,iFFT	Fast-Fourier-Transform und inverse FFT
fortran	Erzeuge Fortran-Code
fourier, invfourier	im *inttrans*-Paket: (inverse) Fourier-Transformation
* fsolve	numerischer Gleichungslöser
GAMMA	$\Gamma(x) = \int_0^\infty t^{x-1}e^{-t}dt$
gcd,igcd	"greatest common divisor", größter gemeinsamer Teiler von Polynomen bzw. Integern
* has,depends	Test auf Abhängigkeit: `has(u(x),x);` $\longrightarrow$ *true*
hastype	Test auf Teilausdrücke eines bestimmten Typs
* help	Aufruf des On-Line-Help-Systems
* ifactor	Primfaktorzerlegung
* iFFT,FFT	(inverse) Fast-Fourier-Transform
* Im,Re	Imaginär- und Realteil komplexer Ausdrücke
indets	"indeterminates": die Unbestimmten eines Ausdrucks
* int,Int	Integration, *Int* ist inert
* interface	Setzen von Interface(Ausgabe)-Optionen
interp	Interpolation
Inverse	inerte Matrix-Inverse, wird von *mod* ausgewertet
* inverse	im *linalg*-Paket: Matrix-Inverse
invlaplace, laplace	im *inttrans*-Paket: (inverse) Laplace-Transformation
invfourier, fourier	im *inttrans*-Paket: (inverse) Fourier-Transformation
* isprime	Test, ob das Argument eine Primzahl ist

	`laplace, invlaplace`	im *inttrans*-Paket: (inverse) Laplace-Transformation
	`latex`	Erzeuge LaTeX-Code
	`lexorder`	Test auf lexikographische Ordnung
*	`lhs,rhs`	"left (right) hand side" einer Gleichung
	`limit`	Grenzwertberechnung
	`ln,log,log10`	Logarithmen
*	`lprint`	"linear print", 1-dimensionale Ausgabe
*	`map,map2`	Anwendung einer Funktion auf die Operanden eines Ausdrucks: `map(f,a+b+c);` $\longrightarrow f(a)+f(b)+f(c)$
	`match`	Patternmatching
*	`matrix`	Deklaration einer Matrix
*	`matrixplot`	im *plots*-Paket: 3D-Graphik der Matrix-Komponenten
*	`max,min`	Maximum, Minimum
	`mellin`	im *inttrans*-Paket: Mellin-Transformation
	`member`	Ist ein Ausdruck Element einer Menge oder Liste?
*	`min,max`	Minimum, Maximum
*	`mod`	"modulo"-Arithmetik: `(6*x+5) mod 4;` $\longrightarrow 2\,x+1$
*	`mtaylor`	multivariate Taylor-Entwicklung
*	`mul,product`	Produktgenerator
*	`nops`	Anzahl der Operanden eines Ausdrucks
*	`normal`	Normalform rationaler Ausdrücke
	`Nullspace`	Kern einer Matrix über Z_p ("modulo p"), inert, wird von *mod* verarbeitet
*	`numer,denom`	"numerator, denominator", Zähler und Nenner
*	`op,nops`	die Operanden eines Ausdrucks
	`order`	Fehlerordnung einer Reihenentwicklung
*	`plot,plot3d`	Erzeuge Graphiken
*	`print`	Ausgabe
*	`product, Product,mul`	Produktgenerator, *Product* ist inert
*	`profile`	Effizienztest von Prozeduren

* `rand`	Erzeuger von Zufallszahlen-Generatoren
* <u>`Re,Im`</u>	Real- und Imaginärteil komplexer Ausdrücke
* <u>`read`</u>	Eingabe aus Dateien
* `readdata`	formatierte numerische Eingabe aus Dateien
* <u>`readlib`</u>	Laden von Bibliotheksroutinen
* `rhs,lhs`	"right (left) hand side" einer Gleichung
* `RootOf`	Inerte Wurzel einer Gleichung, wird z.B. von *allvalues, evalf, simplify,* .. verarbeitet
`roots`	Polynomwurzeln über einem Koeffizientenkörper
`round`	Rundung numerischer Werte
`rsolve`	Löser für rekursive Gleichungen
* <u>`seq`</u>	Folgengenerator
* <u>`series`</u>	Reihenentwickler (Taylor,Laurent,...), z.B.: `series(sqrt(sin(x)),x=0,4);` $\longrightarrow \sqrt{x} - \frac{1}{12}\,x^{5/2} + O(x^{7/2})$
`shake,evalr`	Intervallarithmetik
* `simplify`	universeller Vereinfacher
* `solve`	universeller Gleichungslöser
* <u>`sort`</u>	Sortierer
* <u>`subs`</u>	Substituierer
* <u>`subsop`</u>	Ersetzung von Operanden
<u>`substring`</u>	Extrahiere Teilstrings, z.B.: `substring('Datei7.m',1..6)` $\longrightarrow$ *Datei7*
* `sum,Sum,`<u>`add`</u>	Summengenerator, *Sum* ist inert
`Svd`	"singular value decomposition", Singulärwerte einer numerischen Matrix, inert, wird von *evalf* verarbeitet
* <u>`table`</u>	explizite Table-Erzeugung
* <u>`taylor`</u>	univariate Taylor-Entwicklung
`testeq`	Probabilistischer Test auf Gültigkeit, z.B. `testeq(sin(x)∧2+cos(x)∧2=1);` $\longrightarrow$ *true*
* <u>`time`</u>	Systemzeit
* `trace,debug`	Debugging-Hilfsmittel
<u>`trunc`</u>`, round,frac`	Gauß-Klammer, $frac(x) = x - trunc(x)$
* <u>`type`</u>	Typen-Test
* `unapply`	Konvertierung *expr(x)* zur Abbildung *x* `->` *expr(x)*

* `vector`	Deklaration eines Vektors
* `whattype`	Typ eines Ausdrucks
* `with`	Laden eines Pakets
* `writeto`, `writedata`, `appendto`	Ausgabe in Dateien
`Zeta`	Riemannsche ζ-Funktion
`zip`	Verschmelzung von Listen mittels einer Funktion, z.B.: `zip((x,y)->x+I*y,[a,b],[A,B]);` $\longrightarrow$ *[a+IA,b+IB]*
`ztrans`, `invztrans`	Z-Transformation und Inverse

9.2 Die Anwendungspakete

Viele weitere Systemfunktionen sind, nach mathematischen Bereichen geordnet, in Anwendungspaketen zusammengefaßt und können mittels `with(Paketname);` geladen werden. Auflistung der vorhandenen Pakete:

```
help(index[packages]);
```

Eine Übersicht über die Funktionen ist jeweils mittels `help(Paketname);` zu erhalten.

Paketname	Mathematischer Bereich
DEtools	Werkzeuge zur Behandlung von Differentialgleichungen
Domains	Werkzeuge zur Erzeugung von Algorithmen in speziellen Domains
GF	Galois-Theorie
GaussInt	Gauß-Zahlen
LREtools	Werkzeuge für lineare Rekursionen
combinat	Kombinatorik-Funktionen
combstruct	Kombinatorik-Strukturen
difforms	Differentialformen
finance	Finanzmathematik
genfunc	rationale erzeugende Funktionen
geometry	Euklidische Geometrie
grobner	Gröbner-Basen
group	Gruppentheorie
inttrans	Integraltransformationen
liesymm	Lie-Symmetrien von Differentialgleichungen
linalg	Lineare Algebra
logic	Boolsche Logik
networks	Graphen, Netzwerke
numapprox	Numerische Approximation
numtheory	Zahlentheorie
orthopoly	Orthogonale Polynome
padic	p-adische Zahlen
plots	Graphik
plottools	Graphik-Primitive
powseries	formale Potenzreihen
process	(Unix)-multi-processing
simplex	Lineare Optimierung
stats	Statistik
student	Elementares Kalkül
sumtools	Summen
tensor	Tensorkalkül, Allgemeine Relativitätstheorie
totorder	Ordnungen

10 Algorithmische Aspekte: Gröbner-Basen

Literatur:

[GCL 92] K. O. Geddes, S.R. Czapor, G. Labahn, *Algorithms for Computer Algebra*, Kluwer, Dordrecht, 1992.

[DST 88] J.H. Davenport, Y. Siret, E. Tournier, *Computer Algebra*, Academic Press, London, 1988.

Motivation: Betrachte lineares Gleichungssystem:

$$\begin{array}{ccccccc} c_{11}\,x_1 & + & \cdots & + & c_{1n}\,x_n & = & b_1 \\ \vdots & & \ddots & & \vdots & & \vdots \\ c_{n1}\,x_1 & + & \cdots & + & c_{nn}\,x_n & = & b_n \end{array}$$

Vereinfachung durch Gauß-Elimination (soweit möglich):

$$\begin{array}{ccccccc} r_{11}\,x_1 & + & \cdots & + & r_{1n}\,x_n & = & b'_1 \\ & & \ddots & & \vdots & & \vdots \\ & & & & r_{nn}\,x_n & = & b'_n \end{array}$$

Grundidee hierbei ist: erzeuge 1 Gleichung (die letzte) in möglichst wenigen Variablen (nur x_n), die vorletzte Gleichung definiert dann x_{n-1} in Termen von x_n, die drittletzte Gleichung definiert dann x_{n-2} in Termen von x_{n-1}, x_n, usw.

Ähnliche Eliminationsprozesse sind oft auch für nichtlineare (polynomiale) Gleichungen möglich, z.B.,

$$\left\{ \begin{array}{l} p_1(x,y) = x^2 - y^2 = 0 \ , \\ p_2(x,y) = x^2 - x + 2\,y^2 - y - 1 = 0 \ . \end{array} \right.$$

Ad hoc-Umformungen $(p_1, p_2) \to (p'_1, p'_2) \to (p''_1, p''_2)$:

$$\left\{ \begin{array}{l} p'_1 = p_1 = x^2 - y^2 = 0 \ , \\ p'_2 = p_2 - p_1 = -x + 3\,y^2 - y - 1 = 0 \ . \end{array} \right.$$

$$\left\{ \begin{array}{l} p''_1 = p'_2 = -x + 3\,y^2 - y - 1 = 0 \ , \\ p''_2 = p'_1 + (x + 3\,y^2 - y - 1)\,p'_2 = (3\,y^2 - 1)\,(3\,y + 1)\,(y - 1) = 0 \ . \end{array} \right.$$

Also: Lösung $x = 3\,y^2 - y - 1$ mit $y =$ Wurzel von $(3\,y^2 - 1)\,(3\,y+1)\,(y-1)$.

10.1 Reduktion von Polynomen modulo eines Ideals

Allgemeine Aufgabe: zum System $Q = \{q_1, \ldots, q_k\}$ multivariater Polynome $q_i = q_i(x_1, \ldots, x_n)$ find einen "einfacheren" Satz $G = \{g_1, \ldots, g_{k'}\}$ von Polynomen, deren Nullstellenmenge mit der von Q übereinstimmt. Algebraisch formuliert: die von Q und G erzeugten Polynomideale sollen übereinstimmen.

Definition: *Eine Teilmenge I eines kommutativen Rings R heißt* **Ideal**, *wenn*

- $p, \tilde{p} \in I \quad \Longrightarrow \quad p - \tilde{p} \in I$,
- $p \in I, r \in R \quad \Longrightarrow \quad rp \in I$.

Hier interessiert die Situation: sei F ein Körper (allgemeiner kann man einen Koeffizientenring betrachten), betrachte den Ring

$$R = F[x_1, \ldots, x_n] = \left\{ \sum_{i_1,..,i_n \in \mathbb{N}_0} c_{i_1,..,i_n}\, x_1^{i_1} \cdots x_n^{i_n} \, ; \; c_{i_1,..,i_n} \in F \right\}$$

der Polynome in den Unbestimmten $x_1, \ldots, x_n$ über dem Koeffizienten-Körper (Ring) F (mit 1-Element $1 = x_1^0 \cdots x_n^0$).

Bezeichnung: Sei $Q = \{q_1, \ldots, q_k\} \subset F[x_1, \ldots, x_n]$. Das hiervon erzeugte Ideal ist

$$\langle\, Q \,\rangle = \left\{ \sum_{i=1}^{k} p_i\, q_i \; , \; p_i \in F[x_1, \ldots, x_n] \; , \; q_i \in Q \right\} .$$

Das Erzeugendensystem Q wird auch **Basis** von $\langle\, Q \,\rangle$ genannt.

Aufgabe: entscheide für beliebiges $p \in F[x_1, \ldots, x_n]$, ob $p \in \langle\, Q \,\rangle$ gilt.

Definition: *Eine (zulässige)* **Termordnung** *ist eine totale Ordnung $<$ auf den Monomen ("Termen") $T = \{\, x_1^{i_1} \cdots x_n^{i_n} \, ; \; i_1, \ldots i_n \in \mathbb{N}_0 \,\}$ mit*

- $1 < t \quad \forall t \in T \setminus \{1\}$,
- $s < t \Longrightarrow su < tu \quad \forall s, t, u \in T$.

Beispiel: Pure (inverse) lexicographic ("plex"):

$$x_1^{i_1} \cdots x_n^{i_n} \; < \; x_1^{j_1} \cdots x_n^{j_n} \quad \Longleftrightarrow \quad \exists\, k \text{ mit } i_1 = j_1, \ldots, i_{k-1} = j_{k-1}, i_k < j_k \; .$$

Speziell folgt hierbei $x_n < \ldots < x_1$.

Beispiel: Total degree ("tdeg"):

$$x_1^{i_1} \cdots x_n^{i_n} \; < \; x_1^{j_1} \cdots x_n^{j_n} \quad \Longleftrightarrow \quad \left\{ \begin{array}{c} i_1 + \cdots + i_n < j_1 + \cdots + j_n \\[1ex] \text{oder } \{ \; i_1 + \cdots + i_n = j_1 + \cdots + j_n \\ \text{und} \\ \exists\, k \text{ mit } i_k > j_k, i_{k+1} = j_{k+1}, \ldots, i_n = j_n \\ \} \end{array} \right.$$

(d.h., Ordnung nach dem Polynomgrad mit "inverse lexicographic" bei gleichem Grad). Speziell folgt wieder $x_n < \ldots < x_1$.

Beispiel: $x_1 = x, x_2 = y$:

$$\begin{array}{ll} plex: & 1 < y < y^2 < y^3 < \ldots < x < xy < xy^2 < \ldots < x^2 < x^2y < x^2y^2 < \ldots \\ tdeg: & 1 < y < x < y^2 < xy < x^2 < y^3 < \ldots \end{array}$$

Sei von nun an eine Termordnung vorgegeben.

Bezeichnung: Das Monom höchster Ordnung in $p = \sum c_{i_1,..,i_n}\, x_1^{i_1} \cdots x_n^{i_n}$ sei durch die Potenzen $i_1 = I_1, \ldots, i_n = I_n$ gegeben. Setze:

$$\begin{array}{lcl} hmonom(p) & := & c_{I_1,..,I_n}\, x_1^{I_1} \cdots x_n^{I_n} \;, \\ hcoeff(p) & := & c_{I_1,..,I_n} \\ hterm(p) & := & x_1^{I_1} \cdots x_n^{I_n} \;. \end{array}$$

Konvention: $hmonom(0) = hcoeff(0) = 0$, $hterm(0) = 1$.

Beispiel: $p = 2\,xyz - xy + z + 1 \in \mathbb{Z}[x, y, z]$. Mit plex-Ordnung über x, y, z fallend angeordnet:

$$p = \underbrace{2}_{hcoeff} \underbrace{xyz}_{hterm} - xy + z + 1 \;.$$

Achtung: $p = 2\,xyz - xy + z + 1 \in F[x, y]$, $F = \mathbb{Z}[z]$. Mit plex-Ordnung über x, y fallend angeordnet:

$$p = \underbrace{(2\,z - 1)}_{hcoeff} \underbrace{xy}_{hterm} + (z + 1) \;.$$

Definition: *Seien* $p, q \in F[x_1, \ldots, x_n]$, $p, q \neq 0$. *Wenn in* p *ein Monom existiert, das durch* $hterm(q)$ *teilbar ist, dann läßt sich dieses Monom durch Subtraktion eines geeigneten Vielfachen von* q *eliminieren:*

$$p = \alpha\; hterm(q) + r \;, \quad \alpha, r \in F[x_1, \ldots, x_n], \alpha \neq 0 \;.$$

Setze

$$\tilde{p} = p - \frac{\alpha}{hcoeff(q)}\; q \;.$$

Dann heißt p *zu* $\tilde{p}$ **reduzierbar**, *Schreibweise:* $p \rightarrow_q \tilde{p}$.

Bemerkung: Offensichtlich muß für das eliminierbare Monom $\alpha\; hterm(q)$ gelten:

$$hterm(q) \leq hterm(\alpha)\; hterm(q) \leq hterm(p) \;.$$

Damit wird durch die Reduktion die Ordnung des Polynoms nicht erhöht:

$$hterm(\tilde{p}) \leq hterm(p) \;.$$

Definition: *Sei* $Q = \{q_1, \ldots, q_k\} \subset F[x_1, \ldots, x_n]$, *Wenn zu* $0 \neq p \in F[x_1, \ldots, x_n]$ *ein* $\tilde{p} \neq p$ *und ein* $0 \neq q \in Q$ *existiert mit* $p \rightarrow_q \tilde{p}$, *dann heißt* p <u>*modulo* G</u> *zu* $\tilde{p}$ **reduzierbar**, *Schreibweise:* $p \rightarrow_Q \tilde{p}$.
Wenn keine Reduktion $p \rightarrow_Q \tilde{p} \neq p$ *möglich ist (d.h., für kein* $0 \neq q \in Q$ *teilt* $hterm(q)$ *ein Monom in* p*), dann heißt* p **irreduzibel modulo** Q.
Konvention: 0 wird als irreduzibel angesehen.

Satz: *Es existieren nur endlich viele Reduktionsketten*

$$p \;\rightarrow_Q\; \tilde{p} \;\rightarrow_Q\; \tilde{\tilde{p}} \;\rightarrow_Q\; \ldots$$

bis zur Irreduzibilität modulo Q.

Bew: Theorem 10.1 (Seite 435) in [GCL 92].

Bezeichnung: Wenn $p \rightarrow_Q \tilde{p} \rightarrow_Q \ldots \rightarrow_Q \hat{p}$ mit irreduziblem $\hat{p}$, dann Schreibweise $p \rightarrow_Q^* \hat{p}$ oder auch $\hat{p} = Reduce(p, Q)$.

Achtung: Die reduzierte Form $\hat{p}$ zu p ist nicht eindeutig.

Mit dem letzten Satz terminiert der folgende Reduktionsalgorithmus. Sei

$$R_{p,Q} := \{q \in Q \setminus \{0\} \;;\; hterm(q) \text{ teilt } hterm(p)\} \;.$$

```
Reduce:=proc(p,Q)      # finde p̂ mit p →*_Q p̂
  r := p;  p̂ := 0;
  while r ≠ 0 do
    while R_{r,Q} ≠ { } do
      wähle q ∈ R_{r,Q};
      r := r - (hmonom(r)/hmonom(q)) q ;      # Elimination der höchsten
                                              # Ordnung in r
    od;
    # nun ist hmonom(r) irreduzibel modulo Q. Subtrahiere
    # hmonom(r), um niedrigere Ordnungen in r zu reduzieren:
    p̂ := p̂ + hmonom(r);  r := r - hmonom(r);
  od;
  RETURN(p̂)
end;
```

Beispiel: Ordnung plex:

$$Q := \{\underbrace{x^3yz - xz^2}_{q_1}, \underbrace{xy^2z - xyz}_{q_2}, \underbrace{x^2y^2 - z^2}_{q_3}\} \subset \mathbb{Z}[x,y,z] \,.$$

Reduktion von $p_1 = x^2y^2z - z^3$:

$$p_1 \to_{q_3} \hat{p}_1 = p_1 - z\,q_3 = 0 \,.$$

Reduktion[4] von $p_2 = -x^2y^2z + x^2yz$:

$$p_2 \to_{q_2} \hat{p}_2 = p_2 + x\,q_2 = 0 \,.$$

Beobachtung: $p_1 + p_2 = x^2yz - z^3$ ist irreduzibel modulo Q.

10.2 Gröbner-Basen

Offensichtlich gilt

$$p \in F[x_1,\ldots,x_n] \to_Q^* 0 \qquad \Longrightarrow \qquad p \in \langle\, Q \,\rangle \,.$$

Die Umkehrung gilt jedoch nicht: im letzten Beispiel galt $p_1, p_2 \in \langle\, Q \,\rangle$, aber $p_1 + p_2 \in \langle\, Q \,\rangle$ nicht reduzibel modulo Q.

[4] Eine andere Reduktion von p_2: $p_2 \to_{q_3} \tilde{p}_2 = p_2 + z\,q_3 = x^2yz - z^3$ mit irreduziblem $\tilde{p}_2$.

Definition: *$G = \{g_1, \ldots, g_k\} \subset F[x_1, \ldots, x_n]$ heißt* **Gröbner-Basis** *(des Ideals $\langle G \rangle$), wenn*

$$p \in F[x_1, \ldots, x_n] \rightarrow_G^* 0 \qquad \Longleftrightarrow \qquad p \in \langle G \rangle \, .$$

Das Entscheidungsproblem $p \overset{?}{\in} \langle G \rangle$ wird damit für Gröbner-Basen durch den Reduktionsalgorithmus *Reduce* beantwortet. Das algorithmische Problem $p \overset{?}{\in} \langle Q \rangle$ für beliebiges Q wird dahin verschoben, zu Q eine Gröbner-Basis G mit $\langle Q \rangle = \langle G \rangle$ zu berechnen. Es gilt

Satz: (Hironaka 1964,Buchberger 1970) *Jedes Ideal $\langle Q \rangle \subset F[x_1, \ldots, x_n]$ besitzt eine Gröbner-Basis G mit $\langle Q \rangle = \langle G \rangle$.*

Das entscheidende Hilfsmittel zur Konstruktion von G für ein gegebenes Q ist der Begriff des S-Polynoms:

Definition: *Das S-Polynom für $p, q \in F[x_1, \ldots, x_n] \setminus \{0\}$ ist*

$$S(q,p) = KGV(hmonom(p), hmonom(q)) \left(\frac{p}{hmonom(p)} - \frac{q}{hmonom(q)} \right)$$

mit KGV = kleinstes gemeinsames Vielfaches, also

$$KGV(\, x_1^{i_1} \cdots x_n^{i_n} \, , \, x_1^{j_1} \cdots x_n^{j_n} \,) = x_1^{max(i_1, j_1)} \cdots x_n^{max(i_n, j_n)} \, .$$

Intuitiv: das Monom $KGV(hmonom(p), hmonom(q))$ ist sowohl modulo p als auch modulo q reduzierbar, S gibt die Differenz der beiden Reduktionen an.

Satz: (Charakterisierung von Gröbner-Basen) *Es sind äquivalent:*

a) G ist eine Gröbner-Basis,

b) $S(p,q) \rightarrow_G^* 0 \quad \forall p, q \in G$,

c) $\forall p \in F[x_1, \ldots, x_n]$ gilt: $p \rightarrow_G^* \tilde{p}$, $p \rightarrow_G^* \hat{p} \quad \Longrightarrow \quad \tilde{p} = \hat{p}$.

Beweis: Theorem 10.5, Seite 440 in [GCL 92].

c) besagt, daß für Gröbner-Basen die Reduktion $\hat{p} = Reduce(p, G)$ eindeutig ist.

b) liefert einen einfachen Test, ob $G = \{g_1, \ldots, g_k\}$ eine Gröbner-Basis ist: durchlaufe alle Paare $g_i, g_j \in G$ und prüfe, ob $S(g_i, g_j) \rightarrow_G^* 0$ gilt.

Dies liefert auch ein algorithmisches Konstruktionsverfahren (Buchberger) für Gröbner-Basen. Sei eine Basis $Q = \{q_1, \ldots, q_k\}$ eines Ideals $\langle Q \rangle$ gegeben. Stellt sich für ein Paar $q_i, q_j \in Q$ heraus, daß $h := Reduce(S(q_i, q_j), Q) \neq 0$ ist, so erweitert man einfach die Basis Q um $h \in \langle Q \rangle$, so dass h trivialerweise bezüglich der erweiterten Basis $Q \cup \{h\}$ von $\langle Q \rangle = \langle Q \cup \{h\} \rangle$ zu 0 reduzierbar ist:

```
# berechne eine Gröbner-Basis G mit ⟨ G ⟩ = ⟨ Q ⟩

gbasis:=proc(Q)
  G := Q;
  # Zu G = {g_1, ..., g_k} bilde die Paare g_i, g_j ∈ G:
  Pairs := {[g_i, g_j], 1 ≤ i < j ≤ k}
  while Pairs ≠ {} do
    wähle ein Paar [g_i, g_j] ∈ Pairs;                   #1)
    Pairs := Pairs \ {[g_i, g_j]};
    h := Reduce(S(q_i, q_j), G);                          #2)
    if h ≠ 0 then
      Pairs := Pairs ∪ {[g, h]; g ∈ G};
      G := G ∪ {h};                                       #3)
    fi;
  od;
  RETURN(G);
end;
```

Aus algebraischen Gründen muß der Algorithmus terminieren: Seien

$$G_1(= Q) \ \subset \ G_2 \ \subset \ G_3 \ \subset \ \ldots$$

die in `#3)` anwachsenden Basen von $\langle Q \rangle = \langle G_1 \rangle = \langle G_2 \rangle = \cdots$. Für $H_i := \{hterm(q) \ ; \ q \in Q_i\}$ gilt

$$\langle H_1 \rangle \ \subsetneqq \ \langle H_2 \rangle \ \subsetneqq \ \langle H_3 \rangle \ \subsetneqq \ \ldots$$

denn in `#3)` kommt jeweils ein neuer Term $hterm(h)$ hinzu, der nicht Vielfaches alter hterme ist. Eine solche Kette von Idealen muß auf dem Noether-Ring $F[x_1, \ldots, x_n]$ terminieren.

Durch geschickte Strategien zur Auswahl von $[g_i, g_j] \in Pairs$ in `#1)` wird versucht, die Laufzeit des Algorithmus zu minimieren. Nach gewissen Kriterien kann man sich einige der Reduktion in `#2)` ersparen, z.B. gilt bei teilerfremden $hterm(q_i), hterm(q_j)$, daß sich automatisch $h = 0$ ergibt:

Hilfssatz:

$$KGV(hterm(p), hterm(q)) = hterm(p)\, hterm(q) \quad \Longrightarrow \quad S(q,p) \rightarrow^*_{\{p,q\}} 0 \;.$$

Beweis: Theorem 10.9, Seite 449 in [GCL 92] .

Gröbner-Basen können durch einfache zusätzliche Forderungen eindeutig gemacht werden:

Definition: *$Q \subset F[x_1, \ldots, x_n]$ heißt* **reduziert**, *wenn jedes $q \in Q$ irreduzibel bezüglich $Q \setminus \{q\}$ ist. Q heißt* **monisch**, *wenn $hcoeff(q) = 1\ \forall q \in Q$ gilt.*

Satz: (Buchberger) *Für reduzierte monische Gröbner-Basen G, H mit $\langle G \rangle = \langle H \rangle$ folgt $G = H$.*

Jede Gröbner-Basis läßt sich leicht (nachträglich) in eine reduzierte Gröbner-Basis umwandeln (z.B., Algorithm 10.2 in [GCL'92]). Es ist jedoch effizienter, die reduzierten Basiselemente direkt im Laufe der Algorithmus *gbasis* zu berechnen (Details siehe Kapitel 10.4 in [GCL 92]). Durch einfache Skalierung der Basiselemente ist stets monische Form erreichbar.

Einige Anwendungen von Gröbner-Basen in der Lösung polynomialer Gleichungssysteme:

Satz: *Sei G eine Gröbner-Basis des von $Q = \{q_1, \ldots, q_k\} \subset F[x_1, \ldots, x_n]$ erzeugten Ideals. Dann hat das Gleichungssystem*

$$q_1(x_1, \ldots, x_n) = 0 \;, \quad \ldots \quad , \; q_k(x_1, \ldots, x_n) = 0$$

genau dann Lösungen, wenn G kein konstantes Polynom ($\neq 0$) enthält. (Bei einer reduzierten monischen Gröbner-Basis heißt dies $G \neq \{1\}$).

Beweis: Theorem 10.11, Seite 454 in [GCL 92] .

Satz: *Sei G eine Gröbner-Basis des von $Q = \{q_1, \ldots, q_k\} \subset F[x_1, \ldots, x_n]$ erzeugten Ideals. Dann hat das Gleichungssystem*

$$q_1(x_1, \ldots, x_n) = 0 \;, \quad \ldots \quad , \; q_k(x_1, \ldots, x_n) = 0$$

genau dann endlich viele Lösungen, wenn zu jedem $i = 1, \ldots, n$ *ein* $m_i \in \mathbb{N}$ *existiert mit*

$$x_i^{m_i} \in \{hterm(g)\ ,\ g \in G\}\ .$$

Beweis: Theorem 10.12, Seite 455 in [GCL 92] .