HÖHERE LEHRANSTALT FÜR ELEKTROTECHNIK
INFORMATIONSTECHNIK, ENERGIETECHNIK & INDUSTRIELLE ELEKTRONIK
A-3101 St. Pölten, Waldstrasse 3, Tel: +43(0)2742-75051-300, Fax -230
Homepage: http://et.htlstp.ac.at E-Mail: et@htlstp.ac.at

Maxima
Datenstrukturen und Programmieren

Wilhelm Haager
HTL St. Pölten, Abteilung Elektrotechnik
wilhelm.haager@htlstp.ac.at

8. November 2012

Inhaltsverzeichnis

1 Einführung

1.1 Begriffe

Maxima: Open-Source-Abkömmling des Computeralgebra-Systems *Macsyma*, das ursprünglich 1967–1982 im Auftrag des US-Energieministeriums am MIT entwickelt wurde. 1989 wurde eine Version von Macsyma mit dem Namen *Maxima* unter der *GNU General Public Licence* veröffentlicht und wird nun von einer unabhängigen Gruppe von Anwendern weiterentwickelt. Maxima ist in *Lisp* geschrieben und enthält selbst viele Elemente der funktionalen Programmierung.

Aufgrund seiner Leistungsfähigkeit und freien Verfügbarkeit gibt es eigentlich keinen Grund, es *nicht* zu verwenden.

wxMaxima: Eine von mehreren grafischen Benutzeroberflächen für Maxima. Es ermöglicht die grafische Ausgabe von Formeln, die direkte Ausgabe von Grafiken in das wxMaxima-Ausgabefenster, das Abspeichern von Arbeitssitzungen, sowie (für notorische Mausklickser) den Aufruf der wichtigsten Befehle über Menüs und Befehlsschaltflächen.

Bei der Installation von Maxima wird wxMaxima automatisch als Standard-Interface mitinstalliert.

1.2 Programmieren mit Maxima

Maxima arbeitet wie ein Interpreter: eingegebene Ausdrücke und Anweisungen werden sofort evaluiert bzw. ausgeführt.

In Maxima gibt es keinen formalen Unterschied zwischen *Ausdrücken* als zu verarbeitende Daten und *Anweisungen* als auszuführender Programmcode. Daher sind auch die Begriffe aus*werten* (evaluieren) und aus*führen* gleichwertig.

Eine Abfolge von Ausdrücken (Anweisungen) kann in einem File (Erweiterung `.mac`) abgespeichert und mit dem Befehl `load` oder `batch` geladen und zur Ausführung gebracht werden. Umfangreiche Funktionen und Programmkontrollstrukturen (Blöcke, Bedingungen, Schleifen) machen Maxima zu einer sehr vielseitigen Programmiersprache, mit der eigene Funktionen und ganze Programmpakete für verschiedenste Themenbereiche erstellt werden können.

Maxima ist in der Programmiersprache *Lisp* geschrieben und enthält selbst, so wie diese, viele Elemente der funktionalen Programmierung. Ein Programm im *funktionalen* Stil besteht nicht (nur), aus einer Folge von Anweisungen, sondern in erster Linie aus ineinandergeschachtelten Funktionsaufrufen; im Extremfall aus einer einzigen Schachtelung von vielen Funktionsaufrufen – einem klassischen *One-Liner*.

2 Listen

Listen, geordnete Zusammenstellungen von Elementen, bilden die grundlegende Datenstruktur von Maxima. Listenelemente können unterschiedlichen Typs sein, ein Listenelement kann selbst wieder eine Liste sein. Mit verschachtelten Listen können beliebig komplexe Datenstrukturen aufgebaut werden.

2.1 Erzeugen von Listen

`[`*x1*`,`*x2*`,...`*xn*`]`	erzeugt eine Liste mit den Elementen *x1*, *x2*, ... *xn*
w`[`*n*`]`	Zugriff auf das n-te Listenelement (sowohl lesend als auch schreibend)
w`[`*m*`][`*n*`]`	mn-tes Element einer zweifach geschachtelten Liste
`makelist(`*expr*`,`*n*`,`*n0*`,`*n1*`)`	Erzeugen einer Liste aus dem Ausdruck *expr*, der die Variable *n* enthält, wobei *n* die Ganzzahlen von *n0* bis *n1* durchläuft.
`create_list(`*expr*`,`*n*`,`*liste*`)`	Erzeugen einer Liste aus dem Ausdruck *expr*, der die Variable *n* enthält, wobei *n* jeden Wert der Liste *liste* animmt.
`read_list(`*file*`)`	Erzeugen einer Liste aus dem Datenfile *file*
`read_nested_list(`*file*`)`	Erzeugen einer verschachtelten Liste aus dem Datenfile *file*, jede Zeile im Datenfile wird zu einer Unterliste.
`copylist(`*w*`)`	Kopieren der Liste *w*

Erzeugen von Listen

Auf ein Listenelement wird mit einem ganzzahligen Index in eckigen Klammern, der bei 1 beginnt, zugegriffen. Mit `w[i]:x` kann einem Element der Liste `w` ein Wert zugewiesen werden, es kann aber kein neues Listenelement erzeugt werden. Der Versuch, eine Liste mit *n* Elementen mit `w[n+1]:x` zu erweitern, führt zu einer Fehlermeldung. Bei einer mehrfach verschachtelten Liste sind die Indizes jeweils in einer eckigen Klammer anzugeben.

Erzeugen einer Liste

```
(%i1) li1:[a,b,c,d,e];
(%o1) [a, b, c, d, e]
```

Ändern einzelner Listenelemente, mit dem Index `[n]` wird auf das n-te Element zugegriffen.

```
(%i2)
[li1[2],li1[4]]:[(li1[1]+li1[3])/2,(li1[3]+li1[5])/2];
(%o2) [(c + a)/2, (e + c)/2]
```

Die geänderte Liste

```
(%i3) li1;
(%o3) [ a , (c + a)/2 , c , (e + c)/2 , e ]
```

`makelist` und `create_list` sind die Standardfunktionen zum Erzeugen einer Liste aus einem Ausdruck, wobei eine darin enthaltene Variable für jedes Listenelement einen Ganzzahlwert in einem Bereich bzw. einen Wert aus einer Liste annimmt.

Erzeugen einer Liste mit `makelist`

```
(%i4) li2:makelist(sin(n*%pi/6),n,1,5);
(%o4) [1/2, sqrt(3)/2, 1, sqrt(3)/2, 1/2]
```

Erzeugen einer Liste mit `create_list`

```
(%i5) li3:create_list(k**2,k,li1);
(%o5) [ a^2 , (c + a)^2/4 , c^2 , (e + c)^2/4 , e^2 ]
```

Mit `read_list` und `read_nested_list` werden die Listenelemente aus einem Datenfile eingelesen. Alle Zeichen die keine gültige Zahl oder kein gültiges Symbol ergeben, dienen als Trennzeichen; bei `read_nested_list` wird jede Zeile im Datenfile zu einer Unterliste. Mathematische Ausdrücke müssen im Datenfile als Strings (in doppelte Anführungszeichen eingeschlossen) abgespeichert sein, nach dem Einlesen können sie mit der Funktion `eval_string` wieder in Ausdrücke umgewandelt werden.

Einlesen von Daten (Strings und Gleitkommazahlen) aus einem File in eine Liste

```
(%i6) daten1:read_list("d:/w/maxima/daten.txt");
(%o6) [ x1 , 0.599 , x2 , 0.062 , x3 , 0.281 , x4 ,
0.057 , x5 , 0.414 ]
```

Einlesen in eine geschachtelte Liste, jede Datenzeile wird eine Unterliste.

```
(%i7)
daten2:read_nested_list("d:/w/maxima/daten.txt");
(%o7) [ [ x1 , 0.599 ] , [ x2 , 0.062 ] , [ x3 , 0.281 ]
, [ x4 , 0.057 ] , [ x5 , 0.414 ] ]
```

Beachte: Eingelesene Texte sind Strings und *keine* Namen (von Variablen und Symbolen).

```
(%i8) [is(daten1[1]=x1),is(daten1[1]="x1")];
(%o8) [ false , true ]
```

Sie können aber mit `eval_string` in symbolische Namen umgewandelt werden.

```
(%i9) is(eval_string(daten1[1])=x1);
(%o9) true
```

`copylist(w)` erzeugt die Kopie einer Liste `w`, das Ergebnis ist eine zweite Liste mit den gleichen Elementen. Eine bloße Zuweisung gemäß `w1:w0` erzeugt mit `w1` nur eine *Referenz* auf die Liste `w0`.

Eine Zuweisung einer Liste erzeugt eine Referenz (d. h. einen neuen Namen) auf diese Liste.

```
(%i10) daten1_ref:daten1;
(%o10) [ x1 , 0.599 , x2 , 0.062 , x3 , 0.281 , x4 ,
0.057 , x5 , 0.414 ]
```

`copylist` erzeugt eine echte Kopie.

```
(%i11) daten1_copy:copylist(daten1);
(%o11) [ x1 , 0.599 , x2 , 0.062 , x3 , 0.281 , x4 ,
0.057 , x5 , 0.414 ]
```

Ändern eines Elementes der ursprünglichen Liste ...

```
(%i12) daten1[1]:"changed!";
(%o12) changed!
```

... ändert auch die Referenz, nicht aber die Kopie.

```
(%i13) [daten1_ref[1],daten1_copy[1]];
(%o13) [ changed! , x1 ]
```

Zugriff auf ein Element einer verschachtelten Liste erfolgt über mehrere Indizes; ein Index für ein Symbol, das keine Liste ist, erzeugt eine *indizierte* Variable.

```
(%i14) [daten2[1],daten2[1][2],x[1]];
(%o14) [ [ x1 , 0.599 ] , 0.599 , x1 ]
```

2.2 Elementweise Operationen

Manche Operatoren und Funktionen, auf Listen angewandt, werden auf die einzelnen Listenelemente abgebildet, das heißt, die Operationen werden elementweise durchgeführt.

`[x1,x2,...]:[y1,y2,...]`	elementweise Zuweisung
`[x1,x2,...]+[y1,y2,...]`	elementweise Addition zweier Listen
`[x1,x2,...]-[y1,y2,...]`	elementweise Subtraktion zweier Listen
`[x1,x2,...]*[y1,y2,...]`	elementweise Multiplikation zweier Listen
`[x1,x2,...]/[y1,y2,...]`	elementweise Division zweier Listen
`[x1,x2,...]+a`	elementweise Addition mit einer Zahl *a*
`[x1,x2,...]-a`	elementweise Subtraktion mit einer Zahl *a*
`[x1,x2,...]*a`	elementweise Multiplikation mit einer Zahl *a*
`[x1,x2,...]/a`	elementweise Division durch eine Zahl *a*
`[x1,x2,...]^a`	elementweises Potenzieren

Elementweise Operationen

Bei elementweisen Operationen mit zwei Listen müssen beide Listen die gleiche Anzahl von Elementen haben.

Zu beachten ist, dass nicht *alle* Operationen elementweise durchgeführt werden. Beispielsweise werden die Vergleichsoperatoren (=, <, >, ...) und die logischen Operatoren (`and`, `or`, ...) *nicht* elementweise durchgeführt. Um dies zu erreichen, müßten die Operatoren explizit mit der Funktion `map` auf die einzelnen Listenelemente abgebildet werden (siehe Kapitel 5).

Elementweise Addition zweier Listen

```
(%i15) li1+li2;
```

$$(\%o15)\quad \left[\,a+\frac{1}{2}\,,\ \frac{c+a}{2}+\frac{\sqrt{3}}{2}\,,\ c+1\,,\ \frac{e+c}{2}+\frac{\sqrt{3}}{2}\,,\ e+\frac{1}{2}\,\right]$$

Auch komplexe Berechnungen können elementweise ausgeführt werden.

```
(%i16) 25*li1^2/li2;
```

$$(\%o16)\quad \left[\,50\,a^2\,,\ \frac{25\,(c+a)^2}{2\sqrt{3}}\,,\ 25\,c^2\,,\ \frac{25\,(e+c)^2}{2\sqrt{3}}\,,\ 50\,e^2\,\right]$$

Nicht alle Operationen werden elementweise ausgeführt, beispielsweise das Gleichsetzen.

```
(%i17) li1=li2;
```

$$(\%o17)\quad \left[\,a\,,\ \frac{c+a}{2}\,,\ c\,,\ \frac{e+c}{2}\,,\ e\,\right]=\left[\,\frac{1}{2}\,,\ \frac{\sqrt{3}}{2}\,,\ 1\,,\ \frac{\sqrt{3}}{2}\,,\ \frac{1}{2}\,\right]$$

Umwandlungsfunktionen werden elementweise ausgeführt.

```
(%i18) float(li2);
(%o18) [ 0.5 , 0.866 , 1.0 , 0.866 , 0.5 ]
```

Bei einer Zuweisung wird die linke Seite sinnvollerweise *nicht* ausgewertet. Ist die linke Seite nicht elementweise angegeben, sondern nur als *Name* der Liste, muss sie durch die einzelnen Elemente ersetzt werden, also ausgewertet werden, was mit dem "Doppel-Quote-Operator" erfolgt:

```
''liste : [1,2,3]
```

2.3 Skalarwertige Listenfunktionen

Die folgenden Listenfunktionen liefern einen skalaren Wert als Ergebnis, entweder ein spezielles Listenelement oder Information über die Liste selbst.

`first(`*w*`)`	erstes Listenelement (Alternative zu *w*`[1]`)
`second(`*w*`)`	zweites Listenelement
⋮	
`tenth(`*w*`)`	10. Listenelement
`last(`*w*`)`	letztes Listenelement
`member(`*exp*`,`*w*`)`	überprüft, ob der Ausdruck *exp* Bestandteil der Liste *w* ist; liefert `true` oder `false`
`listp(`*w*`)`	überprüft, ob der Ausdruck *w* eine Liste ist; liefert `true` oder `false`
`length(`*w*`)`	Anzahl der Elemente von *w*
`lmin(`*w*`)`	Minimum aller Listenelemente
`lmax(`*w*`)`	Maximum aller Listenelemente

Skalarwertige Listenfunktionen

Für die ersten 10 Elemente und für das letzte Element gibt es eigene Zugriffsfunktionen `first` ...`tenth` bzw. `last`. Im Gegensatz zur Schreibweise *w[n]* kann aber damit nur *lesend* auf die Listenelemente zugegriffen werden, d. h. diese Funktionen dürfen nicht auf der linken Seite einer Zuweisung stehen.

Die Funktionen `length`, `first`...`last` können nicht nur auf eine Liste, sondern auf einen beliebigen Ausdruck angewandt werden. Dies ist allerdings nur dann sinnvoll, wenn die *Struktur* des Ausdrucks bekannt ist (siehe Abschnitt 5).

Die Funktionen `lmin` und `lmax` liefern das kleinste bzw. größte Listenelement, wenn alle Elemente zu einer Zahl evaluiert werden können. Ergeben nicht alle Listenelemente eine Zahl, dann ist das Ergebnis der Funktionsaufruf der Maxima-Funktionen `min` bzw. `max` als mathematischer Ausdruck.

Der Zugriff auf einzelnen Listenelemente kann über einen *Index* erfolgen, oder über eigene Zugriffsfunktionen.

```
(%i20) first(li1)-last(li2);
(%o20) a - 1/2
```

Überprüfung, ob ein Wert Element einer Liste ist (beachte: Gleitkommazahlen und Brüche sind unterschiedliche Dinge).

```
(%i21) [member(1/2,li2),member(0.5,li2)];
(%o21) [ true , false ]
```

Prüfung, ob eine Variable eine Liste ist

```
(%i22) [listp(daten1[1]),listp(daten2[1])];
(%o22) [ false , true ]
```

Anzahl der Listenelemente

```
(%i23) length(li1);
(%o23) 5
```

Minimum und Maximum von Listenelementen; ergeben einige Elemente keine Zahlenwerte, so ist das Ergebnis der Funktionsaufruf `min` oder `max`.

```
(%i24) [lmin(li2),lmax(li1)];
```

(%o24) $[\frac{1}{2}, \max\left(a, c, \frac{c+a}{2}, e, \frac{e+c}{2}\right)]$

2.4 Listenwertige Listenfunktionen

Die folgenden Listenfunktionen liefern als Ergebnis wieder Listen. Diese Funktionen verändern die als Funktionsparameter angegebene Liste *nicht*, die veränderte Liste ist der *Funktionswert*, der gegebenfalls wieder einem Namen einer Liste zugewiesen werden muss; dies kann auch der gleiche Name wie der Parameter sein, beispielsweise:

```
liste:f(liste)
```

`flatten(`*w*`)`	„Verflachen“ der Liste *x*; d. h. Unterlisten werden aufgelöst und deren Elemente „flach“ in die Liste *w* eingebettet
`cons(`*x*`,`*w*`)`	Hinzufügen des Elements *x* an den Anfang der Liste *w*
`endcons(`*x*`,`*w*`)`	Hinzufügen des Elements *x* an das Ende der Liste *w*
`rest(`*w*`,`*n*`)`	Entfernen der ersten *n* Elemente der Liste *w*
`rest(`*w*`,-`*n*`)`	Entfernen der letzten *n* Elemente der Liste *w*
`append(`*w1*`,`*w2*`)`	Zusammenketten zweier Listen *w1* und *w2*
`join(`*w1*`,`*w2*`)`	Verschränken zweier Listen *w1* und *w2*
`sublist(`*w*`,`*f*`)`	Erzeugen einer Liste mit jenen Elementen der Liste *w*, für die die Funktion *f* den Wert `true` ergibt
`delete(`*x*`,`*w*`)`	Löschen aller Elemente *x* aus der Liste *w*

Verändern von Listen

Die Funktionen `cons`, `endcons` und `rest` sind nicht auf Listen beschränkt, sondern können auf beliebige Ausdrücke angewandt werden, was allerdings nur dann sinnvoll ist, wenn die *Struktur* des Ausdrucks bekannt ist (siehe Abschnitt 5).

Verflachen einer Liste, d. h. Auflösen aller Unterlisten

```
(%i25) flatten([[[[[1]],2],3],[[4,5,6],[7,8]]]);
(%o25) [ 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 ]
```

Hinzufügen von Elementen an den Anfang und an das Ende einer Liste

```
(%i26) endcons("ende",cons("anfang",li2));
(%o26) [ anfang , 1/2 , sqrt(3)/2 , 1 , sqrt(3)/2 , 1/2 , ende ]
```

Entfernen von Listenelementen

```
(%i27) [rest(li1,2),rest(li1,-1)];
(%o27) [ [ c , (e + c)/2 , e ] , [ a , (c + a)/2 , c , (e + c)/2 ] ]
```

Aneinanderhängen zweier Listen

```
(%i28) append(li1,li2);
(%o28) [ a , (c + a)/2 , c , (e + c)/2 , e , 1/2 , sqrt(3)/2 , 1 , sqrt(3)/2 , 1/2 ]
```

Verschränken zweier Listen

```
(%i29) join(li1,li2);
(%o29) [ a , 1/2 , (c + a)/2 , sqrt(3)/2 , c , 1 , (e + c)/2 , sqrt(3)/2 , e , 1/2 ]
```

Alle Elemente einer Liste, für die eine Funktion den Wert `true` ergibt

```
(%i30) sublist(li2,integerp);
(%o30) [ 1 ]
```

Löschen von Elementen aus einer Liste

```
(%i31) delete(1/2,li2);
(%o31) [sqrt(3)/2 , 1 , sqrt(3)/2]
```

2.5 Assoziative Arrays

Im Gegensatz zu einer normalen Liste erfolgt bei einem *assoziativen Array* – in anderen Programmiersprachen auch *Hash* oder *Dictionary* genannt – der Zugriff auf ein Element, den *Wert*, nicht über einen ganzzahligen Index, sondern über einen String, den sogenannten *Schlüssel*; assoziative Arrays bestehen also aus *Schlüssel-Wert* Paaren.

In Maxima sind diese Schlüssel-Wert Paare normale Listenelemente, die auf unterschiedliche Weisen realisiert sein können: *Schlüssel* und *Wert* können zwei Elemente einer Liste sein (was am nächstliegenden ist), aber auch zwei durch einen Vergleichsoperator oder logischen Operator (=, >, <, `and`, `or`, ...) verknüpfte Elemente.

Ist der Schlüssel ein symbolischer Name (und kein String), so ist es sinnvoll, diesen durch Quotieren vor einer unbeabsichtigten Auswertung zu schützen.

`[[`*k1*`,`*v1*`],[`*k2*`,`*v2*`],...]`	assoziatives Array mit Schlüsseln *k1*, *k2*, ... und den Werten *v1*, *v2*, ...
`assoc(`*key*`,`*w*`)`	liefert den zugehörigen Wert zum Schlüssel *key* aus der Liste *w*
`assoc(`*key*`,`*w*`,`*default*`)`	Ist der Schlüssel *key* nicht vorhanden so wird der Wert *default* zurückgegeben.

Assoziative Arrays

Das Hinzufügen und Entfernen von Schlüssel-Wert Paaren erfolgt mit den Funktionen aus Abschnitt 2.4, eine Liste mit allen Schlüsseln oder allen Werten kann mit der Funktion `map` (siehe Abschnitt 5) erzeugt werden.

Unterlisten aus zwei Elementen können als Schlüssel-Wert Paare eines assoziativen Arrays angesehen werden.

```
(%i32) daten2;
(%o32) [ [ x1 , 0.599 ] , [ x2 , 0.062 ] , [ x3 , 0.281 ]
, [ x4 , 0.057 ] , [ x5 , 0.414 ] ]
```

Der Zugriff auf den *Wert* erfolgt über den *Schlüssel* mit der Funktion `assoc`.

```
(%i33) assoc("x1",daten2);
(%o33) 0.599
```

Schlüssel-Wert Paare können auch durch einen Vergleichsoperator verbunden werden; sinnvoll ist das Quotieren der Schlüssel.

```
(%i34) person:['name    > "Rudi",
               'alter   > 18,
               'freunde > ["Emil","Toni","Willi"]];
(%o34) [ name > Rudi , alter > 18 , freunde > [ Emil ,
Toni , Willi ] ]
```

Angabe eines Defaultwertes für den Zugriff mit `assoc` auf ein Element

```
(%i35) assoc('freunde,person,"nicht verfuegbar");
(%o35) [ Emil , Toni , Willi ]
```

Ist ein angegebener Schlüssel nicht existent, so wird der Defaultwert angenommen.

```
(%i36) assoc('gewicht,person,"nicht verfuegbar");
(%o36) nicht verfuegbar
```

Liste aus allen Schlüsseln

```
(%i37) map(lhs,person);
(%o37) [ name , alter , freunde ]
```

Liste aus allen Werten

```
(%i38) map(rhs,person);
(%o38) [ Rudi , 18 , [ Emil , Toni , Willi ] ]
```

3 Vektoren und Matrizen

Matrizen und Vektoren sind in Maxima nicht, wie in *Mathematica*, als zweifach geschachtelte Listen realisiert, sondern sind eine eigene Datenstruktur. Trotzdem haben Listen und Matrizen viele Gemeinsamkeiten, viele Listenfunktionen können auch auf Matrizen angewandt werden.

Vektoren sind Spezialfälle von Matrizen: Ein n-dimensionaler Spaltenvektor ist eine n×1-Matrix, ein n-dimensionaler Zeilenvektor ist eine 1×n-Matrix.

3.1 Erzeugen von Matrizen

`matrix(z1,az2,...zn)`	erzeugt eine Matrix zeilenweise, alle Zeilen *z1*, ...*zn* sind Listen mit der gleichen Anzahl von Elementen.
`matrix([a1,a2,...an])`	n-dimensionaler Zeilenvektor
`matrix([a1],[a2],...[an])`	n-dimensionaler Spaltenvektor
`A[i]`	liefert eine Liste mit den Elementen der `i`-ten Zeile der Matrix *A*
`A[i,j]`	liefert das Element der Matrix *A* in der `i`-ten Zeile und der `j`-ten Spalte
`ident(n)`	n-dimensionale Einheitsmatrix
`zeromatrix(n,m)`	n×m-Nullmatrix
`ematrix(n,m,x,i,j)`	n×m-Matrix mit dem Wert *x*, an der Stelle [*i*,*j*]; alle anderen Elemente sind 0.
`genmatrix(x,i2,j2,i1,j1)`	Generische Erzeugung einer Matrix aus einem Ausdruck *x* mit dem Zeilenindex *i1*...*i2* und dem Spaltenindex *j1*...*j2*. *i1* und *j1* können auch weggelassen werden, default: 1.
`coefmatrix([g1,...gn],[x1,...xm])`	Koeffizientenmatrix der linearen Gleichungen *g1*, ...*gn* in den Variablen *x1*, ...*xm*
`augcoefmatrix([g1,...gn],[x1,...xm])`	Erweiterte Koeffizientenmatrix der linearen Gleichungen *g1*, ...*gn* in den Variablen *x1*, ...*xm*
`jacobian([f1,...fn],[x1,...xm])`	Jacobimatrix der Funktionen *f1*,...*fn* nach den Variablen *x1*, ...*xm*
`copymatrix(A)`	Kopieren einer Matrix

Erzeugen von Matrizen

Für manche Matrizenfunktionen ist das Paket *eigen* zu laden.

Auf eine Matrixzeile wird mit einem ganzzahligen Index in eckigen Klammern, der bei 1 beginnt, zugegriffen; Mit `A[i]:w` kann eine ganze Matrixzeile mit Werten aus der Liste `w` belegt werden; zu einer bestehenden Matrix kann aber keine neue Zeile auf diese Weise hinzugefügt werden.

Mit `A[i,j]` wird auf ein Matrix*element* mit dem Zeilenindex `i` und dem Spaltenindex `j` zugegriffen, mit `A[i,j]:x` kann ein Matrixelement mit dem Wert `x` belegt werden.

Erzeugen einer 3×3-Matrix; die einzelnen Zeilen werden als Listen übergeben.

```
(%i1) A:matrix([0,1,0],[0,0,1],[-6,4,-1]);
```

$$(\%o1)\quad \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ -6 & 4 & -1 \end{bmatrix}$$

Erzeugen eines Spaltenvektors

```
(%i2) B:matrix([1],[0],[0]);
```

$$(\%o2)\quad \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

Erzeugen eines Zeilenvektors

```
(%i3) C:matrix([5,1,1]);
```

$$(\%o3)\quad \begin{bmatrix} 5 & 1 & 1 \end{bmatrix}$$

Erzeugen einer Einheitsmatrix

```
(%i4) I:ident(3);
```

$$(\%o4)\quad \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Die Funktion `genmatrix` erzeugt eine Matrix aus einem Ausdruck x. Dieser Ausdruck kann entweder ein Symbol sein, die Matrixelemente sind dann doppelt indizierte Symbole $x_{1,1}$, $x_{1,2}$, etc.; die Bereiche für die Zeilen- und Spaltenindizes werden als weitere Funktionsargumente angegeben. Die Angabe der Untergrenzen ist optional, ihr Defaultwert ist 1. x kann aber auch ein zweidimensionaler λ-Ausdruck sein, nach dem sich die einzelnen Matrixelemente berechnen, siehe Abschnitt 5.5.

Generische Erzeugung einer Matrix mit indizierten Symbolen

```
(%i5) genmatrix(a,2,3);
```

$$(\%o5)\quad \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \end{bmatrix}$$

Generische Erzeugung einer Matrix mit einem zweidimensionalen λ-Ausdruck

```
(%i6) genmatrix(lambda([u,v],a^u-b*v),2,3,1,0);
```

$$(\%o6)\quad \begin{bmatrix} a & a-b & a-2\,b & a-3\,b \\ a^2 & a^2-b & a^2-2\,b & a^2-3\,b \end{bmatrix}$$

`coefmatrix` erstellt die Koeffizientenmatrix eines linearen Gleichungssystems, `augcoefmatrix` dessen erweiterte Koeffizientenmatrix, die eine zusätzliche Spalte mit den rechten Seiten der Gleichungen enthält.

Lineares Gleichungssystem in den Variablen x und y

```
(%i7)  eqs:[2*x+5*(y-1)=x+1, 2*(3*x+y)=7];
(%o7)  [5(y-1)+2 x=x+1, 2(y+3x)=7]
```

Koeffizientenmatrix des linearen Gleichungssystems

```
(%i8)  coefmatrix(eqs,[x,y]);
```

$$(\%o8)\quad \begin{bmatrix} 1 & 5 \\ 6 & 2 \end{bmatrix}$$

Erweiterte Koeffizientenmatrix des linearen Gleichungssystems

```
(%i9)  augcoefmatrix(eqs,[x,y]);
```

$$(\%o9)\quad \begin{bmatrix} 1 & 5 & -6 \\ 6 & 2 & -7 \end{bmatrix}$$

`jacobian` ermittelt die Jacobimatrix eines Systems von Funktionen $f_1, f_2, \ldots f_m$ in den Variablen $x_1, x_2, \ldots x_n$. Das ist die Matrix der partiellen Ableitungen der Funktionen f_i nach den Variablen x_j:

$$\begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & & & \vdots \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \cdots & \frac{\partial f_m}{\partial x_n} \end{pmatrix}$$

Die Funktionen f_i und die Variablen x_i werden als Listen übergeben.

Berechnung der Jacobimatrix

```
(%i10)  jacobian([x^2/(y+1),x*y+x^2],[x,y]);
```

$$(\%o10)\quad \begin{bmatrix} \frac{2x}{y+1} & -\frac{x^2}{(y+1)^2} \\ y+2x & x \end{bmatrix}$$

`copymatrix(A)` erzeugt die Kopie einer Matrix `A`, das Ergebnis sind zwei voneinander unabhängige Matrizen. Eine bloße Zuweisung gemäß `A1:A0` erzeugt mit `A1` nur eine *Referenz* auf die Matrix `A0`.

Eine Zuweisung einer Matrix erzeugt eine Referenz (d. h. einen neuen Namen) auf diese Matrix.

```
(%i11)  C_ref:C;
(%o11)  [5  1  1]
```

`copylist` erzeugt eine echte Kopie.

```
(%i12)  C_copy:copymatrix(C);
(%o12)  [5  1  1]
```

Ändern eines Elementes der ursprünglichen Matrix ...

```
(%i13)  C[1,3]:0;
(%o13)  0
```

... ändert auch die Referenz, nicht aber die Kopie.

```
(%i14)  [C,C_ref,C_copy];
(%o14)  [[5  1  0],[5  1  0],[5  1  1]]
```

3.2 Matrixoperatoren

`A+B, A-B, A*B, A/B`	elementweise Verknüpfung zweier Matrizen A und B
`A+x, A-x, A*x, A/x, A^x`	elementweise Verknüpfung einer Matrix A mit einem Skalar x
`A.B`	Matrixmultiplikation zweier Matrizen A und B
`A^^n`	n-te Potenz der Matrix A
`invert(A)`	Inversion der Matrix A
`transpose(A)`	Transponierte von A
`ctranspose(A)`	konjugiert Transponierte von A
`scalarmatrixp`	logische Variable; bei `true` wird eine 1×1-Matrix zu einem Skalar (default), bei `false` bleibt es eine Matrix.

Matrixoperatoren

Die Listenoperatoren aus Abschnitt 2.2 arbeiten (mit Ausnahme des Zuweisungsoperators) in gleicher Weise auch für Matrizen elementweise, für die Matrixmultiplikation und das Potenzieren von Matrizen gibt es eigene Operatoren.

Entgegengesetzt den üblichen Rechenregeln kann das Matrizenprodukt „.“ auch auf zwei Zeilenvektoren, zwei Spaltenvektoren und zwei Listen (der gleichen Länge) angewandt werden, ohne die „Dimensionskompatibilität“ zu verletzen. Das Ergebnis ist in allen Fällen das innere Produkt der Vektoren.

Die Inversion einer Matrix erfolgt mit der Funktion `invert`, Transponieren (Vertauschen der Zeilen und Spalten) mit `transpose`.

Elementweise Verknüpfungen bei einer Matrix

```
(%i15) (C+a)^2;
(%o15) matrix([(a+5)^2, (a+1)^2, a^2])
```

Inversion einer Matrix

```
(%i16) A1:invert(A);
(%o16) matrix([2/3, -1/6, -1/6], [1, 0, 0], [0, 1, 0])
```

Matrizenprodukt und elementweises Produkt

```
(%i17) [A.A1, A*A1];
(%o17) [matrix([1, 0, 0], [0, 1, 0], [0, 0, 1]), matrix([0, -1/6, 0], [0, 0, 0], [0, 4, 0])]
```

Potenzieren einer Matrix und elementweises Potenzieren

```
(%i18) [A^^2, A^2];
(%o18) [matrix([0, 0, 1], [-6, 4, -1], [6, -10, 5]), matrix([0, 1, 0], [0, 0, 1], [36, 16, 1])]
```

Das innere Produkt zweier Vektoren ist das Matrizenprodukt eines Zeilenvektors mit einem Spaltenvektor.

```
(%i19) matrix([a,b,c]).transpose(matrix([d,e,f]));
(%o19) c f + b e + a d
```

Auch das Matrizenprodukt zweier Zeilenvektoren (oder Spaltenvektoren) liefert das innere Produkt.

```
(%i20) matrix([a,b,c]).matrix([d,e,f]);
(%o20) c f + b e + a d
```

Auch das Matrizenprodukt zweier Listen liefert das innere Produkt der entsprechenden Vektoren.

```
(%i21) [a,b,c].[d,e,f];
(%o21) c f + b e + a d
```

Ist das Ergebnis von Matrizenoperationen eine 1×1-Matrix, so wird es zu einem Skalar.

```
(%i22) ratsimp(C.invert(s*I-A).B);
```

$$(\%o22)\quad \frac{5\,s^2+5\,s-26}{s^3+s^2-4\,s+6}$$

Wird die globale Variable `scalarmatrixp` auf `false` gesetzt, ...

```
(%i23) scalarmatrixp:false;
(%o23) false
```

... so bleibt eine 1×1-Matrix als Matrix bestehen.

```
(%i24) ratsimp(C.invert(s*I-A).B);
```

$$(\%o24)\quad \left[\frac{5\,s^2+5\,s-26}{s^3+s^2-4\,s+6}\right]$$

Transponieren einer Matrix

```
(%i25) transpose(B);
```

$$(\%o25)\quad \begin{bmatrix}1 & 0 & 0\end{bmatrix}$$

3.3 Kenngrößen von Matrizen

Die Funktion `eigenvalues(A)` liefert eine Liste bestehend aus zwei Unterlisten: eine Liste mit den Eigenwerten der Matrix A, sowie eine Liste mit deren Vielfachheiten. Das analytisch berechnete Ergebnis besteht mitunter aus sehr komplexen Ausdrücken. Es ist dann sinnvoll, das Ergebnis mit Hilfe anderer Funktionen zu vereinfachen, zum Beispiel mit

```
rectform(float(eigenvalues(A))).
```

`eigenvectors(A)` liefert eine Liste bestehend aus mehreren Unterlisten: die erste Unterliste entspricht dem Ergebnis der Funktion `eigenvalues`, besteht also aus den Eigenwerten und ihren Vielfachheiten, die weiteren Unterlisten enthalten die Komponenten der Eigenvektoren.

Überprüfung eines Symbols, ob es eine Matrix ist

```
(%i26) [matrixp(A), matrixp(a)];
(%o26) [ true , false ]
```

Dimension einer Matrix als zweielementige Liste

```
(%i27) matrix_size(A);
(%o27) [ 3 , 3 ]
```

Die Spur einer Matrix ist die Summe aller Hauptdiagonalelemente.

```
(%i28) mat_trace(s*I-A);
(%o28) 3 s + 1
```

Berechnung der Determinante

```
(%i29) expand(determinant(s*I-A));
(%o29) s^3 + s^2 - 4 s + 6
```

Der Rang einer Matrix ist die Anzahl der linear unabhängigen Zeilen- oder Spaltenvektoren.

```
(%i30) rank(A);
(%o30) 3
```

`matrixp(`*A*`)`	überprüft, ob der Ausdruck *A* eine Matrix ist; liefert `true` oder `false`
`matrix_size(`*A*`)`	liefert eine Liste mit Zeilenzahl und Spaltenzahl von *A*
`length(`*A*`)`	Zeilenzahl von *A*
`length(`*A*`[1])`	Spaltenzahl von *A*
`mat_trace(`*A*`)`	Spur von *A* (Summe aller Hauptdiagonalelemente)
`determinant(`*A*`)`	Determinante von *A*
`rank(`*A*`)`	Rang von *A*
`charpoly(`*A*`,`*s*`)`	Charakteristisches Polynom *A* mit der Variable *s*
`eigenvalues(`*A*`)`, `eivals(`*A*`)`	Berechnung der Eigenwerte von *A* und ihrer Vielfachheiten
`eigenvectors(`*A*`)`, `eivects(`*A*`)`	Berechnung der Eigenvektoren und Eigenwerte von *A*
`mat_norm(`*A*`,`*type*`)`	Matrixnorm von *A*; gültige Werte für *type*: `1` ... Eins-Norm `frobenius` ... Frobenius-Norm `inf` ... Unendlich-Norm

Kenngrößen von Matrizen

Charakteristisches Polynom von *A* in der Variable *s*

```
(%i31) p:expand(charpoly(A,s));
(%o31) - s^3 - s^2 + 4 s - 6
```

Eigenwerte der Matrix *A* und ihre Vielfachheiten

```
(%i32) eigenvalues(A);
(%o32) [ [ 1 - %i , %i + 1 , - 3 ] , [ 1 , 1 , 1 ] ]
```

Die Eigenwerte sind die Nullstellen des charakteristischen Polynoms.

```
(%i33) solve(p=0,s);
(%o33) [ s = 1 - %i , s = %i + 1 , s = - 3 ]
```

Verschachtelte Liste aus Eigenwerten (als Ergebnis der Funktion `eigenvalues`) und Eigenvektoren

```
(%i34) ev:eigenvectors(A);
(%o34) [ [ [ 1 - %i , %i + 1 , - 3 ] , [ 1 , 1 , 1 ] ] , [ 1
, 1 - %i , - 2 %i ] , [ 1 , %i + 1 , 2 %i ] , [ 1 , - 3 , 9 ]
]
```

1-Norm (Spaltensummen-Norm) einer Matrix

```
(%i35) mat_norm(A,1);
(%o35) 6
```

Frobeniusnorm (euklidische Norm) einer Matrix

```
(%i36) mat_norm(A,frobenius);
(%o36) sqrt(55)
```

∞-Norm (Zeilensummen-Norm) einer Matrix

```
(%i37) mat_norm(A,inf);
(%o37) 11
```

3.4 Verändern von Matrizen, Extrahieren von Teilen

Die folgenden Funktionen haben nicht den *Nebeneffekt*, die als Funktionsparameter angegebene Matrix zu verändern, sie liefern die geänderte Matrix als *Funktionswert* zurück, dem gegebenfalls wieder ein Name zugewiesen werden muss. Dieser kann auch der gleiche Name wie der Parameter sein, beispielsweise:

```
A:f(A)
```

`addcol(A,s1,s2,...)`	Hinzufügen von Spalten zur Matrix *A*; *s1*, *s2*, ... sind Listen.
`addrow(A,r1,r2,...)`	Hinzufügen von Zeilen zur Matrix *A*; *r1*, *r2*, ... sind Listen.
`col(A,j)`	liefert die j-te Spalte der Matrix *A*; das Ergebnis ist ein Spaltenvektor
`row(A,i)`	liefert die i-te Zeile der Matrix *A*; das Ergebnis ist ein Zeilenvektor
`submatrix(i1,...im,A)`	entfernt die Zeilen *i1*, ...*im* aus der Matrix *A*
`submatrix(A,j1,...jn)`	entfernt die Spalten *j1*, ...*jn* aus der Matrix *A*
`submatrix(i1,...im,A,i1,...in)`	entfernt die entsprechenden Zeilen und Spalten aus *A*
`gramschmidt(A,i)`	Gram-Schmidt Orthogonalisierung von *A*
`triangularize(A)`	Berechnung der oberem Dreiecksmatrix durch Gaußsche Elimination

Verändern von Matrizen, Extrahieren von Teilen

`gramschmidt(A)` führt die Gram-Schmidt-Orthogonalisierung aus. *A* kann dabei eine Matrix oder eine zweifach geschachtelte Liste sein. Das Ergebnis ist eine Liste von Listen, die jeweils die Komponenten der orthogonalen Vektoren enthalten.

Hinzufügen von Spalten zu einer Matrix

```
(%i38) addcol(B,[a,b,c]);
```

$$(\%o38)\quad \begin{bmatrix} 1 & a \\ 0 & b \\ 0 & c \end{bmatrix}$$

Hinzufügen von Zeilen zu einer Matrix

```
(%i39) addrow(C,[a,b,c]);
```

$$(\%o39)\quad \begin{bmatrix} 5 & 1 & 0 \\ a & b & c \end{bmatrix}$$

Matrixzeile als Zeilenvektor

```
(%i40) row(genmatrix(a,4,4),2);
```

$$(\%o40)\quad \begin{bmatrix} a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \end{bmatrix}$$

Matrixspalte als Spaltenvektor

```
(%i41) col(genmatrix(a,4,4),2);
```

$$(\%o41)\quad \begin{bmatrix} a_{1,2} \\ a_{2,2} \\ a_{3,2} \\ a_{4,2} \end{bmatrix}$$

Untermatrix aus der zweiten und dritten Zeile einer Matrix

```
(%i42) submatrix(2,3,genmatrix(a,4,4));
```

$$(\%o42)\quad \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} \end{bmatrix}$$

Untermatrix aus der zweiten und dritten Spalte einer Matrix

```
(%i43) submatrix(genmatrix(a,4,4),2,3);
```

$$(\%o43)\quad \begin{bmatrix} a_{1,1} & a_{1,4} \\ a_{2,1} & a_{2,4} \\ a_{3,1} & a_{3,4} \\ a_{4,1} & a_{4,4} \end{bmatrix}$$

Orthogonalisierung nach Gram-Schmidt

```
(%i44) gs:gramschmidt(matrix([a,b],[c,d]));
```

$$(\%o44)\quad [\,[\,a\,,\,b\,]\,,\,[\,-\frac{b\,(a\,d-b\,c)}{b^2+a^2}\,,\,\frac{a\,(a\,d-b\,c)}{b^2+a^2}\,]\,]$$

Das innere Produkt der Ergebnisvektoren ist 0.

```
(%i45) gs[1].gs[2];
(%o45) 0
```

Erzeugung einer oberen Dreiecksmatrix nach dem Gaußschen Eliminationsverfahren

```
(%i46) triangularize(matrix([a,b],[c,d]));
```

$$(\%o46)\quad \begin{bmatrix} a & b \\ 0 & a\,d-b\,c \end{bmatrix}$$

4 Prozedurales Programmieren

Ausdrücke und Anweisungen werden sequentiell eingegeben oder von einem File eingelesen und sofort ausgewertet bzw. ausgeführt. Eine Abfolge von Anweisungen entspricht dem Grundprinzip des *prozeduralen Programmierens*, wobei der Ablauf durch die klassischen Programmkontrollstrukturen *Block*, *Verzweigung* und *Schleife* gesteuert werden kann.

In Maxima wird jeder Ausdruck durch einen Strichpunkt oder das Dollarzeichen abgeschlossen, wobei das Dollarzeichen die Ausgabe des Ergebnisses unterdrückt. In der Benutzeroberfläche *wxMaxima* kann die Eingabe des Strichpunktes bei Tastatureingabe entfallen, er wird vom Programm automatisch hinzugefügt.

4.1 Sequenzen, Blöcke

Sequenzen und Blöcke dienen zum Zusammenfassen mehrer Ausdrücke, die nach außen wie ein einziger Ausdruck wirken. Die einzelnen Ausdrücke werden durch einen (genau einen, nicht mehr!) Beistrich *getrennt*. Das Ergebnis ist der Wert des letzten Ausdrucks.

Ein Block kann mit `return(x)` vorzeitig verlassen werden, der Parameter `x` wird zum Ergebnis des Blocks. Innerhalb eines Blocks können lokale Variable deklariert werden; bei einer Verschachtelung von Blöcken sind diese in allen untergeordneten Blöcken sichtbar.

`(`*exp1*`,`*exp2*`,...`*expn*`)`	Zusammenfassen der Ausdrücke *exp1*, *exp2*, ...*expn* zu einer Sequenz, Rückgabewert ist das Ergebnis des letzten Ausdrucks *expn*.
`block(`*exp1*`,`*exp2*`,...`*expn*`)`	Zusammenfassen der Ausdrücke *exp1*, *exp2*, ...*expn* zu einem Block, Rückgabewert ist das Ergebnis von *expn*.
`block([`*x1*`,`*x2*`,...`*xn*`],`*exp1*`,`*exp2*`,...`*expn*`)`	Block mit lokalen Variablen *x1*, *x2*, ...*xn*
`return(`*z*`)`	Verlassen des Blocks und Rückgabe von *z* als Ergebnis

Sequenzen, Blöcke

Eine *Sequenz* fasst mehrere durch Beistrich getrennte Ausdrücke zusammen, das Ergebnis ist der letzte Ausdruck.

```
(%i1) f1:(z:s+1,n:s^2+2*s+1,z/n);
(%o1)                          s + 1
                             ---------
                              2
                             s  + 2 s + 1
```

Alle in einer Sequenz verwendeten Variablen sind *global*.

```
(%i2) [z,n];
                                  2
(%o2)                  [ s + 1 , s  + 2 s + 1 ]
```

Ein *Block* ist ähnlich einer Sequenz, erlaubt aber die Deklaration *lokaler* Variablen.

```
(%i3)
f2:block([z,n],z:s+a,n:s^2+b*s+c,z/n);
                                s + a
(%o3)                       ---------
                             2
                            s  + b s + c
```

Änderung lokaler Variable kat keine Auswirkung auf Bereiche außerhalb des Blocks.

```
(%i4)  [z,n];
(%o4)  [ s + 1 , s^2 + 2 s + 1 ]
```

4.2 Bedingungen (if-Anweisung)

`if` *bed* `then` *expr1* `else` *expr2*	Bedingte Auswertung von Ausdrücken *expr1* oder *expr2* (Programmverzweigung) in Abhängigkeit der logischen Werte von *bed*; der else-Teil kann auch entfallen.
`if` *bed1* `then` *expr1* `elseif` *bed2* `then` *expr2* `elseif` *bed3* `then` *expr3* ⋮ `else` *exprn*	Mehrfachverzweigung

Bedingungen

Der Ausdruck *bed* muss einen logischen Wert (`true` oder `false`) ergeben, andernfalls erfolgt eine Fehlermeldung. Wird *bed* zum Wert `true` evaluiert, dann ist das Ergebnis der if-Anweisung der Wert von *expr1*, wird er zu `false` evaluiert, dann ist das Ergebnis der Wert von *expr2*. Ist der optionale else-Teil nicht vorhanden und liefert die Bedingung *bed* den Wert *false*, so ist das Ergebnis der gesamten if-Anweisung ebenfalls `false`.

Im *prozeduralen* Programmierstil wird das Ergebnis der if-Anweisung üblicherweise *nicht* verwendet (zum Beispiel einem Symbol zugewiesen), sondern die Ausdrücke *expr1* und *expr2* sind selbst Anweisungen. Sind mehrere Anweisungen bedingt auszuführen, so sind diese in einer Sequenz oder in einem Block zusammenzufassen.

Bedingung im prozeduralen Stil

```
(%i5)  if random(5.0)>2.5 then res:"gross"
       else res:"klein";
(%o5)  gross
```

„Toggeln“ einer logischen Variable

```
(%i6)  if polyfactor then polyfactor:false
        else polyfactor:true;
(%o6)  true
```

Zuweisungen

```
(%i7)  [zeit:10, summe:0, pz:[]];
(%o7)  [ 10 , 0 , [ ] ]
```

Mehrfachentscheidung

```
(%i8)
if zeit<9 then gruss:"Guten Morgen!"
else if zeit>18 then gruss:"Guten Abend"
else gruss:"Guten Tag!";
(%o8)  Guten Tag!
```

4.3 Schleifen

`for` *z* `:` *z0* `step` *dz* `thru` *z1* `do` *expr*	Wiederholte Ausführung von *expr*, die Anzahl der Durchläufe wird mit der Steuervariablen *z*, die mit der Schrittweite *dz* von *z0* bis *z1* läuft, bestimmt.
`for` *z* `:` *z0* `step` *dz* `while` *bed* `do` *expr*	Wiederholte Ausführung von *expr* solange *bed* den Wert `true` ergibt.
`for` *z* `:` *z0* `step` *dz* `unless` *bed* `do` *expr*	Wiederholte Ausführung von *expr* bis *bed* den Wert `true` ergibt.
`while/unless` *bed* `do` *expr*	Verkürzte Form der Schleife, die Änderung des steuernden Ausdrucks muss hier im Schleifenkörper *expr* erfolgen.

Schleifen

Ist die Änderung der steuernden Variable in jedem Schleifendurchlauf 1, so kann der step-Teil entfallen. Wird der Steuerungsteil `for` *z* `:` *z0* `step` *dz* weggelassen, so sind die entsprechenden Mechanismen zum Beenden der Laufbedingung zur Gänze im Schleifenkörper *expr* zu realisieren, der in diesem Fall ein Block aus mehreren Ausdrücken sein wird. Auch ein alleinstehendes `do` *expr* wäre syntaktisch erlaubt, ist aber kaum sinnvoll, da es zu einer Schleife ohne Abbruchbedingung, also zu einer Endlosschleife führt.

Die Steuervariable *z* ist innerhalb der Schleife lokal, sie beeinflusst eine gleichnamige Variable außerhalb der Schleife nicht. Die Schleife selbst liefert als Ergebnis keinen sinnvoll weiterverwendbaren Wert, der sich aus der Berechnungen innerhalb der Schleife ergibt, sondern immer das Symbol `done`.

Beachte: `unless` bedeutet hier *nicht* (wie in Perl) „wenn nicht“, sondern „solange nicht“.

Liste mit Zahlenwerten

```
(%i9)
werte:[175.5,168.2,188.0,176.3,199.0,186.5];
(%o9)  [ 175.5 , 168.2 , 188.0 , 176.3 , 199.0
, 186.5 ]
```

Aufsummieren der Listenelemente in einer Schleife

```
(%i10)  for n:1 thru length(werte)
            do summe:summe+werte[n];
(%o10)  done
```

Berechnung des Mittelwertes der Listenelemente

```
(%i11)  mw:summe/length(werte);
(%o11)  182.25
```

Aufbau einer Liste von Primzahlen in einer Schleife.
Beachte: Die Schleife selbst liefert das Symbol `done`.

```
(%i12)
for n:2 while n<30 do
    if primep(n) then pz:endcons(n,pz);
(%o12)  done
```

Die Liste mit den Primzahlen

```
(%i13)  pz;
(%o13)  [ 2 , 3 , 5 , 7 , 11 , 13 , 17 , 19 , 23, 29 ]
```

5 Funktionales Programmieren

Eine Verschachtelung von Funktionsaufrufen entspricht dem Grundprinzip des *funktionalen Programmierens*, wobei Schleifen üblicherweise durch Rekursionen oder Funktionen ersetzt werden, die eine wiederholte Berechnung implizit durchführen oder die automatisch auf eine Reihe von Werten (zum Beispiel in einer Liste) angewandt werden.

Erstellung einer Liste von Primzahlen im funktionalen Stil (ohne Schleife)

```
(%i1) sublist(makelist(k,k,2,30),primep);
(%o1) [ 2 , 3 , 5 , 7 , 11 , 13 , 17 , 19 , 23 , 29 ]
```

5.1 Die Struktur von Ausdrücken

Unabhängig von der Komplexität und der äußeren Erscheinung besteht jeder zusammengesetzte Ausdruck aus einem *Operator* (oder Funktionsnamen) und aus *Operanden* (oder Argumenten). Prinzipiell besteht kein Unterschied zwischen einem Operator und einer Funktion, nur ihre „äußere Erscheinung“ ist unterschiedlich: Operatoren werden meist durch Symbole angegeben (zum Beispiel $+$, $*$, %, $=$) und *verknüpfen* Ausdrücke, Funktionen werden meist durch Namen angegeben (zum Beispiel sin, cos, exp, log, max) und haben Ausdrücke als *Parameter*.

Die Operanden sind entweder sogenannte *Atome*, das sind Ausdrücke, die nicht weiter zerlegt werden können (Zahlen und nicht weiter auswertbare Symbole), oder selbst wieder zusammengesetzte Ausdrücke. Damit erhält man für einen zusammengesetzten Ausdruck eine baumartige Struktur mit dem „äußersten“ Operator, dem *Hauptoperator*, als Wurzel. Maxima-intern (bzw. Lisp-intern) ist ein Ausdruck als verschachtelte Liste abgespeichert, wobei jeweils das erste Listenelement den Operator darstellt.

Beispielsweise hat der den Ausdruck

$$u = \sqrt{1+\sin^2 x}$$

folgenden inneren Aufbau, wobei die Operatoren zwecks leichterer Verständlichkeit durch Funktionsnamen ersetzt wurden.

$$equal(u,sqrt(plus(1,power(sin(x),2))))$$

Jede mit einem Operatorsymbol durchgeführte *Verknüpfung* mehrerer Elemente kann auch wie eine *Funktion* angegeben werden, wobei das Operatorsymbol in doppelte Anführungszeichen einzuschließen ist; beispielsweise sind folgende beiden Darstellungen für eine Addition zulässig und gleichwertig:

`a+b+c` ist gleichbedeutend mit `"+"(a,b,c)`

`op(`*expr*`)`	liefert den Hauptoperator des Ausdrucks *expr*.
`args(`*expr*`)`	liefert eine Liste mit den Hauptoperanden des Ausdrucks *expr*.
`length(`*expr*`)`	liefert die Anzahl der Hauptoperanden des Ausdrucks *expr*.
`part(`*expr*`,`*n*`)`	liefert den *n*-ten Hauptoperanden des Ausdrucks *expr*, d. h. jenen Teilausdruck, der dem Index *n* entspricht.
`first(`*w*`),...last(`*w*`)`	erster, ... letzter Hauptoperand (Alternative zu `part`
`part(`*expr*`,`*n1*`,`*n2*`,...)`	liefert jenen Teil von *expr*, der den Indizes *n1*, *n2*, ... entspricht.

Die Struktur von Ausdrücken

Mit der Funktion `args` kann jeder Ausdruck in eine Liste umgewandelt werden; genaugenommen wird dabei nur der Hauptoperator durch den Listenoperator `"["` ersetzt. Der Hauptoperator eines Ausdrucks *expr* wird als „nullter“ Teilausdruck gesehen. Alternativ zu `op(`*expr*`)` kann auch `part(`*expr*`,0)` geschrieben werden.

Die Funktionen `first`, `... last` und `part` sind nur dann sinnvoll anwendbar, wenn die innere Struktur des Ausdrucks bekannt ist; bei kommutativen Operatoren werden die Operanden von Maxima in eine Standardreihenfolge gebracht.

Zusammengesetzter Ausdruck

```
(%i2) z:fun(a)+2^b-5*c+d/(e+f+g);
```

$$(\%o2)\quad \frac{d}{g+f+e} - 5\,c + 2^{b} + \mathrm{fun}(a)$$

Hauptoperator des Ausdrucks

```
(%i3) op(z);
(%o3) +
```

Liste mit den Operanden des Ausdrucks

```
(%i4) args(z);
```

$$(\%o4)\quad \left[\frac{d}{g+f+e}\,,\ -5\,c\,,\ 2^{b}\,,\ \mathrm{fun}(a)\right]$$

Der Hauptoperator ist auch der „nullte“ Teil eines Ausdrucks.

```
(%i5) part(z,0);
(%o5) +
```

Die Funktionen *first*, *second*, ... *last* sind auf beliebige zusammengesetzte Ausdrücke anwendbar.

```
(%i6) [first(z),last(z)];
```

$$(\%o6)\quad \left[\frac{d}{g+f+e}\,,\ \mathrm{fun}(a)\right]$$

Zugriff auf einen beliebigen Teilausdruck in beliebiger Tiefe

```
(%i7) [part(z,1),part(z,1,2),part(z,1,2,1)];
```

$$(\%o7)\quad \left[\frac{d}{g+f+e}\,,\ g+f+e\,,\ g\right]$$

5.2 Bedingungen (if-Ausdruck)

Formal besteht kein Unterschied zur *if-Anweisung* im prozeduralen Stil (Abschnitt 4.2)

Im *funktionalen* Programmierstil enthält ein bedingter Ausdruck aber keine Zuweisungen, sondern ist selbst Teil eines größeren Ausdrucks Dabei wird nur das Endergebnis des *if*-Zweiges oder des *else*-Zweiges weiterverwendet, gegebenenfalls als Aufrufparameter einer weiteren Funktion.

Bedingung im *funktionalen* Stil

```
(%i8)
polyfactor:if polyfactor then false else true;
(%o8)  true
```

5.3 Ersetzen von Operatoren und Teilausdrücken

`apply(`*f*`,`*liste*`)`	Anwenden der Funktion *f* auf die Elemente der Liste *liste*
`substpart(`*x*`,`*expr*`,`*n1*`,`*n2*`,...)`	Ersetzen jenes Teilausdrucks von *expr*, dessen Position durch die Indizes *n1*, *n2*, ... festgelegt ist, durch *x*

Ersetzen von Teilausdrücken und Operatoren

`apply` erzeugt einen Funktionsaufruf der Funktion *f* mit den Elementen einer Liste als Parameter. Genaugenommen wird dabei nur der Listenoperator `"["` durch den Operator *f* ersetzt. Die Anwendung von `apply` ist auf Listen beschränkt; soll es auf einen beliebigen Ausdruck angewandt werden, so muss dieser zuerst mit Hilfe der Funktion *args* in eine Liste umgewandelt werden.

Mit der Funktion `substpart` kann ein beliebiger Teilausdruck, dessen Position im Gesamtausdruck durch die Indizes *n1*, *n2*, ... festgelegt ist, durch einen Ausdruck *x* ersetzt werden. Das Ersetzen des „nullten“ Teilausdrucks entspricht dem Ersetzen des Hauptoperators, ist also eine allgemeinere Form von `apply`, dessen Anwendung nicht auf Listen beschränkt ist.

Liste mit Zahlenwerten

```
(%i9)  werte:[175.5,168.2,188.0,176.3,199.0,186.5];
(%o9)  [ 175.5 , 168.2 , 188.0 , 176.3 , 199.0 , 186.5 ]
```

Berechnung des Mittelwertes ohne Schleife; der Listenoperator „[“ wird durch „+“ ersetzt.

```
(%i10)  mw:apply("+",werte)/length(werte);
(%o10)  182.25
```

Anwenden einer Funktion *f* auf eine Liste, d. h. die Listenelemente werden zu Funktionsargumenten.

```
(%i11)  apply(f,[a,b,c,d,e]);
(%o11)  f(a , b , c , d , e)
```

Verschachtelte Liste

```
(%i12)  liste:[[a,b],[c,d]];
(%o12)  [ [ a , b ] , [ c , d ] ]
```

Umwandlung einer verschachtelten Liste in eine Matrix

```
(%i13)  mat:apply(matrix,liste);
```

$$(\%o13)\quad \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

Umwandlung einer Matrix in eine verschachtelte Liste

```
(%i14) args(mat);
(%o14) [[a,b],[c,d]]
```

Ersetzen eines beliebigen Teilausdrucks

```
(%i15) substpart(xxx,z,1,2,1);
                          d
(%o15)                ------- - 5 c + 2^b + fun(a)
                      xxx+f+e
```

5.4 Abbilden von Funktionen

`map(`*f*`,`*expr*`)`	Abbilden der Funktion *f* auf die Hauptoperanden des Ausdrucks *expr*
`maplist(`*f*`,`*expr*`)`	wie `map`, wandelt jedoch *expr* in eine Liste um.
`map(`*f*`,`*expr1*`,`*expr2*`,...)`	Verknüpfung der entsprechenden Hauptoperanden von *expr1*, *expr2*, ... durch den Operator *f*
`fullmap(`*f*`,`*expr*`)`	Abbilden der Funktion *f* auf jeden Teilausdruck von *expr*
`fullmapl(`*f*`,`*expr*`)`	wie `fullmap`, kann aber nur auf Listen und Matrizen angewandt werden.

Abbilden von Funktionen

`map` wendet eine Funktion *f* auf jeden Operanden (das heißt auf die Argumente des Hauptoperators) eines Ausdrucks an, `fullmap` geht in die Tiefe und wendet `f` auf jeden Teilausdruck an. Häufigster Anwendungsfall von `map` ist die elementweise Behandlung oder Verknüpfung von Listenelementen.

Abbilden einer Funktion *f* auf eine Liste, d. h. *f* wird auf jedes Listenelement angewandt.

```
(%i16) map(f,[a,b,c,d,e]);
(%o16) [f(a),f(b),f(c),f(d),f(e)]
```

Nicht alle Funktionen werden elementweise abgebildet, ...

```
(%i17) log(werte);
(%o17)
log([175.5,168.2,188.0,176.3,199.0,186.5])
```

... was aber mit `map` erzwungen werden kann.

```
(%i18) map(log,werte);
(%o18) [5.1676,5.1252,5.2364,5.1722,5.2933,
5.2284]
```

`map` ist nicht nur auf Listen, sondern auf beliebige Ausdrücke anwendbar.

```
(%i19) map(f,a+b+c+d);
(%o19) f(d)+f(c)+f(b)+f(a)
```

`maplist` ist wie `map`, erzeugt aber immer eine Liste.

```
(%i20) maplist(f,a+b+c+d);
(%o20) [f(d),f(c),f(b),f(a)]
```

Abbilden einer Funktion *f* auf jeden *elementaren* Teilausdruck

```
(%i21) fullmap(f,z);
                        f(d)                 f(b)
(%o21)              ---------- - f(5) f(c) + f(2)    + fun(f(a))
                    f(g)+f(f)+f(e)
```

Verknüpfen zweier Listen mit einem Operator

```
(%i22) map(f,[a,b,c,d],[1,2,3,4]);
(%o22) [f(a,1),f(b,2),f(c,3),f(d,4)]
```

Nicht alle Operatoren verknüpfen zwei Listen elementweise, ...

```
(%i23) [a,b,c,d]=[1,2,3,4];
(%o23) [ a , b , c , d ] = [ 1 , 2 , 3 , 4 ]
```

... was aber mit `map` erzwungen werden kann.

```
(%i24) map("=",[a,b,c,d],[1,2,3,4]);
(%o24) [ a = 1 , b = 2 , c = 3 , d = 4 ]
```

5.5 Anonyme Funktionen

Wird eine (aus mehreren Teilen zusammengesetzte) Funktion nur einmal benötigt, so wäre es umständlich, sie als *benannte* Funktion zu definieren (siehe Abschnitt 6.1) um sie nur ein einziges Mal zu verwenden. Mit einem *λ-Ausdruck* wird eine Funktion direkt bei ihrer einmaligen Verwendung (beispielsweise in einem `map` oder `apply`) definiert, ohne ihr einen Namen zu geben (*anonyme Funktion*).

`lambda([`*x*`],`*expr*`)`	Definieren des Ausdrucks *expr* als anonyme Funktion mit dem formalen Parameter *x*; *expr* kann auch aus mehreren durch Beistriche getrennten Ausdrücken bestehen.
`lambda([`*x1*`,`*x2*`,...],`*expr*`)`	Anonyme Funktion in mehreren Variablen *x1*, *x2*, ...

Anonyme Funktionen

Abbilden einer *anonymen Funktion* auf eine Liste

```
(%i25) map(lambda([u],sin(u)/u),[a,b,c,d,e,f]);
(%o25) [sin(a)/a, sin(b)/b, sin(c)/c, sin(d)/d, sin(e)/e, sin(f)/f]
```

Abbilden einer *anonymen Funktion* auf einen beliebigen Ausdruck

```
(%i26) map(lambda([u],sin(u)/u),a+b+c+d+e+f);
(%o26) sin(f)/f + sin(e)/e + sin(d)/d + sin(c)/c + sin(b)/b + sin(a)/a
```

Bedingter Ausdruck als anonyme Funktion; hier entstehen gänzlich neue Listenelemente.

```
(%i27)
map(lambda([u],if u>180 then "g" else "k"),werte);
(%o27) [ k , k , g , k , g , g ]
```

Mehrdimensionaler λ-Ausdruck, d. h. eine anonyme Funktion in mehreren Variablen

```
(%i28)
map(lambda([u,v],u(v)),[sin,func,hallo],[x1,x2,x3]);
(%o28) [ sin(x1), func(x2), hallo(x3)]
```

6 Funktionen und Pakete

6.1 Benannte Funktionen

f(*x1*,*x2*,...*xn*):=*expr*	Definieren einer Funktion *f* mit den formalen Parametern *x1*, *x2*, ...*xn*
f(*x1*,*x2*,...*xn*,[opts]):=*expr*	Funktion *f* mit den formalen Pflichtparametern *x1*, *x2*, ...*xn* und zusätzlichen optionalen Parametern *opts*

Benannte Funktionen

Der Ausdruck *expr* ist meist ein *Block* (Abschnitt 4.1) mit zusätzlichen lokalen Variablen.

Vorteilhaft für optionale Parameter sind Schlüssel-Wert-Paare im Stil eines assoziativen Arrays in der Form:

$$option1 = value1, \quad option2 = value2, \quad \ldots$$

Bei entsprechender Auswertung im Funktionskörper spielt die Reihenfolge der Parameter keine Rolle, bei Weglassen einzelner Parameter können Defaultwerte zugewiesen werden.

Setzen globaler Variablen: Alle Variablen werden als positiv angenommen, die Ausgabegenauigkeit wird auf 5 signifikante Stellen gesetzt.

```
(%i1)  [assume_pos,fpprintprec]:[true,5];
(%o1)  [ true , 5 ]
```

Definition einer Funktion, die ein Polynom *n*-ter Ordnung in *s* mit zufälligen Koeffizienten zwischen 1 und 10 erzeugt.

```
(%i2)  ranpoly(n) :=
       block([p:0],
         for i:0 step 1 thru n do
         p: p + (1+random(10)) * 's**i,
         p
       )$
```

Erzeugung eines Zufallspolynoms

```
(%i3)  p:ranpoly(6);
(%o3)  10 s^6 + 2 s^5 + 5 s^4 + 6 s^3 + 5 s^2 + 3 s + 3
```

Definition einer Funktion, die aus einem Polynom eine Liste mit den Polynomkoeffizienten erzeugt.

```
(%i4)  coefficient_list(p,s) :=
       block(
         [n:hipow(p,s),liste],
         liste:makelist(i,i,0,n),
         p:expand(p),
         map(lambda([u],coeff(p,s,u)),liste)
       )$
```

Ermittlung der Polynomkoeffizienten

```
(%i5)  coefficient_list(p,s);
(%o5)  [ 3 , 3 , 5 , 6 , 5 , 2 , 10 ]
```

Definition einer Funktion, die Nullstellen eines Polynoms in einer Liste abspeichert.

```
(%i6)  zeros(f):=
       block(
         [polyfactor:false,liste],
         liste:allroots(expand(num(f))),
         map(rhs,liste)
       )$
```

Ermittlung der Nullstellen eines Polynoms

```
(%i7)  zeros(p);
(%o7)  [ 0.754 %i - 0.0133 , - 0.754 %i - 0.0133 , 0.381
%i - 0.669 , - 0.381 %i - 0.669 , 0.744 %i + 0.582 ,
0.582 - 0.744 %i ]
```

Definition einer Funktion, die die Laplace-Rücktransformation einer gebrochen rationalen Funktion in *s* mit Hilfe der numerisch ermittelten Nennernullstellen berechnet.

```
(%i8)  nilt(f,s,t):=
       block(
         [polyfactor:true,ratprint:false,ft,den],
         den:allroots(float(denom(f))),
         ft:ilt(num(f)/den,s,t),
         ev(ft,float,expand)
       )$
```

Die in Maxima eingebaute Funktion *ilt* zur Laplace-Rücktransformation versagt bereits bei Nennerpolynomen dritten Grades.

```
(%i9)  ilt(1/(s^3+2*s**2+3*s+1),s,t);
```

$$(\%o9)\quad \mathrm{ilt}\left(\frac{1}{s^3+2\,s^2+3\,s+1},\,s\,,\,t\right)$$

Für die selbst definierte Funktion *nilt* ist die Rücktransformation beliebiger gebrochen rationaler Funktionen kein Problem (sofern die Polynomkoeffizienten reine Zahlenwerte sind).

```
(%i10)  nilt(1/(s^3+2*s**2+3*s+1),s,t);
(%o10)  - 0.148 %e^- 0.785 t  sin( 1.3071 t )- 0.545
%e^- 0.785 t  cos( 1.3071 t )+ 0.545 %e^- 0.43 t
```

Ein Hash mit drei Schlüssel-Wert-Paaren

```
(%i11)
options:['color=blue,'height=70, 'width=100];
(%o11)  [ color = blue , height = 70 , width = 100 ]
```

Definition einer Funktion, die prüft, ob in einem Hash ein bestimmter Schlüssel existiert.

```
(%i12)  option_exists(o,l) :=
        if member(o,map(first,l)) then true$
```

Herauslesen eines zu einem Schlüssel gehörigen Wertes erfolgt mit der Funktion *assoc*.

```
(%i13)  assoc(color,options);
(%o13)  blue
```

Definition einer Funktion, die ein Element (Schlüssel-Wert-Paar) aus einem Hash löscht.

```
(%i14)  delete_option(o,l) :=
         if option_exists(o,l) then
         sublist(l,lambda([u],first(u) # o)) else l$
```

Definition einer Funktion zum Erzeugen oder Ändern eines Elements in einem Hash.

```
(%i15)  set_option(o,l) :=
        block([li],
          li:if option_exists(first(o),l) then
                delete_option(first(o),l)
            else l,
          endcons(o,li)
        )$
```

Ein neuer Hash-Eintrag wird erzeugt.

```
(%i16)  options:set_option(title="Test",options);
(%o16)  [ color = blue , height = 70 , width = 100 ,
title = Test ]
```

Ein existierender Hash-Eintrag wird geändert.

```
(%i17)  options:set_option(color=red,options);
(%o17)  [ height = 70 , width = 100 , title = Test ,
color = red ]
```

6.2 Files und Directories für Pakete

Eine Zusammenstellung von von Anweisungen, in einem File abgespeichert, wird im Folgenden als *Paket* bezeichnet. Eine Vielzahl von Paketen steht standardmäßig zur Verfügung, mittels eigener Pakete kann die Funktionalität von Maxima beliebig erweitert werden.

In erster Linie wird ein Paket Funktionsdefinitionen enthalten, sinnvoll kann aber auch die Zuweisung globaler Variablen sein, etwa von Defaultwerten oder von benötigten Naturkonstanten.

Funktionen haben *keine* Namensräume; das heißt, ein Funktionsname und die Verwendung einer Funktion sind weder auf einen bestimmten Codebereich noch auf einen bestimmten Teil des Programmablaufes beschränkt; Funktionsnamen sind immer global. Um Namenskonflikte zu vermeiden, sollten daher bei der Definition von Hilfsfunktionen, die nicht für den direkten Aufruf durch den Benutzer gedacht sind, und die daher auch wahrscheinlich nicht dokumentiert sein werden, möglichst keine „gängigen“ Namen verwendet werden; beispielsweise kann ein solcher Name mit einem Unterstrich „_“ beginnen.

Pakete werden üblicherweise mit einem Texteditor erstellt und als File mit der Erweiterung `.mac` abgespeichert.

Ein File mit der Erweiterunng `.wxm` enthält eine abgespeicherte wxMaxima-Arbeitssitzung. Neben den eingegebenen Ausdrücken und als Kommentare eingegebenen Texten enthält es zusätzlich Kommentare, die die grafische Oberfläche von wxMaxima beim Wiedereinlesen steuern. Nachträgliches Editieren eines wxm-Files (zum Ändern, Löschen oder Hinzufügen von Ausdrücken) ist möglich, die Struktur dieser Kommentare muss dabei aber erhalten bleiben.

Abspeichern und Wiedereinlesen einer wxMaxima-Arbeitssitzung erfolgt nicht mit einem *Befehl* in der Eingabezeile, sondern über das Menü *<Datei><Speichern>* bzw. *<Datei><Öffnen>*.

Filetypen:	
name`.mac, `*name*`.mc`	enthält Anweisungen und Funktionsdefinitionen in Maxima- oder Lisp-Code, der beim Laden ausgeführt wird.
name`.lisp`	File mit Lisp-Anweisungen, die beim Laden ausgeführt werden.
name`.wxm`	Abgespeicherte wxMaxima-Arbeitssitzung
Variable für Directories:	
`file_search_maxima`	Liste mit Verzeichnissen (als Strings), die von Maxima beim Laden von Paketen durchsucht werden
`maxima_userdir`	Benutzerverzeichnis für eigene Pakete

Files und Directories

In der Variable `file_search_maxima` ist der Suchpfad von Maxima in Form einer Liste festgelegt, Hinzufügen von Verzeichnissen kann mit der Funktion `cons` (Abschnitt 2.4) erfolgen. Das Maxima-Benutzerverzeichnis `maxima_userdir` wird zwar beim Laden nach Paketen durchsucht, ist aber nicht das Standardverzeichnis für *Ausgaben*. Bei Ausgaben ist es daher sinnvoll, bei der Angabe des Filenamens immer den gesamten Pfadnamen anzugeben.

Die Einträge können neben Verzeichnisnamen auch *Muster* für Filetypen haben, typischerweise

in folgender Form:

Directory/###.{mac,mc}

Definieren eines Maxima-Benutzerverzeichnisses

```
(%i18) maxima_userdir:"d:/w/maxima/###.{mac}";
(%o18) d:/w/maxima/###.{mac}
```

Hinzufügen des Benutzerverzeichnisses zum Maxima-Suchpfad. Dies erfolgt sinnvollerweise im Initialisierungsfile `maxima-init.mac`.

```
(%i19) file_search_maxima:
         cons(maxima_userdir,file_search_maxima);
(%o19) [ d:/w/maxima/###.{mac} ,
```

Laden eines Pakets, das im Benutzerverzeichnis liegt.

```
(%i20) load("coma");
coma v.1.02, (Wilhelm Haager, 21.4.2008)
(%o20) d:/w/maxima/coma.mac
```

6.3 Laden und Abspeichern

`load` ist der Standardbefehl zum Laden von Paketen (mac-Files und lisp-Files). Existiert im Maxima-Suchpfad ein File mit dem Namen `maxima-init.mac` oder `maxima-init.lisp`, so wird es beim Start von Maxima automatisch geladen. Dieses File enthält typischerweise Initialisierungsbefehle, Anweisungen zum Laden weiterer Pakete und Befehle zum Erweitern des Maxima-Suchpfades.

Mit `batch` werden Anweisungen von einem File genau so eingelesen und verarbeitet, als würden sie händisch in der Eingabezeile eingegeben. Jede Eingabe und jedes Ergebnis wird dabei am Bildschirm ausgegeben. Beliebige Filetypen sind zulässig, auch wxm-Files können mit `batch` eingelesen werden. Im Gegensatz zum Laden über das Menü *<Datei><Öffnen>* wird dabei *keine* neue wxMaxima-Sitzung gestartet, alle vorherigen Berechnungen bleiben gespeichert.

Wird beim Abspeichern von Ausdrücken nur ein Filename und nicht der gesamte Pfadname angegeben, wird das File im wxMaxima-Installationsverzeichnis abgespeichert.

`grind` und `tex` liefern die Ergebnisse nicht als *Funktionswert* zurück (diese sind `done` bzw. `false`), sondern haben die entsprechende Ausgabe als *Nebeneffekt*.

Standardmäßig wird bei jedem Schreibbefehl ein bestehendes Ausgabefile neu überschrieben. Hat die globale Variable `file_output_append` den Wert `true`, so wird jede Ausgabe an ein bestehendes File angehängt.

`stringout` schreibt die entsprechenden Ausdrücke als Strings in jener Form in das Ausgabefile, in der sie mit `load` wieder eingelesen werden könnten. Ist die globale Variable `grind` auf `true` gesetzt, so ist die Ausgabe mit Einrückungen, Leerzeichen und Zeilenvorschüben formatiert, anderenfalls erfolgt die Ausgabe unformatiert, eine (beliebig lange) Zeile ohne Leerzeichen pro Ausdruck.

Manche Anweisungen liefern keinen *Ergebniswert*, der (im funktionalen Stil) weiterverwendet wird, sondern produzieren eine Ausgabe als *Nebeneffekt*; dies sind beispielsweise die Anweisungen *disp*, *print*, *grind* oder *tex*. Mit `with_stdout` wird eine solche Ausgabe in ein File umgeleitet. Zu beachten ist, dass `with_stdout` nicht auf beliebige Ausdrücke sinnvoll anwendbar ist, sondern nur auf Anweisungen, die eine Ausgabe (als Nebeneffekt) produzieren.

Wird `with_stdout` in Verbindung mit der Funktion `grind`(*f1*, *f2*, ...) angewandt, so können Funktionsdefinitionen *f1*, *f2*, ... in ein File abgespeichert werden, das mit `load` (in einer anderen Maxima-Arbeitssitzung) als Paket eingelesen werden kann.

`load(`*filename*`)`	Einlesen und Ausführen von Befehlen aus der Datei *filename* ohne Bildschirmausgabe
`batch(`*filename*`)`	Einlesen und Ausführen von Befehlen aus der Datei *filename* mit Bildschirmausgabe
`grind(`*expr1*`,`*expr2*`,...)`	Ausgabe der Ausdrücke *expr1*, *expr2*, ... in der *Eingabeform*, also in einer von Maxima lesbaren Form
`tex(`*expr*`)`	Ausgabe des Ausdrucks *expr* in *TEX-Form*
`file_output_append`	logische Variable, steuert die Ausgabe in ein File: `false` ... Neuschreiben des Files (Default) `true` ... Anhängen an bestehendes File
`grind`	logische Variable, steuert die Formatierung der Ausgabe als String: `false` ... keine Formatierung `true` ... Formatierung mit Einrückungen
`save(`*filename*`,`*expr1*`,`*expr2*`,...)`	Ausgabe der Ausdrücke *expr1*, *expr2*, ... in die Datei *filename* als Lisp-Code
`stringout(`*filename*`,`*expr1*`,`*expr2*`,...)`	Ausgabe der Ausdrücke *expr1*, *expr2*, ... in die Datei *filename* als Strings
`stringout(`*filename*`,functions)`	Schreiben aller benutzerdefinierten Funktionen in das File *filename*
`stringout(`*filename*`,values)`	Schreiben aller Benutzereingaben in das File *filename*
`stringout(`*filename*`,[`*m*`,`*n*`])`	Schreiben aller mit den Labels *m*...*n* markierten Ein- und Ausgaben in das File *filename*
`with_stdout(`*filename*`,`*expr1*`,`*expr2*`,...)`	Schreiben des von den Ausdrücken *expr1*, *expr2*, ... generierten Outputs in die Datei *filename*

Laden und Abspeichern von Maxima-Code

Wird `with_stdout` in Verbindung mit der Funktion `tex` angewandt, werden TEX-Ausdrücke in ein File abgespeichert (der gleiche Effekt könnte auch mit <*copy*>-<*paste*> erreicht werden).

Die Ausgabe von Ausdrücken als Strings soll formatiert (mit Einrückungen und Zeilenvorschüben) erfolgen.

```
(%i21)  grind:true;
(%o21)  true
```

Ausgabe der Definition der Funktion *ranpoly* in ein File.

```
(%i22)
with_stdout("d:/w/maxima/myfunc.mac", grind(ranpoly));
(%o22)  done
```

Alle weiteren Ausgaben in Files sollen ein bestehendes File nicht überschreiben, sondern an das Ende angefügt werden.

```
(%i23)  file_output_append:true;
(%o23)  true
```

Ausgabe einer weiteren Funktionsdefinition (an das Ende eines bestehenden Files).
Beachte: Die Ausgabe erfolgt als *Nebeneffekt*, das *Ergebnis* der Funktion ist `done`.

```
(%i24) with_stdout("d:/w/maxima/myfunc.mac",
             grind(coefficient_list));
(%o24) done
```

Einlesen der Funktionsdefinitionen mit `batch`

```
(%i25) batch("d:/w/maxima/myfunc.mac");
batching #pd:/w/maxima/myfunc.mac
```

Alle eingelesenen Ausdrücke werden beim Einlesen angezeigt und erhalten Input-Labels.

```
(%i26) ranpoly(n):= block ( [ p : 0 ] , for  i  from
0  thru  n  do  p :(random(10)+ 1) ' s^i + p , p )
(%i27) coefficient_list(p , s):= block ( [ n :
hipow(p , s)] , p : expand(p ),
map( lambda( [ u ] , coeff(p , s , u )), makelist(i , i , 0 , n))
)
```

Beim Einlesen mit `load` werden die eingelesenen Ausdrücke *nicht* angezeigt.

```
(%i28) load("d:/w/maxima/myfunc.mac");
(%o28) d:/w/maxima/myfunc.mac
```

Eine Gleichung mit den selben Ausdrücken auf der linken und rechten Seite. Auf der linken Seite wird die Auswertung aber durch *Quotieren* verhindert.

```
(%i29)
g:'integrate(sin(omega*t)*exp(-s*t),t,0,inf)=
   integrate(sin(omega*t)*exp(-s*t),t,0,inf);
```

$$(\%o29)\quad \int_0^{\infty} \%e^{-s\,t}\,\sin(\mathrm{omega}\,t)\,dt = \frac{\mathrm{omega}}{s^2+\mathrm{omega}^2}$$

Ausgabe der Gleichung in Eingabeform (als Nebeneffekt!)

```
(%i30) grind(g);
'integrate(%e^-(s*t)*sin(omega*t),t,0,inf) = omega/(s^2+o
(%o30) done
```

Ausgabe der Gleichung in TEX-Form

```
(%i31) tex(g);
$$\int_{0}^{\infty }{e^ {- s\,t }\,\sin \left(\omega\,t\r
 {{\omega}\over{s^2+\omega^2}}$$
(%o31) false
```

Umleiten der Ausgabe in TEX-Form in ein File

```
(%i32) with_stdout("d:/w/maxima/test.tex",tex(g));
(%o32) false
```

TEX liefert für obigen Ausdruck folgendes perfekt gesetzte Ergebnis:

$$\int_0^{\infty} e^{-s\,t}\,\sin\left(\omega\,t\right)\,dt = \frac{\omega}{s^2+\omega^2}$$