Kurzeinführung in die CAS Maple, MuPAD und Reduce

Begleitmaterial zum Praktikum "Symbolisches Rechnen" Wintersemester 1998/99

H.-G. Gräbe, Institut für Informatik,
`http://www.informatik.uni-leipzig.de/~graebe`

9. Oktober 1998

Inhaltsverzeichnis

1 Die Struktur eines Computeralgebra-Systems

Für diese Kurzbeschreibung wollen wir mit einer gegenüber der Vorlesung vereinfachten Struktur eines CAS arbeiten und in einem solchen die *Oberfläche* (front end), den *Systemkern* (kernel) sowie das in entsprechenden *Bibliotheken* (libraries) implementierte mathematische Wissen unterscheiden. Hier sollen zunächst einige Aspekte dieser drei Einheiten, die allen CAS gemeinsam sind, besprochen werden, ehe wir uns an Hand eines solchen Leitfadens den einzelnen Systemen zuwenden.

1.1 Die Oberfläche

Die meisten Systeme werden mit zwei verschiedene Versionen, einer text-orientierten Kommandozeilen-Version und einer fensterorientierten *integrierten Oberfläche* ausgeliefert. Letztere bietet neben den generellen Vorteilen einer integrierten Entwicklungsumgebung wie etwa einem eigenen Editor die Möglichkeit der extensiven Formatierung von Ausgaben bis hin zu einer buchdruckähnlichen Qualität, hat aber den Nachteil, das man ihre Bedienung erlernen muß.

Wir werden uns deshalb mit der weniger umfangreichen, dadurch aber überschaubarere Funktionalität bietenden Kommandozeilenversion begnügen. In der im Pool getroffenen Installation wird dafür durch die Angabe des jeweiligen Systemnamens `Maple, Mupad` oder `Reduce` das entsprechende CAS in einem extra Fenster, dem *Rechenfenster* gestartet und mit dem Befehl

> `quit;`

wieder verlassen, womit auch automatisch das Rechenfenster geschlossen wird.

Da auch die integrierte Oberfläche nur eindimensionale *Eingaben* (wie fast alle Programme, als Zeichen von einem Eingabestrom) zuläßt, die allerdings stark an der üblichen mathematischen Notation sowie den Variablen- und Steuerstrukturen einer imperativen Programmiersprache orientiert ist, stellt das keine wesentliche Einschränkung dar (wenn man einmal davon absieht, daß in diesem Modus meist keine Graphiken dargestellt werden können).

Zusammen mit einem zweiten Fenster, dem *Textfenster*, in dem der von Ihnen favorisierte Editor läuft, kann auf diese Weise die Ein- und Ausgabe gut verwaltet werden.

Die *Ausgabe* der Ergebnisse der Rechnungen erfolgt normalerweise in einem zweidimensionalen Ausgabeformat, das der üblichen Formatierung mathematischer Ausdrücke nahekommt. Das sieht zwar recht schön aus, ist aber für eine zeichenorientierte Bearbeitung der Ausgabe wie etwa die Übertragung in ein anderes Fenster oder Redirektion in den Eingabestrom nicht geeignet. Jedes der Systeme bietet deshalb die Möglichkeit der unformatierten Ausgabe.

Ein wichtiger Teil der Oberfläche ist das *Hilfesystem*, mit dem sich unkompliziert Informationen zu den verschiedenen Sprachkonstrukten und Funktionen des CAS gewinnen lassen. Ein hypertextbasiertes Hilfesystem ist inzwischen Teil der integrierten Oberflächen aller großen CAS. Maple und MuPAD besitzen darüberhinaus auch in der Kommandozeilenversion ein internes Hilfesystem, das durch

> `?`< *topic* >

aktiviert wird.

Daneben gibt es für jedes der Systeme Dokumentationen auf dem Netz. Hierzu sei auf die folgenden Materialien verwiesen, die Sie auf der Web-Seite zum Praktikum finden.

Maple : [9] und [4]
MuPAD : [8], [7] sowie die umfangreicheren Materialien [2], [1], [6]
Reduce : [5] und [3]

1.2 Der Systemkern

Die großen CAS sind durchgehend Interpreter-Systeme, die in einer Interpreterschleife (Eingabe - Verarbeitung - Ausgabe) Zugriff auf das in ihnen gespeicherte mathematische Wissen erlauben. Eine *Eingabe* besteht dabei in der Regel aus einem über eine oder mehrere Zeilen reichenden (syntaktisch korrekten) Ausdruck, der auf Konstrukte der *internen Programmiersprache*, bereits vereinbarte *Variablen* und automatisch oder explizit aus Bibliotheken geladene *mathematische Funktionen* zurückgreift. Eine solche Eingabe schließt mit einem *Terminationssymbol* ab, nach dem mit einem neuen $< Enter >$ die Evaluation angestoßen wird.

Da symbolische Rechnungen sehr umfangreiche Ausgaben erzeugen können, stehen zwei *Terminationssysmbole* (mit/ohne Ausgabe des Ergebnisses) zur Verfügung. Weitere Sonderzeichen sind Kommentarzeichen.

Eine wichtige Besonderheit symbolverarbeitender Systeme ist die Tatsache, daß Variablen symbolische Werte haben können (und am Anfang auch haben; sie werten zu sich selbst aus). Damit überlappen sich der Namensraum des CAS und der Wertebereich der so benennbaren Variablen. Mehr noch, auch die Namen von Funktionen sind symbolische Ausdrücke und gehören diesem Namensbereich an, wobei noch zwischen "richtigen" Funktionen, d.h. *Funktionsaufrufen* wie etwa $max(3, 1, 4, 5)$, das 5 zurückliefert und *Funktionssysmbolen* wie etwa $sqrt(2)$, das für das Symbol $\sqrt{2}$ steht, zu unterscheiden ist. Da andererseits die Grenzen fließend sind (so ist z.B. $max(a, b)$ mit Symbolen a, b nicht weiter zu vereinfachen, muß also ebenso symbolisch bleiben wie $sqrt(2)$), verwundert es nicht, daß Symbole, Variablennamen und Funktionsnamen von den meisten CAS einheitlich behandelt werden.

Dies ist sogar eines der Grundprinzipien der Sprache LISP, in der jede Variable eine Liste von Eigenschaften zugeordnet bekommt, von denen (u.a.) eine ihr Wert und eine andere der aufzurufende Funktionskörper sein kann. In Maple und MuPAD wird das besonders deutlich durch die Art der Definition von Prozeduren (s.u.).

Wichtige *mathematische Konstanten* stehen mit ihren gebräuchlichen Namen in den Systemen zur Verfügung, was insbesondere für die Eulersche Zahl e und die imaginäre Einheit i Namenskonflikte hervorrufen kann. Deshalb verwenden Maple und MuPAD eine spezielle Mischung von Groß- und Kleinschreibung für diese Symbole. Die vordefinierten mathematischen Konstanten sind meist vor Überschreiben geschützt.

Die Überlappung der Namensräume zwischen Variablen und Symbolen führt bei längeren Sitzungen oft zu Inkonsistenzen, wenn man etwa die Variable x, der man irgendwann einen Wert zugewiesen hat, später als Symbol in einem Ausdruck verwenden möchte. Neben der Möglichkeit, alle Variablen zurückzusetzen, die hier nicht besprochen werden soll (man kann sie durch einen Programmneustart ebenso leicht erreichen), gibt es deshalb Funktionen, mit denen man einzelne Variablen "löschen", d.h. in ihren Ausgangszustand versetzen kann.

Daneben spielt die Möglichkeit, Symbole in Ausdrücken als Variable zu betrachten und ihnen *lokal* gewisse Werte zuzuweisen, eine wichtige Rolle. Das ist über einen *Substitutionsoperator* möglich.

	Maple	MuPAD	Reduce
Termination mit/ohne Ausgabe	; und :	; und :	; und $
Kommentar	# (bis Zeilenende)	# (Kommentar) #	% (bis Zeilenende)
Variable x zurücksetzen	`x:='x';`	`unassign(x);`	`clear x;`
Wichtige math. Konst.	exp(1), Pi, I	E, PI, I	e, pi, i
Substitutionsoperator	`subs(x=a,p(x));`	`subs(p(x),x=a);`	`subst(x=a,p(x));`
unformatierte Ausgabe	`lprint(`$< Expr >$`)`	`PRETTY_PRINT:=FALSE;`	`off nat;`

Tabelle 1 : Zusammenfassende Darstellung.

Daneben gibt es gewöhnlich noch Zeichen(kombinationen), mit denen man auf frühere Ein- oder Ausgaben zurückgreifen kann. Diese Möglichkeiten sollen hier nicht betrachtet werden, da man die entsprechende Funktionalität durch die Verwendung von Variablen bzw. Editierfunktionen erreichen kann.

1.3 Wichtige Daten- und Steuerstrukturen

In Symbolverarbeitungssystemen auftretende Datenstrukturen sind in der Regel dynamischer Natur. Diese beginnen bei Zahlbereichen, die mit Zahlen beliebiger ($\mathbf{Z}, \mathbf{Q}$) oder vom Nutzer einstellbarer ($\mathbf{R}$) Genauigkeit arbeiten, setzen sich fort über symbolische Ausdrücke, die in Form von Parsebäumen dargestellt sind und reichen bis zur Darstellung von Ergebnissen mathematischer Berechnungen, die als Matrizen, Listen oder Mengen erst zur Laufzeit bestimmbarer Größe dynamisch erzeugt werden müssen. Ein entsprechendes für den Nutzer transparentes Speichermanagement kann nur dann effektiv gestaltet werden, wenn es intern auf eine möglichst geringe Zahl von Datentypen zurückgreift.

Ein solches universelles Element zur Darstellung symbolischer Strukturen sind (geschachtelte) *Listen*. Dies ist zunächst klar für Funktions*symbole*. So kann man etwa

```
> f(a,b,c);
```

intern als Liste `(f a b c)` darstellen, da Name und Argumente das Funktionssymbol eindeutig beschreiben. Aber auch Ausdrücke der Form

```
> a+b+c;
```

die intern sofort in Präfixnotation `plus(a,b,c)` umgesetzt werden, kann man auf diese Art speichern. Für kompliziertere Ausdrücke wie etwa

```
> (a+b)*(c+d);
```

ergeben sich dann geschachtelte Listen `(mult (plus a b) (plus c d))`. Auch Strukturen, die nach außen hin eine Sequenz von Elementen darstellen, wie etwa Listen oder Mengen, lassen sich auf diese Weise durch (interne) Listen darstellen. So ist etwa die Liste

```
> [a,b,c];
```

als `(list a b c)`, dagegen die Menge

```
> {a,b,c};
```

als `(set a b c)` darstellbar. Die Elemente einer solchen internen Liste werden aus Konsistenzgründen ab 0 numeriert. Da das Element 0 der Liste eine besondere Information (neben obigen ist dort oft eine Typinformation gespeichert) darstellt, wird es gewöhnlich anders als die restlichen Listenelemente angesprochen.

Mit der zentralen Stellung von Listen benötigt ein CAS auch Sprachkonstrukte zu deren Erzeugung und Manipulation. Kennt man die Länge `nops(l)` einer Liste l und kann auf einzelne Listenelemente `op(l,i)` zugreifen, so kann man hierfür etwa eine `for`-Schleife

```
> for i from 1 to nops(l) do ...  something with op(l,i) ...
```

verwenden. Aus naheliegenden Effizienzgründen stellen die meisten CAS eine spezielle Listentraversion der Form

```
> for x in l do ...  something with x ...
```

zur Verfügung. Ähnliche Effekte kann man mit funktionalen Programmierelementen wie etwa der Funktion `map` erreichen, die Listen in Listen transformiert.

Zur Erzeugung von Listen gibt es weiterhin einen *Sequenzierungsoperator*, der eine Liste aus einzelnen Elementen nach einer Bildungsvorschrift generiert. In Maple und MuPAD wird dafür ein eigener Operator verwendet, in Reduce dagegen die Syntax von `for` dahingehend erweitert, daß sie einen Wert zurückgibt.

Die *Steuerstrukturen* zur Steuerung des Programmablaufs entsprechen, mit der genannten Erweiterung der `for`-Anweisung auf Listentraversion, denen klassicher imperativer Programmiersprachen. Es stehen Verzweigungen, Case-Anweisungen und Schleifen in (wie auch in klassischen Sprachen) von System zu System variierender Syntax zur Verfügung. Eine kurze Beschreibung der Syntax der wichtigsten Steuerstrukturen finden Sie unten in den Kapiteln zu den jeweiligen Systemen, eine ausführliche Beschreibung in deren Dokumentation.

Ebenso kann man einzelne Programmteile in *Unterprogrammen* oder *Funktionen* zusammenfassen. Da alle Datentypen dynamischer Natur sind, unterliegt der Rückgabewert einer Funktion keinen Restriktionen, so daß, ähnlich wie in C und im Unterschied zu Pascal, nicht zwischen Prozeduren und Funktionen unterschieden wird. Da die Speicherverwaltung für den Nutzer transparent ist, gibt es auch keine Referenzparameter. Aufrufparameter werden in der Regel, wie in funktionalen Programmiersprachen üblich, durch *call by value* initialisiert.

1.4 Die mathematischen Bibliotheken

CAS bestehen normalerweise aus einem relativ kleinen Kern, zu dem während des Programmstarts ein gewisses Minimum an mathematischem Wissen hinzugeladen wird. Auf diese Weise wird die im Hauptspeicher zu haltende Programmgröße minimiert, zumal für die meisten Anwendungen nur wenige der Pakete gleichzeitig zum Einsatz kommen. Deshalb müssen für gewisse Belange explizit Pakete nachgeladen oder, insoweit das Nachladen automatisch geschieht, in den globalen Namensraum exportiert werden. Dies gilt in Maple und MuPAD insbesondere für die Pakete zur Matrizenmanipulation.

Maple	`with(`*package*`):`
MuPAD	`export(`*package*`):`
Reduce	`load` *package*`;`

Tabelle 2 : Wie man ein Paket lädt.

Das mathematische Wissen zum *Lösen* gewisser Sachverhalte, d.h. Umwandlung eines impliziten in einen expliziten Zusammenhang, ist gewöhnlich in einem oder mehreren `solve`-Operatoren zusammengefaßt. Dies bezieht sich insbesondere auf das Lösen von polynomialen Gleichungen und polynomialen Gleichungssystemen. Die allgemeine Struktur dieses Operators ist meist

> `solve(`$<Gleichungen>$`,`$<Variablen>$`);`

wobei als erstes Argument einzelne Polynome, explizite Gleichungen etwa der Form $p(x) = 0$ oder Listen bzw. Mengen von Gleichungen angegeben werden können, die bzgl. einer als zweites Argument anzugebenden Variablen bzw. Liste oder Menge von Variablen aufzulösen sind. Neben polynomialen Gleichungssystemen werden wir in den Tutorien die Fähigkeit zum Lösen trigonometrischer Gleichungen testen.

Läßt sich die Gleichung oder das Gleichungssystem (durch das gegebene CAS erfolgreich) in eine explizite Form überführen, so erhält man als Ergebnis eine Menge oder Liste dieser Lösungen, deren Darstellung sich insbesondere für unendliche Lösungsmengen stark unterscheidet. Ist eine solche explizite Lösung nicht möglich oder nicht sinnvoll, so wird als Ergebnis das System in möglicherweise vereinfachter impliziter Form zurückgegeben.

Letzteres bezieht sich insbesondere auf die Darstellung der Nullstellen von Polynomen. Bekanntlich kann man die Nullstellen eines allgemeinen Polynoms vom Grad > 4 nicht mehr durch Radikale ausdrücken. Diese besitzen damit keine exakte Darstellung durch evtl. geschachtelte Wurzelsymbole. Da auch die Darstellung der Nullstellen von Polynomen dritten und vierten Grades durch Radikale sehr kompliziert und für weitere Rechnungen mit ihnen wenig geeignet ist, verwenden die CAS zur Darstellung der Nullstellen des (irreduziblen) Polynoms $p(x)$ die Repräsentation durch ein Funktionssymbol $RootOf(p(x), x)$. Eine solche auf den ersten Blick ungewohnte Darstellung unterscheidet sich nur wenig von der Angabe der expliziten Lösung der Gleichung $x^2 = 2$

als $\{\sqrt{2}, -\sqrt{2}\}$, da ja $\sqrt{2}$ auch nur ein Symbol ist, von dem nicht mehr bekannt ist, als daß sein Quadrat zur Zahl 2 vereinfacht werden kann.

Allerdings sind die CAS unterschiedlich umfassend in der Lage, diese symbolische Information auch auszuwerten. Hinter jedem solchen $RootOf$-Symbol verbergen sich nämlich *mehrere* Lösungen, die man allein durch das $RootOf$-Symbol nicht unterscheiden kann. Die konsequenteste Lösung hierfür hat Mathematica 3.0 vorgelegt, indem statt eines Symbols $RootOf(p(x), x)$ ein Feld von $deg\ p$ Symbolen zur Bezeichnung der unterschiedlichen Nullstellen angelegt wird, mit denen dann individuell gerechnet werden kann. Die anderen CAS verfolgen unterschiedliche Konzepte zur Behandlung dieser Ambiguität, die jedoch oft nicht zu überzeugenden Lösungen führen.

Noch komplizierter wird die Darstellung der Lösungsmenge von trigonometrischen Gleichungen, die meist aus mehreren abzählbar unendlichen Serien von Zahlen bestehen. Die damit auftretenden Probleme werden in einem eigenen Tutorium zu studieren sein.

In den nächsten Kapiteln werden nun zunächst einzelne Sprachelemente der verschiedenen Systeme näher vorgestellt. Dies geschieht ausführlich am Beispiel von MuPAD, während für die anderen Systeme nur noch spezifische Lösungen genauer besprochen werden.

2 MuPAD

Zur praktischen Einführung wollen wir mit einer kommentierten MuPAD-Sitzung beginnen, um die wichtigsten Sprachelemente in Aktion zu zeigen.

2.1 Rechnen mit exakten Zahldarstellungen

```
> 1/5 + 1/2; 2/3*(1/4+3/5); (1/2+1/3)^(-1);
```

$$7/10$$

$$17/30$$

$$6/5$$

```
> v:=1/2+E^2/6; float(v); float(v-sqrt(3));
```

$$\frac{e^2}{6} + 1/2$$

$$1.731509349$$

$$-0.0005414577471$$

```
> sqrt(4); sqrt(6); sqrt(12);
```

$$2$$

$$\sqrt{6}$$

$$2\sqrt{3}$$

Alle Rechnungen werden exakt ausgeführt, Symbole, die nur durch Eigenschaften exakt beschreibbar sind ("e ist der Grenzwert der Reihe ... "), bleiben symbolisch, wobei für die gängigen mathematischen Konstanten oft spezielle Schreibweisen (hier Groß- statt Kleinschreibung; MuPAD unterscheidet das) eingeführt werden, um nicht versehentlich Variablen mit demselben Namen zu verwenden.

Zu jedem solchen symbolischen Ausdruck kann, wenn definiert, ein Näherungswert (beliebig vorgebbarer Genauigkeit) berechnet werden, wozu etwa das mit der Variablen E verbundene Verfahren zu deren genauer Berechnung aufgerufen wird. Dazu ist die Systemkonstante $DIGITS$ entsprechend zu modifizieren. `DIGITS:=NIL` weist ihr (wieder) den Defaultwert 10 zu.

```
> DIGITS:=55: float(sqrt(2)); DIGITS:=NIL:
```

$$1.414213562373095048801688724209698078569671875376948073$$

Daneben gibt es eine Reihe von Funktionen für das Rechnen mit ganzen Zahlen.

```
> igcd(105,315,2898,7368172368127368127368127327); # gcd-Berechnung #
```

$$7$$

```
> u:=333!; # Fakultät, auch fact(333) #
```

```
1033446543458805915609396553829751655062226004168206282343290246\
9783188597914276568552700194849877929894375950252570477080418352\
7325976587456659256047046692271337264772438543178366351306941238\
9371163853300198049622987566547659856882180617030376554048981440\
2234159901540440432134155844542962445153646330595588291605924429\
2113522799434713728172799387209748952603877845782391509318169467\
8641623251666625196542191965183804461805099129440354695893074541\
9743836966520198735201123255884089263272829846640538826979843642\
8857757916415751091787535095800016603920923967986489243754010241\
4788370229814591004688940288039419536998400000000000000000000000\
0000000000000000000000000000000000000000000000000000000000
```

```
> ifactor(u); # Liste der Faktoren mit Vielfachheiten #
```

$$
\begin{aligned}
&[1, 2, 328, 3, 165, 5, 81, 7, 53, 11, 32, 13, 26, 17, 20, 19, 17,\\
&23, 14, 29, 11, 31, 10, 37, 9, 41, 8, 43, 7, 47, 7, 53, 6, 59, 5,\\
&61, 5, 67, 4, 71, 4, 73, 4, 79, 4, 83, 4, 89, 3, 97, 3, 101, 3,\\
&103, 3, 107, 3, 109, 3, 113, 2, 127, 2, 131, 2, 137, 2, 139, 2,\\
&149, 2, 151, 2, 157, 2, 163, 2, 167, 1, 173, 1, 179, 1, 181, 1,\\
&191, 1, 193, 1, 197, 1, 199, 1, 211, 1, 223, 1, 227, 1, 229, 1,\\
&233, 1, 239, 1, 241, 1, 251, 1, 257, 1, 263, 1, 269, 1, 271, 1,\\
&277, 1, 281, 1, 283, 1, 293, 1, 307, 1, 311, 1, 313, 1, 317, 1,\\
&331, 1]
\end{aligned}
$$

```
> u:=fact(40); v:=nextprime(u); # nächstgelegene Primzahl berechnen #
```

$$815915283247897734345611269596115894272000000000$$

$$815915283247897734345611269596115894272000000053$$

```
> isprime(v); # Primzahltest #
```

$$TRUE$$

Dies ist natürlich nur eine Auswahl des in Form von Funktionen zur Verfügung gestellten (algorithmisierten) mathematischen Wissens über ganze Zahlen, über das MuPAD verfügt. Für einen kompletteren Überblick sei auf die Systemdokumentation verwiesen.

2.2 Rechnen mit rationalen Ausdrücken

Auf dieselbe Weise wie rationale Zahlen können auch allgemeinere symbolische Ausdrücke manipuliert werden. Dies sei hier zunächst an rationalen Ausdrücken, die nur freie Variablen enthalten, demonstriert.

```
>  p:= x^2+2*x; expand(p^2);
```

$$2\,x + x^2$$

$$4\,x^2 + 4\,x^3 + x^4$$

```
>  q:=(a+b)^10; expand(q);
```

$$(a+b)^{10}$$

$$a^{10}+b^{10}+10ab^9+10a^9b+45a^2b^8+120a^3b^7+210a^4b^6$$
$$+252a^5b^5+210a^6b^4+120a^7b^3+45a^8b^2$$

Oftmals verbergen sich hinter kompliziert aussehenden rationalen Ausdrücken einfachere, die durch *Normalisierung* der Darstellung, d.h. Überführung in eine Form $\frac{p(x)}{q(x)}$ mit teilerfremden $p(x), q(x)$ gefunden werden können.

```
> d1:=(a^2-b^2-c^2+2*b*c)/((a+b-c)/(a+b+c));
```

$$\frac{(a+b+c)\left(2bc+a^2-b^2-c^2\right)}{a+b-c}$$

```
> normal(d1);
```

$$2ac+a^2-b^2+c^2$$

```
> d2:=(a^2-b^2)/(a-b) - (a^3-b^3)/(a^2-b^2);
```

$$\frac{a^2-b^2}{a-b}-\frac{a^3-b^3}{a^2-b^2}$$

```
> normal(d2);
```

$$\frac{ab}{a+b}$$

```
> d3:=1/(a*(a-b)*(a-c)) + 1/(b*(b-c)*(b-a)) + 1/(c*(c-a)*(c-b));
```

$$\frac{1}{a\,(a-b)\,(a-c)}+\frac{1}{b\,(-a+b)\,(b-c)}+\frac{1}{c\,(-a+c)\,(-b+c)}$$

```
> normal(d3);
```

$$\frac{1}{abc}$$

Nicht jeder Ausdruck wird durch Normalisierung in die einfachstmögliche Form überführt. Statt dessen kann es sinnvoll sein, Teilausdrücke nicht zu expandieren, in Faktoren zu zerlegen, Partialbrüche zu bilden usw. Für diese verschiedenen und in ihren Zielen teilweise widersprüchlichen Simplifikationsstrategien stehen eine Reihe von *Simplifikationsbefehlen* zur Verfügung, und zwar `combine, expand, factor, normal, radsimp, rewrite, simplify`.

2.3 Datenstrukturen

Wegen der besonderen Bedeutung des Kopfsymbols interner Listen, die aus diesen in verschiedenen Kontexten vorkommende sequentielle Strukturen (Mengen, Listen, Funktionsparameter) machen können, sind (externe) Listen in MuPAD (und Maple) abgeleitete Objekte. An zentraler Stelle stehen *Ausdrucksfolgen* (expression sequences), aus denen sich die verschiedenen sequentiellen Strukturen ableiten lassen. Eine solche Ausdrucksfolge besteht einfach aus einer durch Kommata getrennten Folge von regulären Ausdrücken,

```
> u:=v1,v2,v3;
```

$$v1, v2, v3$$

Eine solche Ausdrucksfolge kann nun in verschiedenen Zusammenhängen verwendet werden:

```
> [u]; # Die Liste [v1, v2, v3] #
```

$$[v1, v2, v3]$$

```
> {u}; # Die Menge {v1, v2, v3} #
```

$$\{v1, v2, v3\}$$

```
> max(u); # Ein Funktionsaufruf mit den Argumenten v1, v2, v3 #
```

$$max(v1, v2, v3)$$

```
> _plus(u); # Ein anderer Funktionsaufruf mit v1, v2, v3 #
```

$$v1 + v2 + v3$$

Im letzten Beispiel ist das interne Funktionssymbol verwendet worden, das bei der Verwandlung des Infix-Operators + in einen Präfixoperator verwendet wird.

Ausdruckssequenzen kommen als Bestandteil vieler symbolischer Ausdrücke vor. Man kann aus ihnen diese Sequenzen extrahieren und daraus neue Strukturen aufbauen. Dazu dienen die Operatoren `nops(u)` und `op(u)`

```
> u:=v1 + v2 + v3; L:=op(u); [L]; # Liste der Argumente extrahieren #
```

$$v1 + v2 + v3$$

$$v1, v2, v3$$

$$[v1, v2, v3]$$

Ausdruckssequenzen kann man auch durch den *Sequenzierungsoperator* $, der eine spezielle Version eines `for`-Schleifen-Konstrukts darstellt, erzeugen:

```
>  s:=(i^3 $ i=1..13);
```

$$1, 8, 27, 64, 125, 216, 343, 512, 729, 1000, 1331, 1728, 2197$$

Es wurde eine Ausdruckssequenz generiert, die man nun wie oben bereits beschrieben, weiterverarbeiten kann:

```
>  _plus(s); _mult(s);
```

$$8281$$

$$2414576386840917568389120000000$$

Aus solchen Ausdruckssequenzen kann man insbesondere *Listen* und *Mengen* generieren und auf ihnen die üblichen mathematischen Operationen ausführen.

```
> L:= [a, b, a, b, c]; # Liste erzeugen #
```

$$[a, b, a, b, c]$$

```
> L[2], L[5]; # Zugriff auf Listenelemente #
```

$$b, c$$

```
> contains(L,c), contains(L,100); # Enthaltenseinstest #
```

$$5, 0$$

```
> [a, b] . [c, d] . [b,-a,a] . [c]; # Verkettung von Listen #
```

$$[a, b, c, d, b, -a, a, c]$$

```
> M:= {1,2,3,4,a,b,1,2}; N:= {3,4,a,c};
```

$$\{a, b, 1, 2, 3, 4\}$$

$$\{a, c, 3, 4\}$$

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

$$\{a, b, c, 1, 2, 3, 4\}$$

$$\{a, 3, 4\}$$

```
> {op(L)}; # Liste in Menge verwandeln #
```

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

Schließlich kann man aus solchen Listen (oder Mengen) Teillisten von Elementen mit bestimmten Eigenschaften extrahieren, etwa die Primzahlen bis 50:

```
> s:=$1..50: select(s,isprime);
```

$$2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47$$

Eine weitere wichtige Datenstruktur sind *Felder*, die man für Matrix- und Vektor-Operationen benötigt.

```
> A:= array(1..2,1..2,[ [11,12],[21,22] ]);
```

$$\begin{pmatrix} 11 & 12 \\ 21 & 22 \end{pmatrix}$$

```
> A[1,1] + A[2,2];
```

Schließlich kann man auch *Tabellen* definieren. Im Unterschied zu Listen, deren Elemente mit $1, \ldots, n$ indiziert sind, kann der Index in einer Tabelle ein beliebiger Ausdruck sein, so daß man letztere z.B. als assoziatives Gedächtnis nutzen kann.

```
>  T[a]:= 2: T[3]:= 4: T;
```

$$table(a = 2, 3 = 4)$$

```
>  S:= table(1=2, 3=4);
```

$$table(1 = 2, 3 = 4)$$

```
>  S[1,1]:= 11: S[1,2]:= 22: S;
```

$$table(1 = 2, 3 = 4, (1, 1) = 11, (1, 2) = 22)$$

2.4 Steuerstrukturen

Alle komplexeren Programmkonstrukte sind aus Schlüsselworten und (symbolischen) *Ausdrücken* $< Ex >$ aufgebaut, deren Syntax wir an dieser Stelle nunmehr als hinreichend bekannt voraussetzen wollen. Aus ihnen werden *Anweisungen* $< stmt >$ aufgebaut, deren einfachste die *Zuweisung* eines Werts an eine Variable ist.

Separator:

$< sep >$ `::= ; | : .`

Anweisungsfolge:

$< stseq >$ `::=` $< stmt >$ `[`$< sep >< stseq >$`] .`

Bedingter Sprung (if-Anweisung):

$< ifst >$ `::= if` $< BoolEx >$ `then` $< stseq >$ `[`$< elsepart >$`] end_if .`

$< elsepart >$ `::= else` $< stseq >$ `| elif then` $< stseq >$ `[`$< elsepart >$`].`

Case-Anweisung: `siehe Manual`

For-Anweisung:

$< for >$ `::= for` $< id >$ `from` $< Ex >$ `to` $< Ex >$
`[step` $< Ex >$`] do` $< stseq >$ `end_for |`
`for` $< id >$ `in` $< Ex >$ `do` $< stseq >$ `end_for .`

While-Schleife

$< while >$ `::= while` $< BoolEx >$ `do` $< stseq >$ `end_while .`

Repeat-Schleife

$< repeat >$ `::= repeat` $< stseq >$ `until` $< BoolEx >$ `end_repeat .`

Für die Bedeutung der Ablaufsteuerungsanweisungen `next`, `break` und `quit` sei auf das Manual verwiesen.

Schließlich kann man ebenfalls Funktionen definieren, was wir an einem einfachen Beispiel, der gcd-Berechnung zweier ganzer Zahlen mit Hilfe des Euklidschen Algorithmus mit Ausdruck der Zwischenergebnisse demonstrieren wollen:

```
ourgcd:= proc(a, b) begin
      if b = 0 then a
      else print(Unquoted, "gcd(".a.",".b.")"); ourgcd(b, a mod b)
      end_if
end_proc;
```

Die Funktion ist rekursiv implementiert, wobei zusätzlich ein aus den einzelnen Parametern der Rechnung zusammengebauter String mit einem `print`-Befehl Auskunft über den aktuellen Stand der Rechnung gibt.

Der einheitliche Namensraum für Bezeichner und Symbole, insbesondere Funktionssymbole, erlaubt es, auch Funktionsbezeichner auf dieselbe Weise zu behandeln, so daß hier dem Bezeichner `ourgcd` eine Funktion zugewiesen wird mit der allgemeinen Struktur

$< funct >$ `::= proc(`*Parameter*`) [`*DeklPart*`] begin` $< stseq >$ `end_proc .`

Auch hier sei auf Details im Handbuch verwiesen.

2.5 MuPADs Domain-Konzept. Einbindung mathematischer Pakete

MuPAD verfolgt ein allgemeines Konzept zur Definition mathematischer Strukturen, das sich von den beiden anderen CAS unterscheidet und bereits in Ansätzen Typprüfungen sowie objektorientiertes Herangehen ermöglicht, was aus anderen Programmiersprachen als mächtiges Hilfsmittel bekannt ist. Wir wollen dieses Konzept am Beispiel von Matrizen demonstrieren. Alle bisher betrachteten Ausdrücke gehören zum Domain `Expression`.

Um in MuPAD Matrizen zu definieren, muß zunächst ein entsprechender Datentyp vereinbart werden.

```
> M:=Dom::Matrix();
```

$$Dom :: Matrix(Dom :: ExpressionField(id, iszero))$$

Dies definiert einen Bereich M, dessen Elemente Matrizen mit beliebigen rationalen Ausdrücken als Einträge sein können. Genausogut hätten wir uns auf Matrizen mit ganzzahligen Einträgen beschränken können durch

```
> M1:=Dom::Matrix(Dom::Integer);
```

$$Dom :: Matrix(Dom :: Integer)$$

Die syntaktische Analogie zu Member-Definitionen für Klassen in C++ ist nicht zufällig. `Matrix` und `Integer` sind zwei Konstruktoren aus der Klasse `Dom` der Domainkonstruktoren. Ungewöhnlich ist nur der Rückgabewert: es wird ein neuer *Datentyp* zurückgegeben. MuPAD behandelt auf diese Weise Datentypen wie andere symbolische Ausdrücke, was es ermöglicht, Funktionen und Datentypkonstruktoren auf eine einheitliche Weise zu definieren. Dies führt zu einem Konzept parametrisierter Datentypen, das weit über die Templates in C++ hinausreicht, was an dieser Stelle aber nicht weiter ausgeführt werden soll.

Neue Instanzen von M können nun durch die vererbten Konstruktoren etwa aus verschachtelten Listen gewonnen werden.

```
> A:=M([[x+1,x-1],[x-1,x+1]]);
```

$$\begin{pmatrix} x+1 & x-1 \\ x-1 & x+1 \end{pmatrix}$$

erzeugt eine Matrix mit polynomialen Einträgen, während derselbe Aufruf mit $M1$ zu einem Fehler führt

```
> A1:=M1([[x+1,x-1],[x-1,x+1]]);

Error: unable to define matrix over Dom::Integer
```

weil die Einträge die geforderte Typeigenschaft (ganzzahlig zu sein) nicht erfüllen. Statt dessen könnte man eine ganzzahlige Matrix definieren wie etwa

```
> A1:=M1([[1,2],[3,4]]);
```

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

Mit diesen Matrizen kann man nun die üblichen Operationen wie Multiplikation oder Inversenberechnung ausführen.

```
> B:=A^2; C:=A^(-1);
```

$$\begin{pmatrix} (x+1)^2 + (x-1)^2 & 2\,(x+1)\,(x-1) \\ 2\,(x+1)\,(x-1) & (x+1)^2 + (x-1)^2 \end{pmatrix}$$

$$\begin{pmatrix} \dfrac{\dfrac{(x-1)\,(-x+1)}{(-x-1)\left(x - \dfrac{(x-1)^2}{x+1} + 1\right)} + 1}{x+1} & \dfrac{-x+1}{(x+1)\left(x - \dfrac{(x-1)^2}{x+1} + 1\right)} \\ \dfrac{x-1}{(-x-1)\left(x - \dfrac{(x-1)^2}{x+1} + 1\right)} & \dfrac{1}{x - \dfrac{(x-1)^2}{x+1} + 1} \end{pmatrix}$$

Dasselbe gilt für Instanzen aus dem Bereich M_1, wobei zu beachten ist, daß die Berechnung A_1^{-1} fehlschlägt, weil diese Matrix keine *ganzzahlige* Inverse besitzt.

Schwieriger ist es, die entsprechenden Einträge zu vereinfachen. `simplify(B)` oder `normal(B)` zeigen keine Wirkung. Sehen wir uns dazu die Struktur von B und C an: `op(B)` und `op(C)` liefert die Einträge der Matrizen als Ausdruckssequenz zurück, auf die der Operator `normal` jeweils anzuwenden wäre. Dies kann durch `map` erfolgen:

```
> map(B,normal); map(C,normal);
```

$$\begin{pmatrix} 2x^2+2 & 2x^2-2 \\ 2x^2-2 & 2x^2+2 \end{pmatrix}$$

$$\begin{pmatrix} \dfrac{x+1}{4x} & \dfrac{-x+1}{4x} \\ \dfrac{-x+1}{4x} & \dfrac{x+1}{4x} \end{pmatrix}$$

Schließlich kann man kompliziertere Matrixoperationen wie etwa die Berechnung der entsprechenden Determinante ausführen. Die dazu notwendigen algorithmischen Kenntnisse sind in einem Paket `linalg` enthalten, das explizit nachgeladen werden muß, entweder durch einen qualifizierten Funktionsbezeichner `linalg::det`, der den Paketnamen mit enthält, oder durch Nachladen des gesamten Pakets `linalg` (genauer gesagt erfolgt das Nachladen automatisch; die folgende `export`-Anweisung führt die entsprechenden Bezeichner nur in den globalen Namensraum ein).

```
> export(linalg); det(A); normal(%);
```

$$(x+1)^2 - (x-1)^2$$

$$4x$$

Ein anderes Beispiel für die Anwendung von Domains ist die Definition eines modularen Bereichs:

```
> Z7:= Dom::IntegerMod(7): # Z mod 7 #
```

```
> a := Z7(9); b := Z7(2)-1/Z7(3);
```

$$2 \; mod \; 7$$

$$4 \; mod \; 7$$

Die Polymorphie der arithmetischen Operatoren in diesen Rechnungen ist in einem objektorientierten Ansatz gut bekannt, wo der anzuwendende Operator aus dem Operatorsymbol *und* dem Datentyp der Operanden deduziert wird.

```
> a+b; a*b; a^7; a/b; type(a);
```

$$6 \; mod \; 7$$

$$1 \; mod \; 7$$

$$2 \; mod \; 7$$

$$4 \; mod \; 7$$

$$Dom :: IntegerMod(7)$$

Dies wird auch für das Rechnen mit Polynomen über verschiedenen Grundbereichen genutzt:

```
p:=x^3+x+1;

                          3
                      x + x  + 1

p1:=poly(p);

                          3
                 poly(x + x  + 1, [x])


p2:=poly(p,IntMod(2));

                  3
         poly(x + x  + 1, [x], IntMod(2))

p3:=poly(p,IntMod(3));

                  3
         poly(x + x  + 1, [x], IntMod(3))
```

```
factor(p);

                              3
                    [1, x + x  + 1, 1]

factor(p1);

                              3
               [1, poly(x + x  + 1, [x]), 1]

factor(p2);

                         3
          [1, poly(x + x  + 1, [x], IntMod(2)), 1]

factor(p3);

                                                  2
[1, poly(x - 1, [x], IntMod(3)), 1, poly(x + x  - 1, [x], IntMod(3)), 1]
```

p ist in diesen Rechnungen ein Polynom ohne Typinformation und wird als allgemeiner Ausdruck behandelt. p_1 ist dasselbe Polynom, betrachtet als Polynom in x (mit allgemeinen Koeffizienten). Solche Polynome bilden als mathematische Struktur einen Ring, in dem z.B. Inversenbildung nicht bzw. nur eingeschränkt möglich ist. Damit kann man zwar $1/p$, aber nicht $1/p_1$ bilden. `poly` konstruiert also, wie in objektorientierten Ansätzen üblich, eine Klasse (hier `domain` genannt) zusammen mit den Methoden, mit denen auf Klassenelemente zugegriffen werden kann. Diese hängen neben dem ersten Parameter auch von weiteren ab. So kann man einen Koeffizientenbereich sowie (hier nicht demonstriert) eine Variablenliste als Parameter angeben. Entsprechend sind $p_2 \in \mathbf{Z}/2\mathbf{Z}[x]$ und $p_3 \in \mathbf{Z}/3\mathbf{Z}[x]$ zwei Polynome, die sich aus p durch Reduktion der Koeffizienten ergeben. Der Aufruf von `factor` verwendet diese Typinformation, um die jeweils richtigen Routinen einzubinden. Wir erkennen, daß p_1 und p_2 irreduzibel sind, während p_3 in Faktoren zerlegbar ist. In der Tat ist

$$(x-1)\left(x+x^2-1\right) = -2x + x^3 + 1 \equiv p \ (mod\ 3).$$

2.6 Der Solve-Operator

MuPAD unterscheidet zwischen der Nullstellenbestimmung für einzelne Polynome und für polynomiale Gleichungssysteme. Für ein einzelnes Polynom $p(x)$ findet `solve(p(x), x)` alle Nullstellen und gibt sie als Nullstellen*menge* zurück.

```
> s:=solve((p:=x^2-3*x+2),x);

                              {1, 2}
```

Wir können nun einfach die Probe machen durch

```
> subs(p,x=op(s,i)) $ i=1..2;

                               0, 0
```

Nullstellen werden dabei so weit wie möglich, d.h. für jeden Faktor von $p(x)$ vom Grad < 5, durch Radikale ausgedrückt, was zu teilweise langen Ausgaben führt. Für alle anderen Polynome wird ein `RootOf`-Symbol verwendet:

```
> s:=solve((p:=x^5-x+1),x);

                                       5
                     {RootOf(- x + x  + 1)}
```

Dieses Symbol kann unter einigen Umständen in eine Ausdrucksfolge expandieren, etwa wenn numerische Nähreungswerte gesucht sind:

```
> map(s,float);

{-1.167303978, 0.7648844336 + 0.352471546 I, 0.7648844336 - 0.352471546 I,

- 0.1812324444 - 1.083954101 I, - 0.1812324444 + 1.083954101 I}
```

Für Systeme von Gleichungen hat die Lösung eine leicht andere, ebenfalls verständliche Gestalt:

```
> s:=solve((u:={x^2+y=1,x-y=2}),{x,y});
```

$$\left\{\left[y=x-2,x=-\frac{\sqrt{13}}{2}-1/2\right],\left[y=x-2,x=\frac{\sqrt{13}}{2}-1/2\right]\right\}$$

oder

```
> s:=solve((u:={x^2+y=1,x-y=2}),{x,y}, BackSubstitution=TRUE );
```

$$\left\{\left[y=-\frac{\sqrt{13}}{2}-5/2,x=-\frac{\sqrt{13}}{2}-1/2\right],\left[y=\frac{\sqrt{13}}{2}-5/2,x=\frac{\sqrt{13}}{2}-1/2\right]\right\}$$

```
> simplify(subs(u,op(s,i))) $ i=1..2;

                  {1 = 1, 2 = 2}, {1 = 1, 2 = 2}
```

Hier werden bereits Nullstellen von Polynomen höheren als zweiten Grades durch `RootOf`-Symbole ersetzt, was durch Setzen der Variablen `MaxDegree` verändert werden kann. Insbesondere hat die Lösungsmenge eines Systems aus einem Polynom ein anderes Aussehen als die Menge der Nullstellen, die der `solve`-Operator zurückgibt, wenn das Polynom "pur" eingegeben wird.

Für trigonometrische Systeme erhalten wir dagegen:

```
> s:=solve((u:=sin(x)-cos(x)),x);
```

$$\left\{\left(\frac{\pi}{4},\frac{9\,\pi}{4},-\frac{7\,\pi}{4},\ldots\right),\left(-\frac{3\,\pi}{4},\frac{5\,\pi}{4},-\frac{11\,\pi}{4},\ldots\right)\right\}$$

Dies sind zwei unendliche Serien von Lösungen, wobei sich die Lösungsmenge eigentlich als die Vereinigung dieser beiden Mengen ergibt. Jeder der beiden Einträge gehört zum speziellen Domain *DiscreteSet*, wie man mit

```
> type(op(s,1));

                        Dom::DiscreteSet
```

leicht nachprüft. Intern wird die erwartete Darstellung als Zahlenfolge verwendet:

```
> expr(op(s,1));
```

$$\texttt{Dom::DiscreteSet}\left(\frac{\pi}{4} + 2k\,\pi,\ k = -\infty\ldots\infty, TRUE\right)$$

Die Probe kann mit beliebigen Elementen als diesen Folgen durchgeführt werden, etwa mit dem Eintrag 1001 der ersten Folge:

```
> expand(subs(u,x=op(op(s,1),1001)));
```

$$0$$

3 Maple

Wir wollen zunächst zeigen, wie die für MuPAD demonstrierten Rechnungen mit Maple auszuführen sind und werden dabei zugleich auf wesentliche Unterschiede zu MuPAD hinweisen. Der wohl größte Unterschied besteht darin, daß Maple ein weitgehend typloses System ist.

```
> 1/5 + 1/2; 2/3*(1/4+3/5); (1/2+1/3)^(-1);
```

$$7/10$$

$$\frac{17}{30}$$

$$6/5$$

```
> v:=1/2+exp(1)^2/6; evalf(v); evalf(v-sqrt(3));
```

$$v := \frac{e^2}{6} + 1/2$$

$$1.731509350$$

$$-0.000541458$$

Das Symbol E wird in neueren Maple-Versionen wegen häufiger Namenskollision mit einer gleichlautenden Variablen nicht mehr verwendet. Zur näherungsweisen Berechnung wird keine Typumwandlung, sondern eine Float-Evaluierung `evalf` ausgeführt. Die entsprechende Präzision kann man mit einem zweiten Parameter erreichen:

```
> evalf(sqrt(2),50);
```

$$1.4142135623730950488016887242096980785696718753769 4$$

Die Funktionen zum Rechnen mit ganzen Zahlen haben ähnliche Namen wie in MuPAD.

```
> igcd(105,315,2898,7368172368127368127368127327);
```

$$7$$

```
> u:=333!; # auch factorial(333)
```

```
u := 1033446543458805915609396553829751655062226004168206282343290246978318\
     85979142765685527001948498779298943759502525704770804183527325976587456\
     65925604704669227133726477243854317836635130694123893711638533001980496\
     22987566547659856882180617030376554048981440223415990154044043213415584\
     45429624451536463305955882916059244292113522799434713728172799387209748\
     95260387784578239150931816946786416232516666251965421919651838044618050\
     99129440354695893074541974383696652019873520112325588408926327282984664\
     05388269798436428857757916415751091787535095800016603920923967986489243\
     75401024147883702298145910046889402880394195369984000000000000000000000\
     000000000000000000000000000000000000000000000000000000000000
```

```
> ifactor(u);
```

$$(2)^{328}\ (3)^{165}\ (5)^{81}\ (7)^{53}\ (11)^{32}\ (13)^{26}\ (17)^{20}\ (19)^{17}\ (23)^{14}\ (29)^{11}\ (31)^{10}\ (37)^{9}\ (41)^{8}\ (43)^{7}\ (47)^{7}\ (53)^{6}\ (59)^{5}\ (61)^{5}\ (67)^{4}\ (71)^{4}\ (73)^{4}\ (79)^{4}\ (83)^{4}\ (89)^{3}\ (97)^{3}\ (101)^{3}\ (103)^{3}\ (107)^{3}\ (109)^{3}\ (113)^{2}\ (127)^{2}\ (131)^{2}\ (137)^{2}\ (139)^{2}\ (149)^{2}\ (151)^{2}\ (157)^{2}\ (163)^{2}\ (167)\ (173)\ (179)\ (181)\ (191)\ (193)\ (197)\ (199)\ (211)\ (223)\ (227)\ (229)\ (233)\ (239)\ (241)\ (251)\ (257)\ (263)\ (269)\ (271)\ (277)\ (281)\ (283)\ (293)\ (307)\ (311)\ (313)\ (317)\ (331)$$

```
> u:=factorial(40); v:=nextprime(u);
```

$$u := 815915283247897734345611269596115894272000000000$$

$$v := 815915283247897734345611269596115894272000000053$$

```
> isprime(v);
```

$$true$$

Das Rechnen mit rationalen Ausdrücken erfolgt wie bei MuPAD. Es stehen die *Simplifikationsbefehle* `combine, expand, factor, normal, radnormal, simplify` sowie die speziellen Evaluationsmodi `evalf, evala, evalm` für Float-Zahlen, algebraische Zahlen sowie Matrizen zur Verfügung. Außerdem gibt es eine sehr mächtige Funktion `convert`, die eine Konversion von Ausdrücken eines Typs in einen vorgegebenen anderen Typ ermöglicht, so z.B. Dezimalbrüche in echte Brüche, Winkelfunktionen in Exponentialfunktionen, rationale Funktionen in Partialbrüche etc.

Ebenso wird ein ähnliches Datenstrukturkonzept verfolgt, in dem *Ausdruckssequenzen* eine zentrale Rolle spielen.

```
> u:=v1,v2,v3;
```

$$v1, v2, v3$$

```
> [u]; # Die Liste [v1, v2, v3]
```

$$[v1, v2, v3]$$

```
> {u}; # Die Menge {v1, v2, v3}
```

$$\{v1, v2, v3\}$$

`convert` erlaubt es auch, eine Datenstruktur in eine andere zu verwandeln, indem die Typinformation geändert wird.

```
> convert([u],`+`);
```

$$v1 + v2 + v3$$

In diesem Beispiel werden "Backquotes" verwendet, die den entsprechenden String vor der Auswertung schützen, hier also verhindern, daß beim Auswerten der Argumente während des Funktionsaufrufs von `convert` das + als Additionszeichen erkannt wird, was eine Fehlermeldung erzeugen würde, weil das Additionszeichen + nur als Bestandteil eines Ausdrucks auftreten kann, das zweite Argument aber kein regulärer Ausdruck ist. Seine Bedeutung entfaltet + erst im Inneren der Funktionskörpers von `convert`. Man beachte, daß u hier **keine** Ausdruckssequenz ist, sondern (etwa) eine richtige Liste.

Eine Aufstellung der Operationen, die auf Mengen und Listen ausgeführt werden können, findet man auf der entsprechenden Hilfeseite von Maple.

Ausdruckssequenzen kann man in Maple durch den Sequenzierungsoperator `seq` erzeugen:

```
>  s:=seq(i^3, i=1..13);

 s := 1, 8, 27, 64, 125, 216, 343, 512, 729, 1000, 1331, 1728, 2197
```

Diese Sequenz kann man nun ähnlich wie unter MuPAD bearbeiten:

```
> convert([s],'+'); convert([s],'*');
```

$$8281$$

$$24145763868409175683891200000 0$$

Die Definition von *Feldern* und *Tabellen* ist im wesentlichen so wie in MuPAD, jedoch weicht der Mechanismus der Zuweisung zu Variablen vom üblichen ab.

```
> A:= array(1..2,1..2,[ [11,12],[21,22] ]);
```

$$A := \begin{bmatrix} 11 & 12 \\ 21 & 22 \end{bmatrix}$$

definiert ein zweidimensionales Feld (das etwa als Matrix interpretiert werden kann).

```
> whattype(A);
```

$$string$$

```
> type(A,matrix);
```

$$true$$

```
> A;
```

$$A$$

```
> B:=A^2;
```

$$B := A^2$$

sind sicher am Anfang nicht ganz verständlich. Hier ist jedoch zu beachten, daß Felder und Tabellen nicht automatisch ausgewertet werden, sondern erst durch den Aufruf von `eval` oder wenn es der Kontext bestimmt. Möchte man Felder als Vektoren oder Matrizen interpretieren, so steht dafür der spezielle Auswerteoperator `evalm` zur Verfügung.

```
> evalm(B);
```

$$\begin{bmatrix} 373 & 396 \\ 693 & 736 \end{bmatrix}$$

```
> C:=array(1..2,1..1,[[1],[1]]);
```

$$C := \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

```
> evalm(B&*C);
```

$$\begin{bmatrix} 769 \\ 1429 \end{bmatrix}$$

Beachten Sie in der letzten Anweisung die Verwendung des nichtkommutativen Multiplikationsoperators &∗.

3.1 Steuerstrukturen

Die Steuerstrukturen in Maple stimmen im wesentlichen mit denen von MuPAD überein.

Separator:

$< sep >$ `::= ; | : .`

Anweisungsfolge:

$< stseq >$ `::=` $< stmt >$ $[< sep >< stseq >]$ `.`

Bedingter Sprung (if-Anweisung):

$< ifst >$ `::= if` $< BoolEx >$ `then` $< stseq >$ `[`$< elsepart >$`] fi .`

$< elsepart >$ `::= else` $< stseq >$ `| elif then` $< stseq >$ `[`$< elsepart >$`].`

For/While-Anweisung:

$< for >$ `::= [for` $< id >$`] [from` $< Ex >$`] [by` $< Ex >$`] [to` $< Ex >$`]`
`[while` $< Ex >$`] do` $< stseq >$ `od |`
`[for` $< id >$`] [in` $< Ex >$`] [while` $< Ex >$`] do` $< stseq >$ `od .`

Man beachte, daß Maple nur ein einziges Schleifenkonstrukt kennt, das ähnlich wie in C mit den Anweisungen `next` und `break` ausreichend flexibel ist. Für deren Bedeutung sei auf das Hilfesystem verwiesen.

Funktionsdefinitionen folgen ebenfalls dem Konzept des gemenisamen Namensraumes mit Variablenbezeichnern und Symbolen wie wir es bereits bei MuPAD kennengelernt haben. Als Beispiel betrachten wir wieder die gcd-Berechnung zweier ganzer Zahlen mit Hilfe des Euklidschen Algorithmus mit Ausdruck der Zwischenergebnisse:

```
ourgcd:= proc(a, b)
     if b = 0 then a else print(`gcd(`.a.`,`.b.`)`); ourgcd(b, a mod b) fi
end;
```

Einziger Unterschied zu MuPAD, neben den Backquotes zur Bezeichnung von Strings, ist das Fehlen des Schlüsselworts `begin`, das den Deklarationsteil vom Anweisungsteil trennt. Die allgemeine Syntax lautet hier

$< funct >$ `::= proc(`$Parameter$`) [`$DeklPart$`]` $< stseq >$ `end .`

Maple bietet daneben noch eine weitere Möglichkeit der Funktionsdefinition, die an die entsprechende pfeilbasierte mathematische Notation angelehnt ist:

```
> f:=x -> x^2;
```

definiert etwa die Funktion $f(x) = x^2$. Multivariate und Vektorfunktionen sind ebenfalls erlaubt, wenn man die Argumente bzw. Resultate in Klammern einschließt. Eine solche Definition

```
> f:=vars -> result;
```

ist äquivalent (und intern identisch) mit

```
> f:=proc(vars) option operator, arrow; result end;
```

3.2 Matrizen und lineare Algebra. Einbindung mathematischer Pakete

Auf die Besonderheiten, die bei der Auswertung von Matrixausdrücken zu beachten sind, wurde schon oben hingewiesen. Maple stellt einen Konstruktur `matrix` zur Verfügung, mit dem Matrizen als spezielle Felder definiert werden können.

```
> M:=matrix([[x+1,x-1],[x-1,x+1]]);
```

$$M := \begin{bmatrix} x+1 & x-1 \\ x-1 & x+1 \end{bmatrix}$$

```
> whattype(eval(M));
```

$$array$$

```
> op(eval(M));
```

$$1..2, 1..2, [(1,1) = x+1, (1,2) = x-1, (2,1) = x-1, (2,2) = x+1]$$

Nun können dieselben Umformungen, wie wir sie im entsprechenden Abschnitt zu MuPAD kennengelernt haben, ausgeführt werden.

```
> B:=evalm(M^2); C:=evalm(A^(-1));
```

$$B := \begin{bmatrix} (x+1)^2 + (x-1)^2 & 2\,(x+1)\,(x-1) \\ 2\,(x+1)\,(x-1) & (x+1)^2 + (x-1)^2 \end{bmatrix}$$

$$C := \begin{bmatrix} 1/4\,\frac{x+1}{x} & -1/4\,\frac{x-1}{x} \\ -1/4\,\frac{x-1}{x} & 1/4\,\frac{x+1}{x} \end{bmatrix}$$

```
> map(expand,B);
```

$$\begin{bmatrix} 2\,x^2+2 & 2\,x^2-2 \\ 2\,x^2-2 & 2\,x^2+2 \end{bmatrix}$$

Für die Berechnung der Determinanten muß auch in Maple das entsprechende Paket `linalg` mit Funktionen zur linearen Algebra explizit geladen werden.

```
> with(linalg);
```

```
Warning, new definition for norm
Warning, new definition for trace
[BlockDiagonal, GramSchmidt, JordanBlock, LUdecomp, QRdecomp, Wronskian,
 addcol, addrow, adj, adjoint, angle, augment, backsub, band, basis, bezout,
 blockmatrix, charmat, charpoly, cholesky, col, coldim, colspace, colspan,
 companion, concat, cond, copyinto, crossprod, curl, definite, delcols,
 delrows, det, diag, diverge, dotprod, eigenvals, eigenvalues, eigenvectors,
 eigenvects, entermatrix, equal, exponential, extend, ffgausselim,
 fibonacci, forwardsub, frobenius, gausselim, gaussjord, geneqns, genmatrix,
 grad, hadamard, hermite, hessian, hilbert, htranspose, ihermite, indexfunc,
 innerprod, intbasis, inverse, ismith, issimilar, iszero, jacobian, jordan,
 kernel, laplacian, leastsqrs, linsolve, matadd, matrix, minor, minpoly,
 mulcol, mulrow, multiply, norm, normalize, nullspace, orthog, permanent,
 pivot, potential, randmatrix, randvector, rank, ratform, row, rowdim,
 rowspace, rowspan, rref, scalarmul, singularvals, smith, stack, submatrix,
 subvector, sumbasis, swapcol, swaprow, sylvester, toeplitz, trace,
 transpose, vandermonde, vecpotent, vectdim, vector, wronskian]
```

Dies ist die Liste der Funktionen, die von diesem Paket exportiert werden.

```
> det(M);
```

$$4\,x$$

3.3 Rechnen in modularen Bereichen und mit algebraischen Zahlen

Da Maple im Unterschied zu MuPAD keinen objektorientierten Zugang verfolgt, muß das Rechnen in komplizierteren mathematischen Strukturen auf spezielle Weise organisiert werden. Dies betrifft in Maple das Rechnen in modularen Bereichen und das Rechnen mit algebraischen Zahlen.

Für ersteres steht der Infixoperator *mod* zur Verfügung.

```
>  9 mod 7; 2-1/3 mod 7;
```

$$2$$

$$4$$

Die Rechnungen werden dabei im bzw. über dem entsprechenden Restklassenring, hier $\mathbf{Z}/7\mathbf{Z}$ ausgeführt. Dies bezieht sich nicht nur auf einfache Arithmetik, sondern auch auf Polynom- und Matrixrechnungen über endlichen Körpern einschließlich Faktorisierung.

Dabei sind einige Besonderheiten zu beachten. So ist es sicher nicht sinnvoll, $i^n \; mod \; m$ auf gewöhnlichem Wege zu berechnen, indem im Aufruf `mod(n^i,m)` erst die inneren Argumente ausgewertet werden, d.h. in diesem Fall die möglicherweise lange ganze Zahl i^n bestimmt wird, um sie dann $mod\; m$ zu reduzieren. Statt dessen ist es sinnvoller, einen eigenen modularen Potenzoperator zu definieren, der Zwischenergebnisse stets wieder $mod\; m$ reduziert, d.h. das innere Argument von $mod(i^n, m)$ erst *später* auszuwerten. Dies wird hier durch die Verwendung von &^ statt ^ erreicht, was einem speziellen Zweig der Abarbeitung des Aufrufs `mod` entspricht.

```
> 2&^10;
```

$$2\&\hat{}10$$

&^ hat außerhalb der *mod*-Umgebung keine spezielle Bedutung, die über die eines Funktionssymbols hinausgeht.

```
> 2&^10 mod 3;
```

$$1$$

Dasselbe gilt für eine Reihe weiterer Funktionen, wie etwa `Det, Expand, Factor, Gcd, Normal`, die die jeweiligen *inerten* Varianten der entsprechenden Funktionen darstellen und sogar teilweise zu anderen Ergebnissen führen als ihre nicht inerten Äquivalente. Man beachte, daß jede der inerten Funktionen nur *innerhalb* der `mod`-Umgebung mehr als ein Funktionssymbol ist.

So berechnet etwa

```
> factor(x^3+3) mod 2;
```

$$x^3 + 1$$

die Reduktion der Faktorisierung von $x^3 + 3$ über den ganzen Zahlen, während

```
> Factor(x^3+3) mod 2;
```

$$(x + 1)\left(x^2 + x + 1\right)$$

dasselbe Polynom als Element von $\mathbf{Z}/3\mathbf{Z}[x]$ in Faktoren zerlegt.

Das Ganze gilt sinngemäß auch für das Rechnen über algebraischen Erweiterungskörpern von $\mathbf{Q}$, wo die Funktion `evala` die Rolle von `mod` übernimmt. Die Mechanismen sind ähnlich, wie übrigens auch für die oben behandelte Matrix-Evaluierungsfunktion `evalm`. Für Details sei auf die Systemdokumentation verwiesen.

3.4 Der `Solve`-Operator

Maples `Solve`-Operator folgt in vielen Punkten denselben Regeln wie wir sie für MuPAD kennengelernt haben. Der wohl wichtigste Unterschied findet sich in der Ausgabesyntax, wo Maple stets eine Ausdruckssequenz zurückliefert, die vom Benutzer zur weiteren Verarbeitung explizit in eine Menge oder Liste verwandelt werden muß.

```
> p:=x^2-3*x+2; s:=[solve(p,x)];

                              2
                        p := x  - 3 x + 2

                           s := [1, 2]
```

Mit ähnlichen Sprachmitteln, bei gegenüber MuPAD leicht veränderter Syntax, können wir auch hier die Probe machen:

```
> seq(subs(x=op(i,s),p),i=1..2);

                               0, 0
```

Nullstellen werden standardmäßig bei einzelnen Gleichungen bis zum Grad 3 in Radikalen aufgelöst. Diese Einstellung kann mit der Variablen `_EnvExplicit` verändert werden.

```
> s:=solve(x^5-x+1,x);

                                     5
                      s := RootOf(_Z  - _Z + 1)

> allvalues(s);

-1.167303978, -.1812324445 - 1.083954101 I, -.1812324445 + 1.083954101 I,

    .7648844336 - .3524715460 I, .7648844336 + .3524715460 I
```

Mit diesem Kommando kann man `RootOf`-Symbole expandieren, wobei entweder Näherungswerte oder Ausdrücke in Radikalen erzeugt werden.

Bei Gleichungssystemen werden selbst für quadratische Irrationalitäten `RootOf`-Symbole verwendet.

```
> u:={x^2+y=1,x-y=2}; s:=solve(u,{x,y});

                           2
                    u := {x  + y = 1, x - y = 2}

                         2                            2
     s := {y = RootOf(_Z  + 5 _Z + 3), x = RootOf(_Z  + 5 _Z + 3) + 2}

> s1:=[allvalues(s)];

                             1/2                      1/2
s1 := [{y = - 5/2 + 1/2 13   , x = - 1/2 + 1/2 13   },

                          1/2                      1/2
    {y = - 5/2 - 1/2 13   , x = - 1/2 - 1/2 13   }]
```

```
> seq(simplify(subs(op(i,s1),u)),i=1..2);

                    {1 = 1, 2 = 2}, {1 = 1, 2 = 2}
```

Zum Lösen einzelner Gleichungstypen gibt es verschiedene Varianten des Solve-Operators: `dsolve` für Differentialgleichungen, `fsolve` für numerische Lösungen, `isolve` für ganzzahlige Lösungen, `msolve` für Aufgaben mit Kongruenzen, `rsolve` für Rekursionsbeziehungen und `linsolve` aus dem Paket `linalg`für Matrix-Gleichungen.

Für trigonometrische Systeme erhalten wir mit den Standardeinstellungen:

```
> u:=sin(x)-cos(x); s:=solve(u,x);

                         u := sin(x) - cos(x)

                              s := 1/4 Pi
```

d.h. nur eine partielle Lösung. Dies kann man durch Setzen der globalen Variablen

```
> _EnvAllSolutions := true
```

ändern. Danach ergibt sich

```
> s1:=solve(u,x);

                         s1 := 1/4 Pi + Pi _Z~
```

Die Tilde hinter der neu eingeführten Variablen `_Z` weist darauf hin, daß für diese eine Eigenschaft gespeichert ist, in diesem Fall eine ganze Zahl zu repräsentieren.

```
> simplify(subs(x=s,u)); simplify(subs(x=s1,u));

                                   0

               sin(1/4 Pi + Pi _Z~) - cos(1/4 Pi + Pi _Z~)
```

Diese Rechnung zeigt, daß `simplify` mit dieser Zusatzinformation nicht anzufangen weiß. Verwenden wir statt dessen

```
> expand(subs(x=s1,u));

                                   0
```

so erhalten wir auch in diesem Fall das für eine erfolgreiche Probe notwendige Ergebnis.

4 Reduce

Reduce ist eines der "alten" Computeralgebrasysteme, das stärker als die anderen auf LISP-Konzepten basiert. Ein kleiner Kern stellt den LISP-Dialekt RLISP zur Verfügung, in dem seinerseits die komplexeren mathematischen Algorithmen implementiert sind. Dieser trotz Pascal-ähnlicher Notation für den ungeübten Benutzer kryptische *Symbolische Modus* wird überlagert von einer Schnittstelle zur äußeren Welt, dem *Algebraischen Modus*, mit dem wir uns im weiteren ausschließlich beschäftigen werden. Er bietet den gesamten (inhaltlichen) Komfort, den man von einer Nutzerschnittstelle eines CAS erwartet, erlaubt es allerdings nicht, sich Kenntnisse über die Art der Implementierung einzelner Algorithmen zu verschaffen oder gar selbst die Systemfähigkeiten wesentlich zu erweitern. Beides ist ohne tieferes Eindringen in die Regeln des Programmierens im symbolischen Modus nicht möglich, obwohl die Quellen fast aller Algorithmen offenliegen.

Die beiden grundlegenden Datenstrukturen, die in Reduce Verwendung finden, sind *Listen* (ohne den Umweg über Ausdruckssequenzen) und *Standardquotienten* (s.q.). Letztere sind rationale Ausdrücke mit Koeffizienten aus dem *aktiven Zahlbereich* und werden automatisch entsprechend den jeweils aktuellen Einstellungen normalisiert, standardmäßig also in ihre Normalform überführt. So führt

```
> q:=(a+b)^10;


 10        9           8  2         7  3         6  4         5  5         4  6
a   + 10*a *b + 45*a *b  + 120*a *b  + 210*a *b  + 252*a *b  + 210*a *b

        3  7        2  8          9    10
 + 120*a *b  + 45*a *b  + 10*a*b  + b
```

sofort zu einer voll expandierten Form, was das Vergleichen von Ausdrücken vereinfacht. Dasselbe gilt für rationale Funktionen, wenn mit dem Schalter `on gcd;` die (manchmal kostspielige) gcd-Berechnung von Zähler und Nenner zugeschaltet worden ist. Die Normalisierungsstrategie kann durch diverse Schalter und Befehle gesteuert werden. Allerdings erreicht man mit den Standardvorgaben bereits recht gute Ergebnisse, so daß wir zum Thema "Schalter" auf [3] oder [5] verweisen.

Die Steuerung der auszuführenden Rechnung wird über das Setzen einer Reihe von *Schaltern* und Parametern ausgeführt. Diese unterteilen sich in

- Traceschalter (time),
- Schalter zur Ausgabeformatierung (nat),
- Schalter zur Zahlbereichsauswahl (rational, rounded, complex, modular) und
- Schalter zur Ablaufsteuerung.

Schalter sind reservierte Schlüsselworte, die ihre Bedeutung mit `on` < *switch* >; und `off` < *switch* >; ändern. Für die Bedeutung einzelner Schalter sei ebenfalls auf [3] oder [5] verwiesen. Der Status der wichtigsten Schalter kann mit dem Aufruf

```
> switches;
```

aus dem Paket `assist` (vorher laden!) ausgegeben werden.

Elementare Rechnungen folgen einer ähnlichen Syntax wie auch in den bisher vorgestellten CAS und führen zu ähnlichen Ergebnissen. Man beachte jedoch, daß Reduce, wenigstens in der Standardeinstellung, nicht zwischen Groß- und Kleinschreibung unterscheidet.

```
> 1/5 + 1/2; 2/3*(1/4+3/5); (1/2+1/3)^(-1);
```

$$\frac{7}{10}$$

$$\frac{17}{30}$$

$$\frac{6}{5}$$

```
> v:=1/2+E^2/6;
```

```
  2
 e  + 3
--------
   6
```

Im Gegensatz zu den beiden anderen Systemen verfolgt Reduce bei der Auswahl des Grundbereichs, über welchem die Rechnungen erfolgen sollen, das Konzept des *aktiven Zahlbereichs*, aus dem Koeffizienten (u.a.) für Standardquotienten gewählt werden. Standardmäßig sind das die ganzen Zahlen. In andere Zahlbereiche kann durch Schalter umgeschaltet werden.

Zahlbereich	Schalter
Ganze Zahlen	Standard
rationale Zahlen	`on rational;`
Floatzahlen	`on rounded;`
exakte komplexe Zahlen	`on complex;`
komplexe Floatzahlen	`on complex,rounded;`
modularer Zahlbereich	`on modular; setmod <module>;`
algebraische Zahlen	siehe arnum-Paket

Tabelle 3 : Verschiedene Zahlbereiche in Reduce

So liefert etwa

```
> on rounded; v; (v-sqrt(3)); off rounded;
```

$$1.73150934982$$

$$-0.000541457747102$$

Nach dem Rückschalten sind wir wieder im Bereich exakter Rechnungen, wobei selbst Zahlen in Float-Notation als ganzzahlige Brüche dargestellt werden. Im folgenden haben wir außerdem die Eigenart von Reduce verwendet, daß das Argument einer einstelligen Funktion nicht in Klammern gesetzt werden muß.

```
> sqrt 4; sqrt 6; sqrt 12; 1.78;
```

$$2$$

$$sqrt(6)$$

$$2 * sqrt(3)$$

$$\frac{89}{50}$$

Der Befehl `precision` $< nr >$; erlaubt es, die Genauigkeit reeller Approximationen einzustellen. Es wird die vorher eingestellte Genauigkeit zurückgegeben (Standard: 12).

```
> on rounded; precision 55; sqrt(2); precision nil; off rounded;
```

12

1.414213562373095048801688724209698078569671875376948073

55

Rechnungen mit ganzen Zahlen lassen sich mit einem ähnlichen Arsenal an Funktionen wie oben bewältigen, wobei die Unterschiede größer als zwischen Maple und MuPAD sind. So ist etwa `gcd` nur eine zweistellige Funktion und `!` darf generell nicht als Funktionssymbol verwendet werden, weil es in Reduce ein Sonderzeichen ist, das die spezielle Bedeutung des nachfolgenden Zeichens aufhebt. Stattdessen muß die Funktion `factorial(n)` explizit verwendet werden. `factor` dient zur Ausgabeformatierung und hat damit eine andere Bedeutung als erwartet. Stattdessen muß `factorize(p)` zur Faktorisierung einer ganzen Zahl oder eines polynomialen Ausdrucks verwendet werden. Dabei wird aber eine Liste *aller* Faktoren mit Mehrfacheinträgen entsprechend der Vielfachheit zurückgegeben.

```
> u:=factorial(33); v:=factorize(u)$ length(v);
```

$$u := 8683317618811886495518194401280000000$$

67

Die Ausgabe der Liste selbst wurde unterdrückt. Wir können das Ergebnis prüfen, indem wir alle Listenelemente aus `v` aufmultiplizieren:

```
> u-(for each x in v product x);
```

$$0$$

Die beiden Befehle

```
> u:=factorial(40); v:=nextprime(u);
```

funktionieren wie in den beiden anderen Systemen. Jedoch kann man die Primzahleigenschaft nicht so einfach testen. Da der algebraische Modus nur ein Interface zum eigentlichen Kern darstellt, der in einer anderen Sprache geschrieben ist, unterscheidet Reduce genau zwischen algebraischen und booleschen Operatoren. Erstere geben einen algebraischen Ausdruck, also in der Regel einen s.q., zurück, letztere einen booleschen Wert, der nur innerhalb von Verzweigungen oder Schleifentests ausgewertet werden kann. Eine direkte algebraische Auswertung oder Zuweisung zu einer Variablen ist nicht möglich. Die zur Verfügung stehenden *booleschen Operatoren* sind neben den booleschen Infixoperatoren insbesondere

`EVENP(U)`	bestimmt, ob die Zahl U gerade ist,
`FIXP(U)`	bestimmt, ob der Ausdruck U zu einer ganzen Zahl auswertet,
`FREEOF(U,V)`	bestimmt, ob der Ausdruck U den Kern V nicht enthält,
`NUMBERP(U)`	bestimmt, ob der Ausdruck zu einer Zahl des gerade aktiven Zahlbereichs auswertet,
`ORDP(U,V)`	bestimmt, ob U oberhalb V in der internen bzw. vereinbarten Ordnung der Kerne steht,
`PRIMEP(U)`	bestimmt, ob U prim ist.

Wir können also schreiben

```
> if primep v then yes else no;
```

$$yes$$

wobei das Ergebnis *yes* eine (von uns vereinbarte) symbolische Variable dieses Namens ist.

4.1 Datenstrukturen

Die in MuPAD und Maple möglichen Manipulationen mit Listen und Ausdruckssequenzen sind typische Vorgehensweisen von LISP und in Reduce nur im symbolischen Modus möglich. Das Hauptobjekt zur Aggregation verschiedener Daten sind aber auch im algebraischen Modus von Reduce (möglicherweise geschachtelte) *Listen.* Beispiele :

```
{a,b,c}
{1,a-b,c=d}
{{a},{{b,c},d},e}.
{}  % Die leere Liste
```

Folgende Listenoperationen stehen zur Verfügung :

PART(<list>,n)	der n-te Listeneintrag
LENGTH(<list>)	die Länge der Liste
FIRST(<list>)	das erste Listenelement
SECOND(<list>)	das zweite Listenelement
THIRD(<list>)	das dritte Listenelement
REST(<list>)	die Liste ohne das erste Element
<element> . <list>	an den Kopf der Liste wird ein Element angefügt
APPEND(<list>,<list>)	Aneinanderhängen zweier Listen
REVERSE(<list>)	Aufreihung der Listenelemente in umgekehrter Reihenfolge

Die wichtigsten Datentypaggregate in Reduce sind *Operatoren, Felder* und als spezielle Instanz letzterer *Matrizen.* Sie stellen die Möglichkeit zur Verfügung, auf indizierte Variablen zurückzugreifen. Solche Deklarationen sind stets, auch wenn sie innerhalb von Blöcken erfolgen, *globaler* Natur.

Felder haben dabei eine definierte Parameterzahl und durch nichtnegative ganzzahlige Werte beschränkte Indexbereiche. Bei ihrer Definition werden die Feldelemente mit 0 vorinitialisiert. Sie können also nicht für sich selbst stehen. Um dies zu erreichen muß man eine Operatordeklaration verwenden. *Matrizen* sind spezielle Arrays mit 2 Parametern.

```
array a(10),b(2,3,4);
matrix m(4,2);
```

definiert a und b als Arrays, deren Indizes jeweils von 0 bis zur angegebenen Grenze reichen. Auf einzelne Elemente kann man z.B. mit A(2) oder, sofern x zu einer ganzen Zahl zwischen 0 und 3 auswertet, auch mit B(1,x,2) zugreifen. Als Array deklarierte Bezeichner können nicht als gewöhnliche Variablen oder Prozedurnamen usw. verwendet werden. Jedoch ist eine Redeklaration als Array (auch anderer Dimension oder Arität) möglich.

Operatoren erlauben es, eigene Funktionssymbole zu vereinbaren und ihnen gewisse Eigenschaften zuzuweisen.

```
> operator f;
```

definiert etwa das Symbol f zum Funktionssymbol, so daß danach Ausdrücke der Form $f(x), f(2)$ usw. gebildet werden können.

Solche Operatoren können zu Infixoperatoren, zu linearen Funktionen, zu nichtkommutativen Operatoren und als abhängig von gewissen Variablen definiert werden. So wird etwa die Ableitung df(f,x) im Normalfall zu 0 auswerten, weil f ein von x unabhängiges, also *konstantes* Symbol ist. Ist jedoch f ein Funktionssymbol, das als von x abhängig vereinbart ist, so bleibt df(f,x) als neues Funktionssymbol für die Ableitung f' unausgewertet stehen. Für Details sei wiederum auf das Handbuch verwiesen.

Matrizen sind spezielle zweidimensionale Arrays. Matrizendeklarationen sind auf zwei Arten möglich.

```
matrix m(3,5);
```

deklariert m als 3×5-Matrix und initialisiert die Elemente mit 0.

```
m:=mat((a,b,c),(d,e,f));

     [a  b  c]
m := [       ]
     [d  e  f]
```

gibt eine Matrix unmittelbar ein. `m(i,j)` greift auf das entsprechende Matrixelement zu. Eine 3×3-Hilbertmatrix kann man z.B. definieren durch

```
matrix h(3,3);
for i:=1:3 do for j:=1:3 do h(i,j):=1/(i+j)$
h;

[ 1    1    1 ]
[---  ---  ---]
[ 2    3    4 ]
[             ]
[ 1    1    1 ]
[---  ---  ---]
[ 3    4    5 ]
[             ]
[ 1    1    1 ]
[---  ---  ---]
[ 4    5    6 ]
```

Auf Matrizen sind neben den üblichen arithmetischen Operationen einer Algebra über den arithmetischen Ausdrücken von Reduce weitere Operationen definiert (m steht dabei für eine Matrix) :

`length m`	Dimension von m (als Zweierliste)
`det m`	Determinante von m
`mateigen(m,x)`	Eigenwerte und Eigenvektoren von m.
`tp m`	transponiert m
`trace m`	Spur von m

Tabelle 4 : Operatoren auf Matrizen

Diese sowie eine Reihe von Funktionen der linearen Algebra sind nicht extra zu importieren. Für die Matrix

```
a:=mat((x-1,x+1),(x+1,x-1));

     [x - 1  x + 1]
a := [            ]
     [x + 1  x - 1]
```

erhalten wir außerdem mit

```
b:=a^2; c:=a^(-1);

     [    2           2     ]
     [2*(x  + 1)  2*(x  - 1)]
b := [                      ]
     [    2           2     ]
     [2*(x  - 1)  2*(x  + 1)]



     [  - x + 1     x + 1   ]
     [----------   -------  ]
     [   4*x         4*x    ]
c := [                      ]
     [  x + 1       - x + 1 ]
     [ -------    ----------]
     [   4*x         4*x    ]
```

die Ergebnisse gleich in normalisierter Form.

4.2 Steuerstrukturen

Reduce lehnt sich hier sehr stark an die Syntax von Pascal an. Neben einfachen Anweisungen spielen *Verbundanweisungen* eine wichtige Rolle, die man auf zwei Arten definieren kann, als *Gruppenanweisung*

```
<< anw_1; anw_2; ... ; anw_n >>
```

oder als *Block*

```
BEGIN
[<lokale Variablendeklaration>]
<Statements, getrennt durch Terminatoren>
END
```

Der Block ist dabei die allgemeinere Struktur, da er neben (durch `scalar` $a, b, c, \ldots$; zu vereinbarenden) lokalen Variablen auch `return`-Anweisungen enthalten kann.

Eine *Verzweigung* hat die Syntax

```
IF <boolean expression> THEN <statement> [ELSE <statement>]
```

Die Zweige bestehen wie in Pascal aus jeweils genau einer Anweisung, die auch eine Verbundanweisung sein kann.

Eine *Schleife* hat die Syntax

```
WHILE <boolean expression> DO <statement>
```

oder

```
REPEAT <statement> UNTIL <boolean expression>
```

Man beachte, daß in beiden Fällen nur eine (Verbund-)Anweisung (ohne Terminator) im Schleifenkörper steht.

In Reduce liefert jede Anweisung einen Wert zurück. So sind Formeln der Art

```
> max:=if a<b then b else a;
```

möglich.

Das wichtigste Schleifenkonstrukt ist jedoch die *for-Schleife*, die in zwei grundlegend verschiedenen Formen zur Verfügung steht:

> `for` $<id>$`:=`$<nr>$`:`$<nr>$ $<action>$ $<statement>$
> oder
> `for each` $<id>$ `in` $<liste>$ $<action>$ $<statement>$

wobei

```
<action> ::= do|product|sum|collect|join.
```

Die erste Alternative definiert eine Iteration über einen gewissen Bereich einer Laufvariablen, während die untere Alternative über alle Elemente einer Liste iteriert. Laufvariablen sind dabei bzgl. der FOR-Schleife (selbstdeklarierende) *lokale* Variablen, d.h. verdecken gleichnamige Variablen aus der Umgebung.

Die einzelnen Aktionsoptionen bedeuten folgendes :

do	wertet die entsprechenden Ausdrücke aus, ohne sich diese Werte in einer schleifeninternen Variablen zu merken
collect	wertet die Ausdücke aus und sammelt sie in einer Liste auf
join	die Ausdücke liefern als Wert selbst wieder Listen zurück, die aneinandergehängt werden
sum	die (algebraischen) Ausdrücke werden addiert
product	die (algebraischen) Ausdrücke werden multipliziert

Dabei wird außer bei *do* der ermittelte Wert als Wert des FOR-Ausdrucks zurückgegeben. Wie in PASCAL wird die Schleife nicht ausgeführt, wenn die Zählvariable bereits bei der Initialisierung außerhalb des Zählbereichs liegt.

Beispiele :

```
array a(6),b(6);
for i := 2 : 5 do
        <<a(i):=i^2; b(i):=i^3; write(a(i)," - ",b(i));>>;

4 - 8

9 - 27

16 - 64

25 - 125

for i := 1 : 6 collect a(i);

{0,4,9,16,25,0}
```

Die erste Schleife speichert die entsprechenden Quadrate und Kuben in den vorher deklarierten Feldern a und b unter Verwendung einer Gruppenanweisung. Die zweite Schleife konstruiert eine Liste, wobei die nichtbelegten Feldelemente ihren Initialwert 0 haben.

Zur Verwendung von *sum* und *product* :

```
c := for j:=1:9 sum j^2;

        c := 285
```

```
on factor; % Schaltet auf faktorisierte Normalformen um.
d := for k:=0:4 product (x+k);

        d := (x + 4)*(x + 3)*(x + 2)*(x + 1)*x

clear c,d;
```

Zur Verwendung von Iteratoren auf Listen :

```
for each x in {a,b,c} collect x^2;

  2  2  2
{a ,b ,c }

for each x in {a,b,c} collect {x^2};

   2    2    2
{{a },{b },{c }}

for each x in {a,b,c} join {x+1,x-1};

{a + 1, a - 1, b + 1, b - 1, c + 1, c - 1}
```

Die Join-Option kann besonders zur *selektiven Listenauswahl* verwendet werden. Durch Aneinanderhängen von Listen der Längen 1 und 0 sammelt die folgende Schleife unter den ersten 50 Zahlen nur die Primzahlen auf :

```
for i := 1:50 join if primep(i) then {i} else {};

{2,3,5,7,11,13,17,19,23,29,31,37,41,43,47}
```

Funktionen lassen sich ebenfalls ähnlich wie in Pascal definieren:

```
[algebraic] procedure <proc-name>(<Parameter>);
            <statement>;
```

Als Statement kommt eine einzelne Anweisung, eine Gruppen- oder eine Blockanweisung in Frage. Kompliziertere Funktionen werden gewöhnlich durch eine Blockanweisung definiert, da hier lokale Variable und Return-Werte verfügbar sind. Als Beispiel betrachten wir wieder die Berechnung des gcd mit dem Euklidschen Algorithmus:

```
procedure ourgcd(a,b);
     if b=0 then a else begin write("gcd(",a,",",b,")"); ourgcd(b,a mod b) end;
```

4.3 Der `Solve`-Operator

Der `Solve`-Operator von Reduce gibt nach einheitlichen Grundsätzen für eine Gleichung eine Liste der Lösungen und für ein Gleichungssystem eine Liste der Lösungstupel zurück.

```
> s:=solve((p:=x^2-3*x+2),x);


   s := {x=2,x=1}
```

Die Probe ergibt wieder die Korrektheit der Rechnung.

```
> for each u in s collect sub(u,p);


  {0,0}
```

Dabei werden standardmäßig nur quadratische Irrationalitäten verwendet. Alle anderen algebraischen Zahlen werden durch `root_of`-Ausdrücke dargestellt. Dies kann man durch den Schalter `fullroots` (Standard: off) beeinflussen. Stellt man ihn um, so wird die trigonometrische Form der Nullstellen von Gleichungen höheren Grades verwendet. Durch Setzen eines weiteren Schalters `trigform` (Standard: on) kann man zur gewohnten Form in Radikalen übergehen.

```
> s:=solve((p:=x^5-x+1),x);

                     5
  s := {x=root_of(x_  - x_ + 1,x_,tag_4)}
```

Dieses Symbol expandiert etwa bei numerischer Nähreung:

```
> on rounded; s; expand_cases(s); off rounded;


  {x=one_of({0.764884433601 + 0.352471546032*i,
             0.764884433601 - 0.352471546032*i,
              - 0.18123244447 + 1.08395410132*i,
              - 0.18123244447 - 1.08395410132*i,
              - 1.16730397826},tag_4)}

  {x=0.764884433601 + 0.352471546032*i,
   x=0.764884433601 - 0.352471546032*i,
   x= - 0.18123244447 + 1.08395410132*i,
   x= - 0.18123244447 - 1.08395410132*i,
   x= - 1.16730397826}
```

Für Systeme von Gleichungen kann man ähnlich vorgehen:

```
> s:=solve((u:=x^2+y=1,x-y=2),{x,y});


            sqrt(13) - 5     sqrt(13) - 1
  s := {{y=--------------,x=--------------},
                 2                2

             - sqrt(13) - 5      - (sqrt(13) + 1)
        {y=-----------------,x=-------------------}}
                   2                    2
```

Für die Probe beachte man, daß Reduce standardmäßig die linke Seite von Gleichungen nicht auswertet. Deshalb ergibt

```
> for each x in s collect sub(x,u);


    2                  2
  {{x  + y=1,x - y=2},{x  + y=1,x - y=2}}
```

nicht das gewünschte Ergebnis. Stattdessen kann man den Schalter `evallhseqp` (Standard: off) ändern oder aber einfach prüfen, ob die gefundenen Werte Nullstellen der Differenz zwischen linker und rechter Seite jeder der Gleichungen aus u sind:

```
> for each x in s collect sub(x,for each y in u collect lhs y - rhs y);


   {{0,0},{0,0}}
```

Zur automatischen Lösung trigonometrischer Systeme verfügt Reduce nur über wenig Kenntnisse. So liefert es für das trigonometrische System

```
> s:=solve((u:=sin(x)-1/2),x);


             pi*(12*arbint(2) + 5)     pi*(12*arbint(2) + 1)
   s := {x=-----------------------,x=-----------------------}
                       6                          6
```

zwar die vollständige und korrekte Lösung, aber eine Probe führt nur für spezielle Werte zum Erfolg:

```
> for each x in s collect sub(x,u);

            12*arbint(2)*pi + 5*pi
     2*sin(------------------------) - 1
                      6
   {--------------------------------------,
                      2

             12*arbint(2)*pi + pi
     2*sin(----------------------) - 1
                     6
    ------------------------------------}
                     2


> s1:=for i:=-5:5 join sub(arbint(2)=i,s)$

> for each x in s1 collect sub(x,u);


   {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0}
```

Für das trigonometrische System, das wir mit den anderen CAS untersucht haben, erhalten wir ein wenig brauchbares Resultat, das offensichtlich mit dem verwendeten Lösungsverfahren für solche trigonometrischen Gleichungen zu tun hat.

```
> s:=solve((u:=sin(x)-cos(x)),x);


   s := {x=2*(arbint(3)*pi + atan(sqrt(2) - 1)),

         x=2*(arbint(3)*pi - atan(sqrt(2) + 1))}
```

Erst nachdem man den Schalter `rationalize` (Default: off) zugeschaltet hat, bekommt man als akzeptable Lösung

```
> on rationalize; s;


       8*arbint(3)*pi + pi      8*arbint(3)*pi - 3*pi
   {x=--------------------,x=----------------------}
               4                        4
```

Die Begründung für diesen Effekt ist jedoch recht subtil. Allgemein läßt sich sagen, daß Reduce recht bescheidene Möglichkeiten zum Lösen goniometrischer Gleichungen besitzt, jedoch relativ einfach vom Nutzer durch ein entsprechendes System von Vereinfachungsregeln ergänzt werden kann.

4.4 Lokale Substitutionsregeln

In Reduce erfolgt die Vereinfachung von Ausdrücken mit Funktionssymbolen auf besonders einfache und übersichtliche Art und Weise. Mit jedem Funktionssymbol ist eine Liste von Simplifikationsregeln verbunden, die beim Auftreten dieses Symbols in einem Ausdruck nach passenden Mustern durchsucht und so lange angewendet werden, bis keine Simplifikation mehr möglich ist. Mit `showrules` $< FSymb >$; kann man diese Regeln extrahieren. So liefert etwa

```
> showrules log;

  {log(1) => 0,
  log(e) => 1,
  log(e**~x) => x,
  df(log(~x),~x) => 1/x}$
```

die dem System bekannten Regeln für das Vereinfachen von Logarithmus-Ausdrücken mit speziellem Argument oder aber spezieller Form. Die ersten beiden Regeln sind nur für die speziellen Symbole $log(1)$ und $log(e)$ anwendbar, während die letzten beiden Formeln *Muster* sind, in denen x als Platzhalter für einen beliebigen Ausdruck steht, der dann auch für alle Instanzen von x in dieser Regel einzusetzen ist. Solche Variablen sind durch eine Tilde gekennzeichnet.

Solche Regeln können auch vom Nutzer definiert und *lokal* angewendet werden und liefern damit eine alternative Möglichkeit zum Substitutionsoperator `sub`. Eine solche *Regelliste* ist eine Liste von Regeln der Form

```
<expression> => <expression> (WHEN <boolean expression>)
```

So definiert etwa die Liste

```
trig0:={cos(~x)*cos(~y) => (cos(x+y)+cos(x-y))/2,
        cos(~n*pi)      => (-1)^n when remainder(n,2)=0};
```

ein Regelsystem, das Produkte von Kosinus-Funktionen in Summen von Kosinus-Funktionen verwandelt. In beiden Formeln stehen x, y, n als Mustervariablen, worauf wieder die Tilde hinweist. Die zweite Regel wird nur dann angewendet, wenn die Nebenbedingung erfüllt, d.h. n eine gerade ganze Zahl ist.

Solche Regeln können global vereinbart werden (siehe Handbuch) oder aber lokal durch eine `where`-Anweisung angewendet werden.

```
> cos(x+y)*cos(x-y) where trig0;
```

```
 cos(2*x) + cos(2*y)
---------------------
          2
```

Hier werden die Mustervariablen also durch entsprechende Ausdrücke $x+y$ und $x-y$ ersetzt.

```
> cos(x)*cos(y)*cos(z) where trig0;
```

```
 cos(x - y - z) + cos(x - y + z) + cos(x + y - z) + cos(x + y + z)
-------------------------------------------------------------------
                                4
```

Das Ergebnis erhält man durch mehrfache Anwendung der Regeln, wobei als Zwischenschritt

```
 cos(x - y) * cos(z) + cos(x + y) * cos(z)
-------------------------------------------
                    2
```

entstanden ist, worauf die Regel noch einmal anwendbar ist. Die generelle Syntax einer `where`-Anweisung ist

```
<expression> where <rule list>;
```

Man beachte, daß die Bindungskraft des Zuweisungsoperators (derzeit) stärker als die von `where` ist und damit etwa

```
> u:=cos(x)*cos(y) where trig0; u;
```

```
 cos(x - y) + cos(x + y)
-------------------------
            2


cos(x)*cos(y)
```

ein auf den ersten Blick überraschendes Ergebnis liefert. Die Eingabe wird jedoch als

```
> (u:=cos(x)*cos(y)) where trig0;
```

interpretiert. Das gewünschte Ergebnis liefert

```
> u:=(cos(x)*cos(y) where trig0); u;
```

```
      cos(x - y) + cos(x + y)
u := -------------------------
                 2


 cos(x - y) + cos(x + y)
-------------------------
            2
```

Allerdings wird

```
> cos(x)^2 where trig0;


       2
cos(x)
```

dabei noch nicht in eine Summe transformiert, da dieser Ausdruck nicht die Form eines Produkts, sondern einer Potenz hat. Ein kompletteres Regelsystem wäre etwa

```
trig1 := {cos(~x)*cos(~y) => (cos(x+y)+cos(x-y))/2,
          cos(~x)*sin(~y) => (sin(x+y)-sin(x-y))/2,
          sin(~x)*sin(~y) => (cos(x-y)-cos(x+y))/2,
          cos(~x)^2       => (1+cos(2*x))/2,
          sin(~x)^2       => (1-cos(2*x))/2};
```

mit dem sich auch Potenzen in Summen zerlegen lassen.

```
> cos(x)^2 where trig1; cos(x)^6 where trig1;


 cos(2*x) + 1
--------------
      2


cos(6*x) + 6*cos(4*x) + 15*cos(2*x) + 10
------------------------------------------
                    32
```

Solche Regelsysteme können recht komplex sein. Betrachten Sie dazu etwa die Systemregeln für die Sinus-Funktion.

```
> showrules sin;
```

Zur Reihenfolge der Abarbeitung der Regeln sowie die Konstruktion komplizierterer boolescher Nebenbedingungen mit `when` sei auf das Handbuch verwiesen.

Literatur

[1] K. Drescher, F. Postel, and T. Schulze. Advanced demonstrations with MuPAD 1.3. Automath technical report, Univ.-GH Paderborn, 1997.

[2] B. Fuchssteiner et al. *MuPAD: Multi Processing Algebra Data Tool. Version 1.2.2.* Birkhäuser, 1993. As online version part of the MuPAD distribution.

[3] H.-G. Gräbe. Skript zum Reduce-Praktikum (SS 1994). Univ. Leipzig, 1994.

[4] D. Gruntz, K. Meier, and M. Monagan. Maple - An Introduction. Manuskript, ETH Zürich, 1992.

[5] A.C. Hearn and J.P. Fitch. *Reduce User's Manual 3.6.* Konrad-Zuse-Zentrum, Berlin, 1995.

[6] C. Heckler, F. Postel, G. Siek, and S. Wehmeier. Introduction to the computer algebra system MuPAD - a tutorial approach. Vorabversion, September 1997, 125 Seiten.

[7] F. Postel. A demonstration tour through MuPAD 1.3. Automath technical report, Univ.-GH Paderborn, 1997.

[8] A. Sorgatz. MuPAD-Schnellkurs. http://www.mupad.de/DEMO/basics.html, 1995.

[9] A. Walz. *Maple V - Rechnen und Programmieren mit Release 4.* Oldenbourg, 1998.