Scriptum zum REDUCE -Praktikum

Dr. H.-G. Gräbe
Institut für Informatik

11. März 1994

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Was ist Computeralgebra ?

Mathematik ist ebenso wie die anderen Naturwissenschaften eine sehr experimentelle Wissenschaft, wenn auch ihre Experimente mit "Bleistift und Papier" von besonderer Art sind. Jedoch auch für sie gilt, daß nur das Aufspüren von Gesetzmäßigkeiten in einer Fülle empirischen Materials den Weg zu neuen Einsichten weist. Es ist deshalb nicht verwunderlich, daß die überwiegende Zahl mathematischer Problemlösungen von aufwendigen Berechnungen begleitet sind, die nur selten darin gipfeln, daß sich neue Erkenntnisse in Form von mathematischen Sätzen formulieren (und beweisen) lassen. In Zeiten, wo elementare Rechenfertigkeiten noch nicht zur Allgemeinbildung gehörten, genossen Rechenmeister wie A. Ries, die sich darüberhinaus die Verbreitung mathematischen Wissens auf ihre Fahnen geschrieben hatten, großes Ansehen. Heutige Anwendungen der Mathematik verwenden einen wesentlich größeren technischen Apparat, der qualifiziert angewendet sein möchte.

Berühmte Mathematiker, ob nun Euler, Gauß, Fermat, die Gebrüder Bernoulli oder andere, waren nicht nur geniale Zeitgenossen, sondern stets auch hervorragende Rechenkünstler. So hat Gauß 1801 im Alter von 24 Jahren seinen Ruf als herausragender Mathematiker dadurch gefestigt, daß er die Bahn des gerade entdeckten, aber sonst in keiner Weise bedeutenden Planetoiden Ceres berechnet hat. Gauß unternahm diese Berechnungen, nachdem er bereits seine (heute) berühmten *Disquisitiones Arithmeticae* veröffentlicht hatte.

Im Zuge der weiteren Entwicklung, insbesondere im 20. Jahrhundert, verschob sich der Stil mathematischer Forschung stärker hin zu qualitativen Aussagen. Mitentscheidend dafür könnte die Tatsache gewesen sein, daß die Berechnungen allmählich zu lang und zu kompliziert wurden und sich nicht mehr ohne Hilfsmittel sinnvoll ausführen ließen. Jedoch sind auch heute mit umfangreichen Berechnungen verbundene Anwendungen der Mathematik eher die Regel als die Ausnahme. Noch vor wenigen Jahrzehnten prägten deshalb Tafeln, Rechenstab, Nomogramme und andere einfache Rechenhilfsmittel das Berufsbild des Ingenieurs. Mit der Erfindung moderner digitaler Computer sind hier neue Horizonte eröffnet worden.

Historisch wurde das Wort Computer zuerst mit einer Maschine zur schnellen Ausführung numerischer Rechnungen verbunden. Damit sind nicht nur einfache arithmetische Operationen gemeint, sondern auch kompliziertere Anwendungen wie das Berechnen numerischer Werte mathematischer Funktionen, die Approximation von Nullstellen gegebener Polynome oder von Eigenwerten gegebener Matrizen. All diesen Anwendungen ist gemein, daß sie sich, unter Verwendung ausgefeilter Programmiersprachen, letzlich allein auf das Rechnen mit (Compu-

ter)zahlen zurückführen lassen. Sowohl von der Aufgabenstellung her als auch der Methodik wird dabei mit Näherungswerten gerechnet und die Frage der Relevanz der erzielten Ergebnisse (Fehlertoleranzen) ist gesondert zu untersuchen. Solche *numerischen Anwendungen*, insbesondere aus der Hochenergiephysik, der Quantenchemie und anderen stark mathematisierten Gebieten der Naturwissenschaften sind auch heute noch der Hauptposten in der Auslastung der "mainframes", großer und moderner Rechensysteme. Sie sind Gegenstand einer ganzen Forschungsrichtung im Grenzgebiet zwischen Mathematik und Informatik, dem *wissenschaftlichen Rechnen*, die in Deutschland mit mehreren wichtigen Lehrstühlen und einem großen Institut, dem Konrad-Zuse-Zentrum in Berlin, vertreten ist.

Betrachtet man die Möglichkeiten des Computers näher, so erkennt man, daß mit diesen numerischen Verfahren seine Potenzen keineswegs ausgereizt sind. Spätestens seit Turing ist bekannt, daß ein moderner digitaler Computer als eine *universelle* Maschine zu verstehen ist, die *jeden beliebig gearteten Algorithmus*, d.h. eine exakt spezifizierte Prozedur, auszuführen in der Lage ist.

Ein weiteres großes Einsatzgebiet hat sich der Computer im Bereich der Textverarbeitung erschlossen. Mit dem Siegeszug der Kleinrechentechnik in den letzten 20 Jahren entwickelte er sich dabei vom Spielzeug und der erweiterten Schreibmaschine hin zu einem unentbehrlichen Werkzeug der Büroorganisation, wobei vor allem seine Fähigkeit, (geschriebene) Information speichern, umordnen und verändern zu können, eine zentrale Rolle spielt. In diesem Anwendungssektor kommt also die Fähigkeit des Computers zum Einsatz, *symbolische Information* zu verarbeiten. In der Hand des Ingenieurs und Wissenschaftlers entwickelt er sich dabei zu einem Instrument, das nicht nur den Rechenschieber, sondern auch zunehmend Formelsammlungen abzulösen in der Lage ist. Wissen kann dabei in datenbankähnlichen Strukturen viel effizienter verwaltet werden als bisher. Auf diesem Niveau handelt es sich allerdings vorerst um eine syntaktische Verarbeitung, wo der Computer den *Sinn der Information* noch nicht in die Verarbeitung einzubeziehen vermag.

Mathematische Berechnungen haben gewöhnlich neben der rein numerischen Komponente eine weitere wichtige Seite, die wir *symbolische und algebraische Berechnungen* nennen wollen. Es geht auch hier um die Umformung symbolischer Information nach mehr oder minder komplizierten Regeln, aber unter Einbeziehung eines gewissen Verständnisses der damit verbundenen Semantik. Kurz gesprochen kann man dieses Gebiet als *das Rechnen mit Symbolen* bezeichnen, die mathematische Objekte repräsentieren. Diese Objekte können neben ganzen, rationalen, reellen oder komplexen Zahlen (beliebiger Genauigkeit) auch algebraische Ausdrücke, Polynome, rationale Funktionen, Gleichungssysteme oder sogar noch abstraktere mathematische Objekte wie Gruppen, Ringe, Algebren und deren Elemente sein. Mehr noch, das Adjektiv *symbolisch* bedeutet dabei, daß das Ziel der mathematischen Problemstellung nicht die Suche nach numerischen Werten, sondern nach einer geschlossenen oder approximativen Formel im Sinne des deduktiven Mathematikverständnisses ist. *Algebraisch* bedeutet, daß eine *exakte* mathematische Ableitung aus den Ausgangsgrößen durchgeführt wird, anstatt näherungsweise Fließkommaarithmetik einzusetzen. Beispiele für solche algebraisch-symbolischen Umformungen sind die Polynomfaktorisierung, die (unbestimmte) Differentiation und Integration, die Reihenentwicklung von Funktionen, analytische Lösungen von Differentialgleichungen, exakte Lösungen polynomialer Gleichungssysteme oder die Simplifikation mathematischer Formeln.

In den letzten 25 Jahren hat man große Fortschritte in der theoretischen Fundierung dieser symbolischen Methoden erreicht. Es sind Werkzeuge entstanden, die es erlauben, ein großes Stück mathematischen Know-hows Anwendern als black box unmittelbar zugänglich zu ma-

chen. Diese junge, sich stark entwickelnde Disziplin im Grenzgebiet zwischen Mathematik und Informatik hat noch keinen traditionellen Namen : symbolisches Rechnen, Formelmanipulation oder Computeralgebra sind einige im deutschen Sprachraum gängige Bezeichnungen. Wir wollen im folgenden zur Bezeichnung dieses Gebiets die letztere bemühen.

R. Loos definiert Computeralgebra als den Teil der Computer Science (ein Wort, das man wohl nur sehr unvollständig mit "Informatik" ins Deutsche übertragen kann), der sich mit *Design, Analyse, Implementierung und Anwendung algebraischer Algorithmen* beschäftigt. Etwas pessimistischer formuliert derselbe Autor, daß Computeralgebra solche Probleme zum Gegenstand hat,

> die die Algebra als zu rechenintensiv
>
> und die die Informatik als zu algebraisch

zurückweist. In diesem Spannungsfeld zwischen Mathematik und Informatik findet die Computeralgebra aber zunehmend ihren eigenen Platz. Sie nimmt damit zugleich eine wichtige Stellung für Entwicklungsimpulse aus beiden Gebieten ein. So mag es nicht verwundern, daß große Durchbrüche der letzten Jahre sowohl in der Mathematik als auch in der Informatik die von der Computeralgebra produzierten Werkzeuge wesentlich beeinflußt haben und umgekehrt.

Die von der Computeralgebra produzierten Werkzeuge spielen eine stark zunehmende Rolle sowohl in den Natur- als auch in den Ingenieurwissenschaften. Eine Liste von thematischen Schwerpunkten des RISC Linz, des von Prof. Buchberger geleiteten Research Institute for Symbolic Computation an der Universität Linz (Österreich) aus dem Jahr 1990 mag dies belegen :

- automatische Geometriebeweise
- Roboterprogrammierung
- computergestützte Produktion (Simulation und Steuerung von NC-Maschinen)
- computergestützte Molekularsynthese (aus Ingredienzien und möglichen Reaktionen)
- Simulation und Kontrolle von Tunnelvortrieb
- Expertensysteme für Fabrikationskontrolle (Lagerhaltung, Just-in-time-Produktion)

Für einen vollständigeren Überblick über Tendenzen und Anwendungen der Computeralgebra siehe [1].

1.2 Computeralgebrasysteme (CAS)

Die ersten Anfänge der Entwicklung von Programmen der Computeralgebra reichen bis in die frühen 50er Jahre zurück. So hat etwa H.G. Kahrimanian 1953 an der Temple University in Philadelphia eine Master Thesis zur analytischen Differentiation geschrieben, in deren Rahmen er auch ein Assemblerprogramm für die Univac I erstellte. Ende der 50er und Anfang der 60er Jahre wurden vor allem am MIT (Massachusetts Institute of Technology) große Forschungsanstrengungen unternommen, symbolisches Rechnen mit eigenen Hochsprachen zu entwickeln (Formula ALGOL, ABC ALGOL, ALADIN, ...). Von diesen Sprachen hat

sich bis heute vor allem LISP als die Grundlage für die meisten Computeralgebrasysteme erhalten.

In den 60er und 70er Jahren wurden erste Versuche verschiedener Autoren mit umfangreicheren symbolischen Programmpaketen auf der damals verfügbaren bescheidenen Hardware gemacht. Diese mündeten Ende der 70er Jahre in mehrere große Programmpakete, die in veränderter und weiterentwickelter Form, unter teilweise anderem Namen, auch heute noch die "Szene" wesentlich prägen. Sie sind, grob sortiert nach fallenden Hardwareanforderungen, in Tabelle 1 zusammengefaßt.

System	Autoren	Institution
Scratchpad →AXIOM	R. Jenks, M. Bronstein, J.H. Davenport, B. Trager, J. Grabmeier, S.M. Watt	IBM / NAG
MACSYMA	D. Armbruster	Macsyma Inc., Arlington
SMP →Mathematica	D. Cole, S. Wolfram, R. Maeder	Wolfram Research, Inc.
REDUCE	A. Hearn, M. MacCallum, H. Melenk, W. Neun, J.P. Fitch, F. Wright, H. Caprasse, E. Schrüfer u.a.	RAND Corporation, Santa Monica, Zuse-Zentrum Berlin
Maple	B.W. Char, K.O. Geddes, G.H. Gonnet, B. Leong, M.B. Monagan, S.M. Watt	University of Waterloo
muMATH-79 →DERIVE	A. Rich, D. Stoutemyer	Soft Warehouse Inc., Honolulu

Tabelle 1 : Die großen CAS allgemeiner Ausrichtung

In den letzten fünf bis zehn Jahren wurden alle diese Systeme, die bis dahin nur in einer mehr oder weniger experimentellen Fassung vorlagen, mit größerem Aufwand unter marktorientierten Gesichtspunkten weiterentwickelt. Schwerpunkt waren dabei vor allem die Einbindung von Windowsystemen, hypertextbasierte Hilfesysteme und leistungsfähige Graphikmoduln. Sie besitzen damit heute in der Regel eine ausgereifte Benutzeroberfläche, sind gut dokumentiert und auf den verschiedensten Plattformen (bis hin zu 386 PC) verfügbar. Zu den Systemen gibt es einführende Bücher und Bücher zum vertieften Gebrauch, für spezielle Anwendungen und zum Einsatz in der Lehre. Zudem erscheinen regelmäßig Nutzerinformationen und Informationen über frei zugängliche Anwenderpakete. Weiterhin haben sich Benutzergruppen gebildet, und es werden Anwendertagungen durchgeführt. Die Quelltexte der mathematischen Algorithmen sind bei AXIOM, MAPLE und REDUCE dem Benutzer zugänglich.

Zugleich entstanden und entstehen auch neue Systeme allgemeiner Ausrichtung. Da CAS jedoch einen hohen Entwicklungsaufwand erfordern (mehr als 100 Mannjahre Programmierarbeiten), sind diese neuen Systeme in der Regel nicht sehr leistungsfähig oder haben nur einen sehr eingeschränkten Anwendungsbereich.

Daneben gibt es eine große Anzahl kleinerer experimenteller Systeme mit eingeschränkter Ausrichtung. Sie wurden für symbolische Rechnungen auf sehr begrenzten Gebieten der Mathematik oder Physik entwickelt und erreichen durch speziell angepaßte Datenstrukturen und

Algorithmen oftmals eine erstaunliche Leistungsfähigkeit. Sie sind meist das Produkt kleiner Gruppen von Spezialisten, die das Topwissen ihres Fachgebiets mit guten Programmierkenntnissen verbinden konnten. Beispiele für solche Systeme sind in der Physik SCHOONSHIP, FORM (Hochenergiephysik), CAMAL (Himmelsmechanik), SHEEP und STENSOR (allgemeine Relativitätstheorie) sowie in der Mathematik Cayley und GAP (Gruppentheorie), PARI, KANT, SIMATH (Zahlentheorie), Aldes-SAC 2, Macaulay, AlPi, CoCoA, Singular, Felix, MAS, SAC, Bergman (kommutative und nichtkommutative Algebra) und LIE (Lietheorie).

Kapitel 2

Das Computeralgebrasystem REDUCE

Die erste Version von REDUCE wurde Ende der 60er Jahre von Anthony C. Hearn entwickelt und publiziert. Ausgangspunkt waren formale Rechnungen in der Hochenergiephysik (Feynman-Graphen, Wirkungsquerschnitte etc.), die sich von Hand nur schwierig durchführen lassen. REDUCE 2 (1970) wurde dann erstmals auch von anderen Wissenschaftlern genutzt und stellt so den eigentlichen Startpunkt des Entstehens einer REDUCE-Nutzergemeinde dar. REDUCE 3 (1983) enthielt verschiedene wesentlich neue Pakete wie z. B. für unbestimmte Integration, multivariate Polynomfaktorisierung, eine reelle Arithmetik beliebiger Genauigkeit und einen Gleichungslöser.

Heute wird REDUCE in einem kontinuierlichen Prozeß unter Federführung von A. Hearn von einem weltweiten System von Nutzern ständig weiterentwickelt und erneuert. Wie auch andere Systeme des symbolischen Rechnens profitiert REDUCE dabei von der stürmischen Entwicklung der Computerhardware und den Möglichkeiten, die sich aus einer weltweiten Vernetzung der wissenschaftlichen Arbeitsgruppen ergeben. Mit den Versionen REDUCE 3.1 bis 3.5, die in den letzten Jahren in etwa jährlichen Intervallen herausgegeben wurden, kamen weitere Nutzerpakete hinzu, wurden Fehler beseitigt und neue Programmiertechniken eingeführt. Im Oktober 1993 wurde REDUCE in der aktuellen Version 3.5 freigegeben.

REDUCE läuft auf einer Vielzahl von Rechnerplattformen, vom 386 DOS PC (mit wenigstens 4 MB) bis hinauf zum Cray Supercomputer. Die klassiche Plattform mit genügender Rechenleistung auch für größere Probleme sind jedoch UNIX-basierte Workstations.

REDUCE ist primär ausgerichtet auf die Lösung mathematischer Probleme im Bereich technisch-wissenschaftlicher Anwendungen, und zwar insbesondere auf größere Probleme mit sehr umfangreichen und schwierigen formalen Rechnungen. Dementsprechend finden sich zwar in REDUCE durchaus Operatoren, die eine Standardaufgabe wie etwa das Lösen kleinerer Gleichungssysteme oder die Berechnung einer Determinanten (mit symbolischen und/oder numerischen Einträgen) unmittelbar und vollständig lösen. Typischer ist jedoch die Verwendung von REDUCE auch zur analytischen Verarbeitung der erhaltenen Ergebnisse, was eine komplexe Kombination von Einzelschritten erfordert. Deshalb wurde bei der Entwicklung von REDUCE vor allem auf die Bereitstellung leistungsfähiger und modularer Operatoren Wert gelegt, die im Rahmen einer interaktiven Programmiersprache wie Bausteine zum Zusammenstellen einer Problemlösung genutzt werden können.

REDUCE ist damit, ähnlich wie die meisten anderen CAS allgemeiner Ausrichtung, mehr

eine Programmierumgebung, die algebraische Werkzeuge zur Verfügung stellt, als ein starrer algebraischer Kalkulator.

In Einzelfällen kann es sogar erforderlich sein, einen vollständig neuen symbolischen Kalkül in der internen Programmiersprache RLISP einzurichten. REDUCE bietet dafür einen reichen Satz von Schnittstellen auf mehreren Ebenen an, und die Bibliothek enthält zahlreiche Vorbilder. In neueren Versionen wird auch eine Modularisierung mit den zugehörigen Datenkapselungskonzepten stärker unterstützt, soweit dies in einer LISP-basierten Sprache überhaupt möglich ist.

REDUCE ist in RLISP geschrieben, einer an der Syntax von ALGOL-60 orientierten Sprache, die eine 1-1-Umsetzung in Standard Lisp (SL) erfährt. In REDUCE 1 war von A. Hearn LISP als Hostsprache verwendet worden, um die Portabilität auf verschiedene Rechnertypen zu erleichtern. Mit der Evolution der verschiedenen LISP-Dialekte war das nicht mehr gewährleistet, so daß im Umfeld von REDUCE SL als ein beschränkter Lisp-Sprachumfang entstand, der (nach möglicherweise kleinen Ergänzungen) von den meisten anderen Lisp-Dialekten auch unterstützt wird und für die Belange der Kodierung symbolischer Rechnungen ausreicht. Bis 1981 wurde SL vor allem dafür verwendet, RLISP auf einen von der Plattform unterstützten LISP-Dialekt abzubilden. Mit der Implemetierung eines eigenen Portable Standard LISP (PSL) Compilers wurde auch in dieser Richtung mehr Unabhängigkeit und Effizienz erreicht. Dies und die Tatsache, daß der LISP-Kern ob seiner Kompaktheit schnell auf andere Plattformen umsetzbar ist, sind die Hauptgründe dafür, daß REDUCE auf einer Vielzahl verschiedener Rechnertypen verfügbar ist, ohne dabei wesentliche Änderungen in den Quellen selbst vornehmen zu müssen. Dies ist auch der Grund, daß systemabhängige Funktionen wie z.B. Graphikteile und Online-Hilfesysteme, bisher von REDUCE nur ungenügend unterstützt werden. Mit der Standardisierung von Fensterumgebungen sind aber auch hier in den letzten Jahren neue Entwicklungen angestoßen worden.

Der LISP-Kern selbst wurde lange Zeit unverändert gelassen, hat jedoch mit RLISP-88 eine erste Erweiterung erfahren und wird voraussichtlich mit der Einführung modularer Datenkapselungskonzepte weiteren Modifikationen unterworfen. REDUCE enthält (im Profipaket) einen Compiler, der zum Interpreter weitgehend syntaktisch äquivalent ist und es auch dem Nutzer gestattet, eigene Prozeduren und Moduln in eine schnell ladbare Maschinenform zu verwandeln, die die Ausführung beschleunigt.

Als LISP-basiertes System ist RLISP intern selbst ungetypt. Allerdings bietet es auf einem semantischen Level verschiedene algebraische Datentypen (Polynome, Matrizen, Operatoren, Zahlbereiche usw.) einschließlich umfangreicher Bibliotheksfunktionen zu deren Bearbeitung an. Die Kenntnis dieser Datenstrukturen vorausgesetzt stellt RLISP also ein auf einer imperativen Programmiersprache aufbauendes System von Werkzeugen zur Programmierung (und interaktiven Abarbeitung) symbolisch-algebraischer Algorithmen dar. Da man diese Kenntnisse nur von sehr fortgeschrittenen Nutzern erwarten kann, sind die meisten dieser Werkzeuge auch in eine interaktive (und in Grenzen programmierbare) algebraische Oberfläche eingebunden, den *algebraischen Modus*. Dieser erlaubt eine stark an die übliche mathematische Notation angelehnte Eingabe, die über ein Schnittstellensystem dem *symbolischen Modus* zur Bearbeitung übergeben wird. Das Ergebnis wird dann von derselben Schnittstelle in eine an der mathematischen Notation orientierte Form zurücktransformiert und ausgegeben.

REDUCE besteht aus einer Vielzahl in compilierter Form vorliegender FASL (fast load) Moduln. Von diesen wird ein Teil (die Zahlberiechsarithmetik, die Polynomarithmetik usw.) bei der Installation in einem Imagefile *reduce.img* abgelegt, das bei jedem Start von REDUCE

geladen wird und somit von Anfang an zur Verfügung steht. Weitere Moduln werden zur Laufzeit entweder automatisch oder auf explizite Nutzeranforderung hin dazugeladen.

Kapitel 3

Der algebraische Modus

3.1 Der Interperterzyklus

Nach dem Start von REDUCE meldet sich das System mit einem Banner der Art

```
REDUCE 3.5, 15-Oct-93 ...
```

und geht danach in den *Interpreterzyklus* über. Dieser besteht aus den Teilen

- Prompt mit der Nummer der Eingabeaufforderung
- Eingabe durch den Nutzer (diese kann über mehrere Zeilen gehen)
- Eingabeabschluß durch einen *Terminator*

  ; für Auswertung mit Ergebnisausgabe
  $ für Auswertung ohne Ergebnisausgabe

  Auswertung und evtl. Ausgabe des Ergebnisses durch das System

Mit dem Befehl `quit;` wird die REDUCE Sitzung schließlich verlassen.

In der *Auswertungsphase* werden dabei rekursiv

- der eingegebene Ausdruck analysiert und in eine kanonische Form überführt,
- in den Ausdruck eingehende Variablen durch ihnen zugewiesene Werte ersetzt und
- systeminterne sowie nutzerdefinierte Simplifikationsregeln angewandt.

Auf früher eingegebene Ausdrücke und Resultate kann man im folgenden zugreifen, indem man die Ergebnisse Variablen zuweist, z.B.

```
u := (x+y+z)^2;
```

oder sie mit folgenden Hilfsmitteln lokalisiert :

saveas < *name* >	speichert *ws* unter dem angegebenen Variablennamen
input < n >	Eingabe < n > erneut auswerten
ws < n >	Ausgabe < n > einsetzen
ws	die letzte Ausgabe einsetzen
display < n >	die letzten < n > Eingaben anzeigen
display all	alle Eingaben anzeigen

Beispiele :

```
1: factorize(x^2-1);

        {X - 1,X + 1}

2: df(sin(exp(x)),x);

          X      X
        E *COS(E )

3: int(1/(x^3-1),x);

                      2*X + 1           2
  - 2*SQRT(3)*ATAN(---------) - LOG(X  + X + 1) + 2*LOG(X - 1)
                      SQRT(3)
-------------------------------------------------------------
                               6

4: solve(x^4+4,x);

        {X= - (I + 1),

         X=I - 1,

         X= - I + 1,

         X=I + 1}
```

REDUCE bietet auf verschiedenen Plattformen (insbesondere unter Windows) weiterhin einen zeilenweisen Eingabepuffer, der mit den Kursortasten durchblättert werden kann. Hat man mit `off nat;` auf eine lineare Ausgabe geschaltet, kann man schließlich auch mit den üblichen Fensterkopiertechniken Ausgaben für neue Eingaben abgreifen.

Ausdrücke, denen man Werte zuweisen kann, bezeichnet man dabei als *Kerne*. Neben Bezeichnern sind dies im wesentlichen Operatorausdrücke und Feldelemente. Jeder solche Kern ist entweder vorinitialisiert (Feldelemente) oder steht am Anfang für sich selbst, d.h. wertet zu sich selbst aus und benötigt keine Grundinitialisierung, wie man das z.B. von PASCAL her gewohnt ist. Im weiteren Verlauf einer Sitzung hat man aber sehr wohl zwischen dem Bezeichner und seinem Wert zu unterscheiden, da letzterer sich durch Zuweisungen ändern kann.

Die Eingaben für den Interpreterzyklus können sowohl interaktiv erfolgen (*user mode*) als auch von einer Datei (*batch mode*). Die Verwendung von Fileumlenkungen auf Betriebssystemebene muß REDUCE jedoch angezeigt werden (durch `off usermode;`), um auch hier den Batchmodus wirksam werden zu lassen. Die beiden Modi unterscheiden sich vor allem in der Art der Fehlerbehandlung.

3.2 Die Syntax

3.2.1 Bezeichner und Zahleingaben

REDUCE Symbole werden aus

1. den 26 Buchstaben a bis z
2. den 10 Ziffern `0` bis `9` und
3. den Spezialzeichen `_ ! " $ % ' ( ) + , - . * / ^ : ; < > = { } <blank>`

zusammengesetzt. Mit Ausnahme von Strings und Zeichen, die auf ein ! folgen, wird Groß- oder Kleinschreibung dabei ignoriert und (abhängig von der Implementierung) intern auf Groß- oder Kleinbuchstaben abgebildet. *Alpha*, *ALPHA* oder *alpha* referenzieren also denselben Bezeichner.

Die *Spezialzeichen* haben folgende Bedeutung :

!	Entwertung des darauffolgenden Zeichens als Sonderzeichen
"..."	String
$;	Terminator
%	Beginn eines Zeilenkommentars
'	Lisp Quote-Zeichen
.	Lisp Cons-Operator oder Dezimalpunkt
(...)	Klammern
+ - * / ^	Algebraische Operationszeichen
,	Separator innerhalb von Listen
{ }	Listenkonstruktor
:=	Zuweisungsoperator
< > <= >= =	Vergleichsoperatoren

Tabelle 2 : Bedeutung der Spezialzeichen in REDUCE

Bezeichner bestehen in REDUCE aus einem oder mehreren alphanumerischen Zeichen, die mit einem Buchstaben beginnen müssen. Die maximale Länge signifikanter Symbole ist implementationsabhängig, aber normalerweise größer als 24. _ wird *innerhalb* eines Bezeichners ebenfalls akzeptiert. Sonderzeichen wie $*, -, =$ und selbst das Leerzeichen können ebenfalls in Bezeichner aufgenommen werden, wenn sie auf das Entwertungszeichen ! folgen. Z.B. sind

```
a               az              p1
q23p            a_very_long_variable
light!-years    d!*!*n          good! morning
!$sign          !5goldrings
```

alles gültige Bezeichner. Bezeichner mit Sonderzeichen sollte man allerdings nicht verwenden, da viele interne Systemvariable diese Gestalt haben.

Bezeichner werden für Variablennamen, Marken, Arraynamen, Operatoren und Prozeduren verwendet, wovon einige bereits durch das System belegt sind. Dazu gehören insbesondere E, die Basis des natürlichen Logarithmus, und $I = \sqrt{-1}$.

REDUCE bietet verschiedene *Zahltypen* an. Ganze Zahlen sind in der Form

```
-2, 5396, +32
```

als vorzeichenbehaftete oder vorzeichenlose Folge von Ziffern darzustellen. Da mit exakten Zahlen gerechnet wird, können diese Folgen im Prinzip beliebige Länge erreichen. Sie dürfen allerdings keine Blanks enthalten. Zum gegenwärtigen Zeitpunkt wird eine Ausgabe über mehrere Zeilen nur durch die Terminalausgabe gebrochen. Eine sich über mehrere Zeilen erstreckende Eingabe *einer* Zahl ist nicht möglich, da REDUCE (noch) kein Zeilenfortsetzungszeichen kennt.

Andere Zahlen wie rationale Zahlen, komplexe Zahlen oder algebraische Zahlen werden aus ganzen Zahlen und entsprechenden algebraischen Operationen zusammengesetzt, wie z.B.

```
-2/3, 5396+32*I, SQRT(3)
```

Man kann in REDUCE auch von der exakten auf eine approximative Zahldarstellung umschalten (*on rounded;*), die eine näherungsweise numerische Auswertung von Ausdrücken gestattet. Solche Zahlen kann man auf zwei Arten angeben :

1. als eine vorzeichenbehaftete oder vorzeichenlose Folge von Ziffern, die einen Dezimalpunkt enthält oder auf einen solchen endet

2. als eine ganze Zahl, gefolgt von einem dezimalen Exponent, dargestellt durch ein *unmittelbar folgendes* `E`, gefolgt von einer ganzen Zahl mit oder ohne Vorzeichen.

`32. +32.0 0.32E2` und `320.E-1` sind alles Darstellungen reeller Zahlen.

Warnung: Eine solche Zahl darf *nicht* mit einem Dezimalpunkt beginnen, da dies mit dem Cons-Operator verwechselt wird. Nicht erlaubt sind also Darstellunegn der Form : `.5 -.23 +.12`; man verwende `0.5 -0.23 +0.12` stattdessen.

3.2.2 Reservierte Bezeichner

Variable	algebraischer Modus	symbolischer Modus
e	Basis des natürlichen Logarithmus. Hat unter *on rounded* einen numerischen Wert.	
I	I^2 wird durch -1 ersetzt. Gilt nicht für Zähl- oder Prozedurvariable.	
infinity	∞	
nil	Synonym für 0	Die leere Liste, Wahrheitswert $false$.
pi	π. Hat unter *on rounded* einen numerischen Wert.	
t		Wahrheitswert *true*

Tabelle 3 : Wichtige reservierte Variablen

3.2.3 Kommentare

Es gibt zwei Möglichkeiten, Kommentare in REDUCE Quelltexte einzufügen :

Alle Zeichen nach einem % auf derselben Zeile werden ignoriert.

Ein mit dem Schlüsselwort *comment* beginnender Befehl wird nicht analysiert, d.h. der gesamte Text nach diesem Schlüsselwort bis zum nächsten Terminator wird ignoriert.

3.2.4 Präfix- und Infixoperatoren

Operatoren sind in REDUCE intern generell als *Präfixoperatoren* dargestellt. Allerdings besteht die Möglichkeit, Operatoren als *Infixoperatoren* zu deklarieren. Die entsprechende Notation wird dann vom Parser in die Präfixnotation umgewandelt. Infixoperatoren haben dabei eine definierte *Bindungskraft* bzgl. anderer Operatoren, die der mathematischen Notation weitgehend entspricht.

Beispiele :

`a + b * c`	$= a + (b * c)$
`a * b ** c`	$= a * (b^c)$
`sqrt(2) + b * c - d`	$= \sqrt{2} + (b * c) - d$
`a + b * cos(c-d) - r`	$= a + (b * \cos(c - d)) - r$

Bei unären Präfixoperatoren kann man die Klammern um das Argument weglassen, sofern das nicht zu syntaktischen Mißverständnissen führt. Sie haben eine höhere Bindungskraft als alle Infixoperatoren.

Beispiele :

`a + b - c`	Leerzeichen optional
`x<y and y=z`	Leerzeichen notwendig
`cos y`	$= \cos(y)$
`cos (-y)`	Klammern notwendig, sonst wird - nicht als unärer Operator aufgefaßt
`log cos y`	$= \log(\cos(y))$
`log cos (a+b)`	$= \log(\cos(a + b))$
`cos a*b`	$= (\cos(a)) * b$

Präfixoperatoren gleicher Bindungskraft werden in der Regel linksassoziativ zusammengefaßt.

Beispiele :

`a/b/c`	$= (a/b)/c$
`a / b*c`	$= (a/b) * c$
`a^b^c`	$= (a^b)^c$

Dies gilt nicht für den Zuweisungs- und den Cons-Operator : `a:=b:=c` wird als `a:=(b:=c)` interpretiert und weist den Variablen a und b den Wert von c zu.

Operatoren können

- rein abstrakt als Symbole ohne weitere Eigenschaften (z.B. $f(x), g(x,y,z)$),
- als Symbole mit (teilweise) zugewiesenen Werten (z.B. $f(1) := 2$ oder über *LET*-Regeln),
- als Symbole mit Simplifikationsregeln (unter Verwendung allgemeinerer Substitutionsverfahren)
- oder als vollständig über Prozedurdeklarationen definierte Funktionsaufrufe

verwendet werden.

Die folgenden Infixoperatoren, aufgelistet nach wachsender *Bindungskraft*, sind im System verfügbar :

```
<infix operator>::= where|:=|=>|or|and|not|member|memq|=|neq|eq|
                    >=|>|<=|<|+|-|*|/|^|**|.
```

Man kann diese Operatoren in folgende Teilklassen untergliedern :

```
<assignment operator>   ::= :=
<logical operator>      ::= or|and|not|member|memq
<relational operator>   ::= =|neq|eq|>=|>|<=|<
<substitution operator> ::= where|=>
<arithmetic operator>   ::= +|-|*|/|^|**
<construction operator> ::= .
```

Die nicht alphanumerischen Operatorzeichen werden intern durch einen entsprechenden alphanumerischen Bezeichner ersetzt, mit dem sie auch direkt angesprochen werden können :

`:=`	`setq` (der Zuweisungsoperator)
`=>`	`replaceby`
`=`	`equal`
`>=`	`geq`
`>`	`greaterp`
`<=`	`leq`
`<`	`lessp`
`+`	`plus`
`-`	`difference`
`-`	`minus` (der unäre Operator)
`*`	`times`
`/`	`quotient`
`/`	`recip` (der unäre Operator)
`^oder **`	`expt` (Potenzieren)
`.`	`cons`

Für `NEQ` (not equal) gibt es kein spezielles Zeichen. Die Operatoren `<,<=,>,>=` können nur für den Vergleich von Zahlen eingesetzt werden.

3.2.5 Ausdrücke

REDUCE unterscheidet

- skalare Ausdrücke (dies ist die Regelform eines alleinstehenden Ausdrucks, der ein algebraisches Ergebnis liefert),
- ganzzahlige Ausdrücke (die bei Auswertung eine ganze Zahl zurückgeben),
- boolesche Ausdrücke (die einen Wahrheitswert zurückgeben und so in entsprechenden Verzweigungs- und Schleifenkonstrukten auftreten) und
- Gleichungen.

Skalare Ausdrücke sind Ausdrücke, die unter Verwendung der 5 arithmetischen Operationen `+ - * / ^` und Klammern aus Zahlen, "skalaren" Variablen und Operator- oder Prozeduraufrufen mit skalaren Ausdrücken als Argumente zusammengesetzt sind.

Beispiele :

```
x
x^3 - 2*y/(2*z^2 - df(x,z))
(p^2 + m^2)^(1/2)*log (y/m)
a(5) + b(i,q)
```

Zuweisungen, gewöhnlich in Klammern, können ebenfalls Teil eines skalaren Ausdrucks sein :

```
w + (c:=x+y) + z
```

Ausdrücke werden von innen nach außen ausgewertet, indem rekursiv die entsprechenden Teilausdrücke durch ihren Wert ersetzt und die internen und vereinbarten Simplifikationsregeln angewandt werden, insbesondere Expansion von Ausdrücken, Sammeln gleichartiger Terme, Auswertung von Ableitungen und anderer Operatoren.

Ergänzend zu den booleschen Infixoperatoren, die bereits oben aufgelistet wurden, gibt es eine Reihe von booleschen *Typoperatoren* :

`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 ganzen oder reellen Zahl 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.

Boolesche Ausdrücke können nur in `IF`, `FOR`, `WHILE` und `UNTIL` Konstrukten verwendet werden. Eine direkte algebraische Auswertung oder Zuweisung zu einer Variablen ist nicht möglich. Boolesche Ausdrücke mit `AND` und `OR` werden kurz ausgewertet, d.h. nur soweit, bis der Wert feststeht. So versucht z.B.

```
numberp x and numberp y and x>y
```

nur, $x > y$ für *Zahlen*werte x und y auszuwerten. Algebraische Prozeduren können als boolesche Prozeduren verwendet werden, wobei der Wert 0 als $false$ und alle davon verschiedenen Werte als *true* interpretiert werden. Das erlaubt es, auch im algebraischen Modus eigene boolesche Prozeduren zu definieren.

Gleichungen sind spezielle Ausdrücke der Form

```
<expression> = <expression>.
```

Gleichungen, in denen beide Ausdrücke zu Zahlen auswerten, haben eine booleschen Wert, ansonsten stellen Gleichungen einen eigenen Typ dar, auf den man z.B. den `SOLVE`-Operator anwenden kann. Bei der Auswertung von Gleichungen wird normalerweise nur die rechte Seite ausgewertet. Mit `on evallhseqp;` kann man auch die Auswertung der linken Seite erzwingen. Die Selektoren `LHS` und `RHS` geben die linke bzw. rechte Seite einer Gleichung zurück.

3.2.6 Schalter

Der Ablauf des REDUCE Interpreterzyklus kann durch das Setzen einer Vielzahl von Schaltern beeinflußt werden. Diese unterteilen sich in

- Traceschalter (3.2.10),
- Schalter zur Ausgabeformatierung (3.7),
- Schalter zur Zahlbereichsauswahl (3.4.2) und
- Schalter zur Ablaufsteuerung (3.6.4).

Schalter sind reservierte Schlüsselworte, die ihre Bedeutung mit `on <switch>;` und `off <switch>;` ändern.

3.2.7 Listen

Listen sind in REDUCE das Hauptobjekt zur Aggregation verschiedener Daten. Listen bestehen aus einer endlichen Aufzählung anderer Objekte (welche selbst wieder Listen sein können), die in geschweifte Klammern eingeschlossen und durch Kommata getrennt sind. {} ist die leere Liste.

Beispiele :

```
{a,b,c}

{1,a-b,c=d}

{{a},{{b,c},d},e}.
```

Folgende *Listenoperationen* stehen zur Verfügung :

`PART(<list>,n)`	der n-te Listeneintrag
`PART(<list>,n1,n2,...)`	der Eintrag $(n1, n2, \ldots)$ in einer geschachtelten Listen
`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

3.2.8 Deklarationen und Zuweisungen

Nichtinitialisierte Bezeichner stehen grundsätzlich für sich selbst. Ihnen kann jedoch über verschiedene Mechanismen eine neue Bedeutung zugewiesen werden.

Die einfachste Umbewertung ist die *Wertzuweisung* :

```
<Wertzuweisung> ::= <expression> := <expression>
```

Beispiele :

```
a1 := b + c
h(l,m) := x-2*y
k(3,5) := x-2*y,
```

wobei h und k Operatoren sind.

Bei einer Wertzuweisung wird nur die rechte Seite ausgewertet. Es ist möglich, mehrere Zuweisungen zusammenzufassen :

```
<expression> := ... := <expression> := <expression>
```

weist allen Ausdrücken außer dem letzten den Wert dieses letzten Ausdrucks zu.

In manchen Situationen muß auch die linke Seite einer Zuweisung ausgewertet werden. Dies erlaubt der `SET`-Operator :

```
SET(<expression>,<expression>);
```

So weist z.B. die Anweisungskette

```
j := 23;
set(mkid(a,j),x);
```

der Variablen `A23` den Wert `X` zu.

Eine weitere Umbewertung kann geschehen, indem man Bezeichner für verschiedene REDUCE Objekte reserviert. Dies sind insbesondere *Arrays* und *Operatoren* sowie spezielle Instanzen derselben wie z.B. Matrizen. Beide stellen die Möglichkeit zur Verfügung, auf indizierte Variablen zurückzugreifen. Solche Deklarationen sind stets, auch wenn sie innerhalb von Blöcken erfolgen, *globaler* Natur.

Arrays haben dabei eine definierte Parameterzahl und durch nichtnegative ganzzahlige Werte beschränkte Indexbereiche. Bei ihrer Definition werden die Arrayelemente 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);
```

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.

Weiterhin können eigene Operatoren eingeführt werden. Dies wird im nächsten Abschnitt beschrieben.

Vereinbarte Deklarationen und Zuweisungen können mit dem Kommando

```
clear <var1>,<var2>,...;
```

gelöscht werden.

Schließlich gibt es noch die Möglichkeit, Makros zu definieren, die durch den Parser ersetzt werden, bevor der Ausdruck evaluiert wird. Diese werden mit

```
DEFINE <macro1>,<macro2>,...;
```

vereinbart, wobei jedes Makro ein Ausdruck der Form

```
<identifier> = <number>|<identifier>|<operator>|
                  <reserved word>|<expression>
```

ist.

Beispiel :

```
define pi_halbe=pi/2, drittel=1/3;
```

3.2.9 Operatoren

Eigene *Operatoren* können mit dem Aufruf `operator` $<$ *name* $>$`;` eingeführt werden. Sie sind nach ihrer Definition (selbst wenn dies lokal in einem Block geschah) globale Objekte und in allen Prozeduren zugänglich. Operatoren sind Funktionsnamen vergleichbar, denen jedoch initial keine konkrete Funktion zugewiesen ist. Derselbe (neudefinierte) Operator kann mit unterschiedlichen Typen von Argumenten und sogar unterschiedlich langen Parameterlisten aufgerufen werden und stellt somit eine sehr allgemeine Art indizierter Variablen zur Verfügung. Die entsprechenden Funktionswerte stehen initial für sich selbst, können aber mit anderen Werten belegt werden.

Beispiel :

```
operator h,g1,arctan;
```

fügt die Präfixoperatoren `H`, `G1` und `ARCTAN` zum System hinzu. Das erlaubt es, Symbole wie `h(w), h(x,y,z), g1(p+q), arctan(u/v)` zu verwenden.

`infix` und `precedence` können einen solchen Operator als *Infixoperator* deklarieren und seine *Bindungskraft* einstellen.

Beispiel :

```
operator mm;
infix mm;
precedence mm,-;
a - b mm c - d;
a * b mm c * d;
```

deklariert einen neuen Operator *mm* als Infixoperator und setzt seine Bindungskraft unmittelbar *nach* der von $-$. Der Ausdruck in der vorletzten Zeile wertet also zu $a - mm(b, c) - d$ aus, während der Ausdruck in der letzten Zeile zu $mm(ab, cd)$ auswertet.

Operatoren können Eigenschaften zugeschrieben werden, die ihr Simplifikationsverhalten beeinflußen. Standardmäßig wird angenommen, daß Operatoren für rellwertige, ansonsten aber beliebige Funktionen stehen.

`even op;`	deklariert den Operator *op* als gerade Funktion bzgl. des ersten Parameters
`odd op;`	deklariert den Operator *op* als ungerade Funktion bzgl. des ersten Parameters
`linear op;`	deklariert den Operator *op* als linearen Operator im ersten Parameter über Potenzen des zweiten Parameters. Der zweite Parameter muß dazu ein Kern sein.
`symmetric op;`	deklariert den Operator *op* als symmetrischen Operator, d.h. sein Wert ist unabhängig von der Reihenfolge der Parameter
`antisymmetric op;`	deklariert den Operator *op* als antisymmetrischen Operator, d.h. sein Wert ändert das Vorzeichen bei einer Transposition zweier Parameter
`noncom op;`	deklariert den Operator *op* als Funktion mit Werten in einem nichtkommutativen Bereich (einer nichtkommutativen Algebra über dem aktuellen Zahlbereich)

Tabelle 4 : Definierbare Operatoreigenschaften

Beispiele :

```
linear f;
f(a*x^5+b*x+c,x);
```

```
           5
        F(X ,X)*A + F(X,X)*B + F(1,X)*C

noncom u,v;
u(x)*u(y)-u(y)*u(x);

        U(X)*U(Y) - U(Y)*U(X)

u(x)*v(y)-v(y)*u(x);

        U(X)*V(Y) - V(Y)*U(X)
```

Man beachte, daß die Ausdrücke `u(x)*u(y)-u(y)*u(x)` und `u(x)*v(y)-v(y)*u(x)` bei Simplifikation unverändert bleiben und nicht zu 0 vereinfachen. Mit Hilfe geeigneter Regelsysteme kann man für solche nichtkommutativen Operatoren spezielle Kommutatorrelationen definieren. Im folgenden zeigen wir eine Invarianzeigenschaft in einer antikommutativen Struktur :

```
operator s; off nat;
noncom s;       % Vereinbart nichtkommutativen Operator
rules:={s(~k)*s(~l) => - s(l)*s(k) when k>l}$
                % Antikommutatorregeln
inv:=for i:=1:3 sum s(i)^2;
                % Summe der Quadrate

        INV := S(3)**2 + S(2)**2 + S(1)**2$

for i:=1:3 collect (inv*s(i)-s(i)*inv);
                % Kommutator ohne Regeln

{S(3)**2*S(1) + S(2)**2*S(1) - S(1)*S(3)**2 - S(1)*S(2)**2,
S(3)**2*S(2) - S(2)*S(3)**2 - S(2)*S(1)**2 + S(1)**2*S(2),
 - S(3)*S(2)**2 - S(3)*S(1)**2 + S(2)**2*S(3) + S(1)**2*S(3)}$

ws where rules; % Kommutator mit Regeln

        {0,0,0}$
```

Kerne können untereinander in Abhängigkeit stehen. Z.B. kann f eine Funktion sein, die von x und y abhängt. Dies kann man dem System durch

```
depend f,x,y;
```

bekanntgeben.

Beispiel :

```
df(f,x);
```

```
        0

depend f,x,y;
df(f,x);

        DF(F,X)
```

Die Bindung kann durch

```
nodepend f,x,y;
```

wieder (auch teilweise) aufgehoben werden.

3.2.10 Fehlermeldungen und Tracing

In REDUCE gibt es verschiedene Arten von *Fehlermeldungen* : Warnungen, Unklarheiten und fatale Fehler. Warnungen werden durch Ausschriften mit 3 Sternen eingeleitet und führen nicht zur Programmunterbrechung. Mit `off msg` können sie unterdrückt werden. Bei Unklarheiten wird im *user mode* eine Rückfrage an den Nutzer gestellt, im *batch mode* mit einer plausiblen Annahme fortgesetzt. Fehler, die zu einem Abbruch der Prozedur führen, werden durch Ausschriften mit 5 Sternen angegeben. Danach startet der Interpreter den nächsten Zyklus.

Tracing unterteilt sich in drei Arten :

1. Zwischenausschriften bei rechenintensiven algebraischen Prozeduren wie Gleichungslösen, Faktorisieren usw.
2. Anzeige systemspezifischer Informationen wie Laufzeiten, Eingabeecho usw. und
3. Aufrufverfolgung zur Fehlersuche.

Ersteres ist standardmäßig abgeschaltet und kann für verschiedene Prozeduren durch Setzen entsprechender Schalter ($on < tracing >;$) zugeschaltet werden.

Funktion	Traceschalter	Effekt
Integration	trint	Tracefunktion des Integrators aus dem Kern
	tra	zusätzliche Tracefunktion des Pakets *algint*
Faktorisierung	trfac	Verfolgung der Reduktion des Problems
	trallfac	Detailliertere Tracefunktion
	timings	Angabe von Zeiten für die verschiedenen Schritte im Faktorisierungalgorithmus

Tabelle 5 : Traceschalter für verschiedene Operatoren

`on time`	Ausgabe der CPU-Zeit jedes Interpreterzyklus
`showtime;`	Ausgabe der verbrauchten CPU-Zeit seit dem letzten Aufruf dieses Kommandos (oder seit Start der Sitzung)
`on gc`	Ausgabe von Informationen über garbage collection

Tabelle 6 : Ausgabe systemspezifischer Informationen

Die Verfolgung von Prozeduraufrufen kann erreicht werden, indem man die zu verfolgenden Prozeduren mit

```
tr <proc1>,<proc2>,<proc3>;
```

markiert. Bei jedem Aufruf der genannten Prozeduren werden nun die Aufrufparameter (allerdings im symbolischen Modus) und der Returnwert angegeben.

```
untr <proc1>,<proc2>,<proc3>;
```

beendet dieses Tracing wieder. Eine intensivere Aufrufverfolgung (auch aller Schleifendurchläufe) von *nichtcompilierten* Prozeduren ist mit `trst proc1,proc2,...` möglich. Den Aufrufstack kann man mit `on backtrace;` verfolgen. Allerdings ist für das Verständnis der dabei ausgegebenen Information die gute Kenntnis der internen Struktur von REDUCE notwendig.

3.3 Die Steuerstrukturen

REDUCE stellt die in anderen imperativen Programmiersprachen üblichen Steuerstrukturen ebenfalls zur Verfügung. Ihre Syntax ist im folgenden zusammengestellt.

3.3.1 Verbundanweisungen

Verbundanweisungen stellen Konstrukte dar, die mehrere Anweisungen zu einer zusammenfassen. REDUCE bietet hierfür zwei unterschiedliche Formen an, *Gruppenanweisungen* und *Blöcke*.

Gruppenanweisung

Gruppenanweisungen werden an solchen Stellen verwendet, wo REDUCE eine einzelne Anweisung erwartet, jedoch eine Serie von Anweisungen auszuführen ist. Eine solche Gruppenanweisung besteht aus mehreren durch $ oder ; getrennten Anweisungen, die in << ... >> eingeschlossen sind. Der (ausgegebene) Rückgabewert der Gruppenanweisung ist der Wert der letzten Anweisung der Gruppe. Man beachte, daß eine Gruppenanweisung, deren Wert von Bedeutung ist, innerhalb der Klammern *nicht* durch einen Terminator abgeschlossen wird, da die Terminatoren *Trenn*zeichen sind.

Beispiel : Vertauschen zweier Zahlen a und b

```
if a>b then << c:=a; a:=b; b:=c >>;
```

Blöcke

Ein umfassenderes Konstrukt sind *Blöcke*. Diese fassen ganze Teilprogramme zu einer einzigen Anweisung zusammen. Obwohl sie typischerweise für die Unterprogrammdefinition verwendet werden, können solche Blöcke auch an anderen Stellen eingesetzt werden. Auch eine Schachtelung von Blöcken ist möglich. Die allgemeine Syntax ist die folgende :

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

Lokale Variablen können verschiedenen Typs sein :

Typ	Bedeutung	Initialisierung
`scalar`	Regeltyp	0 oder NIL
`integer`	Wert muß eine ganze Zahl sein	0
`real`	Wert muß eine relle Zahl sein (wird derzeit auf `scalar` abgebildet)	

Tabelle 7 : Variablentypen im algebraischen Modus

Beispiele :

```
scalar a,b,c;
scalar z;
integer k,l;
```

Die Variablendeklaration muß am Anfang des Blocks *vor* allen anderen Anweisungen (auch Operator- oder Arraydeklarationen) erfolgen. Nichtdeklarierte Variablen referenzieren die Variablen gleichen Namens aus der Umgebung des Blocks.

Die folgenden Anweisungen im Körper des Blocks können auch Operator- und Arraydeklarationen enthalten. Diese Deklarationen sind jedoch globaler Natur, ihre Gültigkeit also nicht auf den Block beschränkt.

Eine Blockanweisung gibt wie jede andere Anweisung einen Wert zurück. Dieser ist 0 oder NIL, wenn nicht durch einen RETURN-Befehl etwas anderes vereinbart ist.

```
RETURN <wert>
```

kann an jeder Stelle des Blocks stehen und beendet die Blockabarbeitung mit der Rückgabe des entsprechenden Werts. `RETURN` (ohne Parameter) beendet die Abarbeitung und gibt den Standardwert zurück. In Verbindung mit Konditionalanweisungen kann es auch sinnvoll sein, in einem Block mehrere RETURN-Anweisungen zu haben. Jedoch müssen diese stets in der *obersten* Ebene des Blocks stehen. Ein Rücksprung aus Gruppenanweisungen, Schleifen etc., die *innerhalb* des Blocks verwendet werden, ebenso wie der Rücksprung über mehrere Blockebenen, ist nicht möglich.

Beispiel :

```
return x+y;
return m;
return;
```

Hat n einen positiven ganzzahligen Wert, so berechnet z.B. der folgende Block die Bernoullizahl B_n, die sich aus der Taylorreihenentwicklung

$$1 - \frac{x}{2} - \frac{x}{e^x - 1} = \sum_{k=0}^{\infty} (-1)^k B_k \frac{x^{2k}}{(2k)!}$$

ergibt.

```
begin scalar f,g,sgn;
   f:=1-x/2-x/(exp(x)-1);
   g:=df(f,x,2k);
   sgn:=(-1)**k;
   return sgn*limit(g,x,0)
end;
```

Eine Gruppenanweisung kann als Block ohne lokale Variablen und RETURN-Befehl geschrieben werden.

3.3.2 Verzweigungen

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

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

Beispiele:

```
if x=5 then a:=b+c else d:=e+f

if x=5 and numberp y
   then <<ff:=q1; a:=b+c>>
   else <<ff:=q2; d:=e+f>>
```

Verzweigungen sind rechtsassoziativ, d.h.

```
IF <a> THEN <b> ELSE IF <c> THEN <d> ELSE <e>
```

ist äquivalent zu:

```
IF <a> THEN <b> ELSE (IF <c> THEN <d> ELSE <e>)
```

Das Konstrukt

```
IF <a> THEN IF <b> THEN <c> ELSE <d>
```

wird aufgelöst zu

```
IF <a> THEN (IF <b> THEN <c> ELSE <d>).
```

Entsprechend dem Datenverständnis von LISP sind Verzweigungen selbst wieder Ausdrücke, die einen Wert zurückliefern. Man kann sie also in andere Ausdrücke nach den oben beschriebenen Regeln einbauen. Der Wert ist gleich dem des ausgeführten Zweiges (oder 0 bzw. NIL, wenn kein Zweig ausgeführt wurde).

Beispiele :

```
a:=if x<5 then 123 else 456;
b:=u + v^(if numberp z then 10*z  else 1) + w;
```

3.3.3 FOR-Schleifen

REDUCE bietet eine Reihe verschiedener FOR-Schleifenkonstrukte an, die sich insbesondere an dem algebraischen Charakter als auch der starken Verwendung von Listen orientieren.

$$\texttt{FOR}\left\{\begin{array}{l}\langle var\rangle\ \texttt{:=}\ \langle number\rangle\ \left\{\begin{array}{c}\texttt{STEP}\ \langle number\rangle\ \texttt{UNTIL}\\ \texttt{:}\end{array}\right\}\langle number\rangle \\ \texttt{EACH}\ \langle var\rangle\ \texttt{IN}\ \langle list\rangle\end{array}\right\}\langle action\rangle\ \langle exprn\rangle$$

wobei

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

Die obere 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 Aktionen bedeuten dabei 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 Zählberiechs liegt.

Beispiele :

1.
```
array a(10),b(10);
for i := 5 step 1 until 10 do <<a(i):=i^2; b(i):=i^3>>;
```
Diese Schleife speichert die entsprechenden Quadrate und Kuben in den vorher deklarierten Feldern A und B. i

2. Zur Vereinfachung kann das oft verwendete `STEP 1 UNTIL` durch einen Doppelpunkt (:) abgekürzt werden :

```
for i := 5 : 10 collect a(i);

     {25,36,49,64,81,100}
```

3. Zur Verwendung von *sum* und *product* :

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

        C := 165
```

```
on factor;
d := for k:=0 step 1 until 4 product (x+k);

        D := (X + 4)*(X + 3)*(X + 2)*(X + 1)*X
```

4. 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^2};

          2  2  2
        {A ,B ,C }
```

5. Die folgende *selektive Listenauswahl* sammelt die Primzahlen bis 50 in einer Liste 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}
```

3.3.4 WHILE-Schleifen

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

Als Anweisung kann wiederum eine Verbundanweisung verwwendet werden.

Beispiel : Eine iterative Lösung der Gleichung $x + sin(x) = 1$

```
on rounded;
old:=0; new:=1;

        OLD := 0

        NEW := 1

while abs(old-new) > 1E-3 do
        << old:=new; new:=1-sin(old) >>;

new;

        0.510508595505
```

3.3.5 REPEAT-Schleifen

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

Als Anweisung kann wiederum eine Verbundanweisung verwendet werden.

Beispiel : Dieselbe Aufgabe wie oben, jedoch mit einer REPEAT-Schleife bearbeitet

```
on rounded;
new:=0;

        NEW := 0

repeat << old:=new; new:=1-sin(old) >>
until abs(old-new) < 1E-5 ;

new;

        0.510968968302
```

3.3.6 GO TO

```
<go to statement> ::= GO TO <label> | GOTO <label>
<label> ::= <identifier>
<labeled statement> ::= <label>:<statement>
```

GO TO Sprünge sind nur innerhalb eines Blocks und nur auf dessen höchster Ebene (d.h. nicht aus oder in Schleifen oder Verbundanweisungen) möglich.

Beispiel : Berechnung von n !

```
begin scalar m;
    m:=1;
 l: if n=0 then return m;
    m:=m*n;
    n:=n-1;
    go to l
end;
```

3.3.7 Prozeduren

REDUCE erlaubt die Definition von (nicht ineinandergeschachtelten) Prozeduren als einen speziellen Typ von Operator, der immer ausgewertet wird :

```
PROCEDURE <name>(<var_1>,...,<var_n>);<statement>;
```

Der Prozedurkörper besteht aus einer einzigen Anweisung, die in den meisten Fällen ein Block ist.

3.4 Datentypen und Operatoren

Obwohl LISP und damit auch der symbolische Mode von REDUCE eine ungetypte Sprache ist, stellt der algebraische Mode wenigstens auf semantischer Ebene Daten verschiedener Typen zur Verfügung. Diese sind teilweise mit expliziten Typdeklarationen verbunden. Im folgenden sind die wichtigsten Datentypen in REDUCE zusammen mit ihren Operatoren zusammengestellt.

3.4.1 Listen

Siehe 3.2.7

3.4.2 Zahlbereiche

Zahlbereiche sind insbesondere im Zusammenhang mit Polynomen als deren Koeffizientenbereiche interessant. REDUCE stellt dafür als Standardzahlbereich die (exakte) Integerarithmetik für ganze Zahlen beliebiger Länge zur Verfügung. In andere Zahlbereiche kann durch Schalter umgeschaltet werden.

Zahlbereich	Schalter
algebraische Zahlen	siehe arnum-Paket
exakte ganze Zahlen	Standard
exakte komplexe Zahlen	`on complex`
Floatzahlen (Maschinengenauigkeit)	`on rounded`
Floatzahlen beliebiger Genauigkeit	`on roundbf`
komplexe Floatzahlen	`on complex,rounded`
modularer Zahlbereich	`on modular; setmod <module>;`
rationale Zahlen	`on rational`

Tabelle 8 : Verschiedene Zahlbereiche in REDUCE

Weitere Schalter und Parameter :

`on rationalize`	Nenner werden rational gemacht
`on centered_mod`	symmetrische modulare Reste verwenden

Die reelle Arithmetik

`on rounded` schaltet auf die reelle Fließkommaarithmetik um. Damit sind (approximative) numerische Auswertungen vieler algebraischer Fragestellungen möglich. Insbesondere produziert der SOLVE-Operator für Polynome in einer Variablen Näherungen für deren komplexe Nullstellen, während die exakte Auswertung definierende Polynome für die entsprechenden algebraischen Zahlen liefert.

Weitere Schalter und Parameter :

`precision <p>`	Setzen der Zahl gültiger Ziffern in der Mantisse (`precison 0` liefert die gerade gesetzte Genauigkeit zurück)
`print_precision <p>`	Setzen der Ausgabelänge der Mantisse ($\leq$ *precision*)
`off numval`	Numerische Auswertung mathematischer Funktionen abschalten
`off bfspace`	Ausgabe der rellen Zahlen ohne Zwischenräume
`on adjprec`	Adjustiert *precision* entsprechend der Genauigkeit der Eingabewerte
`off roundall`	Rationale Eingabezahlen werden nicht in reelle umgewandelt

Die komplexe Arithmetik

Zuschaltung der komplexen Arithmetik bewirkt nur eine andere Interpretation des Symbols I. `off complex` betrachtet I als Variable mit der Regel `I**2=-1`, `on complex` dagegen als Teil des Koeffizientenbereichs. Entsprechend liefert etwa `factorize(x**2+1)` im ersten Fall {`x**2+1`}, im zweiten Fall dagegen {`x+i,x-i`}.

Dieser Schalter kann sinnvoll mit `rational` oder `rounded` verknüpft werden.

Die modulare Arithmetik

`setmod <p>` setzt den Modul der entsprechenden Arithmetik, `on modular` schaltet sie zu. Danach werden alle Rechnungen über dem entsprechenden modularen Koeffizientenbereich ausgeführt. Standardmäßig werden die positiven Reste $[0 \ldots p-1]$ verwendet, mit `on centered_mod` dagegen die symmetrischen Repräsentanten $[-\frac{p-1}{2} \ldots \frac{p-1}{2}]$. `setmod 0` gibt den gerade gesetzten Modul zurück.

3.4.3 Mathematische Funktionen

Für die folgenden mathematischen Funktionen existieren numerische Prozeduren, die diese zu einem Zahlenwert auswerten, wenn die Argumente Zahlenwerte sind. Sind Argumente dagegen symbolische Ausdrücke, so wird ein möglicherweise simplifizierter symbolischer Ausdruck zurückgegeben. Zur Simplifikation werden die dem System bekannten bzw. vom Nutzer eingeführten Regeln angewendet.

`abs(x)`	absoluter Betrag
`atan2(y,x)`	$= arg(x + iy) \in [-\pi, \pi]$
`cbrt(x)`	Kubikwurzel
`ceiling(x)`	kleinste ganze Zahl oberhalb von x
`conj(x)`	die konjugiert komplexe Zahl
`exp(x)`	die Exponentialfunktion (mit der Basis e)
`factorial(x)`	Fakultät
`fix(x)`	ganzer Teil von x
`floor(x)`	größte ganze Zahl unterhalb von x
`hypot(x,y)`	$\sqrt{x^2 + y^2}$
`impart(x)`	Imaginärteil
`ln(x)=log(x)`	natürlicher Logarithmus
`logb(x,b)`	$\log_b(x)$ (=log(x)/log(b))
`log10(x)`	$\log_{10}(x)$ (=logb(x,10)
`max(a,b,...)`	Maximum
`min(a,b,...)`	Minimum
`nextprime(x)`	Nächste Primzahl (x muß ganzzahlig sein)
`random(x)`	Zufallszahl z im Bereich $0 \leq z < x$. (x muß eine positive ganze Zahl sein)
`repart(x)`	Realteil
`round(x)`	nächstegelegene ganze Zahl
`sign(x)`	Vorzeichen $(-1, 0, 1)$ von x
`sqrt(x)`	Quadratwurzel

Tabelle 9 : Mathematische Funktionen I

Trigonometrische Funktionen	sin	cos	tan	cot	sec	csc
Arcusfunktionen	asin	acos	atan	acot	asec	acsc
Hyperbolischen Funktionen	sinh	cosh	tanh	coth	sech	csch
Areafunktionen	asinh	acosh	atanh	acoth	asech	acsch

Tabelle 10 : Mathematische Funktionen II

Daneben stellt REDUCE mathematische Funktionen zur Verfügung, von denen nur einzelne Werte und Simplifikationsregeln bekannt sind. Der größte Teil dieser Funktionen ist in den Paketen *specfn* und *specfn2* enthalten und umfaßt

- Bernoullizahlen und Eulerzahlen
- Stirlingzahlen
- Binomialkoeffizienten
- Pochhammersymbole
- die Gammafunktion
- die Psi-Funktion und ihre Ableitungen
- die Riemann Zeta Funktion

- die Bessel-Funktionen J und Y erster und zweiter Art
- die modifizierten Bessel-Funktionen I und K
- die Hankel-Funktionen H1 und H2
- die Kummerschen hypergeometrischen Funktionen M und U
- die Beta-, Struve-, Lommel- und Whittaker-Funktionen
- das Exponentialintegral, die Sinus- und Cosinusintegrale
- die hyperbolischen Sinus- und Cosinusintegrale
- die Fresnel-Integrale und die Fehlerfunktion erf
- die Dilog Funktion
- Hermite Polynome
- Jacobi Polynome
- Legendre Polynome
- Laguerre Polynome
- Chebyshev Polynome
- Gegenbauer Polynome
- Euler Polynome
- Bernoulli Polynome
- Verallgemeinerte Hypergeometrische Funktionen
- Mejer's G-Funktion

3.4.4 Polynome und rationale Ausdrücke

Polynome stellen den grundlegenden Datentyp dar, der von REDUCE verarbeitet wird. Dabei unterscheidet sich die interne Darstellung im Gegensatz zu vielen anderen Operatoren wesentlich von der externen.

Polynome werden in rekursiver Form mit Hinblick auf eine interne (meist alphabetische) Reihenfolge der in ihnen vorkommenden Kerne dargestellt. Mit

```
korder v1,...,vn;
```

kann man bestimmen, daß statt dessen die Kerne `v1,...,vn` in dieser Reihenfolge vorrangig behandelt werden. Ein weiterer Aufruf von `korder` überschreibt diese Vereinbarung. Insbesondere führt `korder nil;` zur internen Ordnung zurück.

Entsprechend der rekursiven Definition kann man auf die Teile eines Polynoms nur von oben her zugreifen :

`coeff(poly,x)`	Liste der Koeffizienten des Polynoms *poly* bzgl. der Variablen x, beginnend mit dem Koeffizienten vor x^0
`coeffn(poly,x,n)`	Koeffizient des Polynoms *poly* vor x^n
`deg(poly,x)`	Grad des Polynoms *poly* bzgl. der Variablen x
`lcof(poly,x)`	Leitkoeffizient des Polynoms *poly* bzgl. der Variablen x
`lterm(poly,x)`	Leitterm des Polynoms *poly* bzgl. der Variablen x
`mainvar(poly)`	Hauptvariable des Polynoms *poly*
`reduct(poly,x)`	Reduktum des Polynoms *poly* bzgl. der Variablen x

Tabelle 11 : Selektoren für Polynome

Mit Polynomen sind außer den arithmetischen Grundoperationen eine Reihe weiterer Operationen möglich :

`decompose(poly)`	Zerlegung eines Polynoms in eine Liste *list* von ineinandergeschachtelten Ausdrücken, so daß $poly = (first\ list)\ where\ rest(list)$
`gcd(poly1,poly2)`	Bestimmung des größten gemeinsamen Teilers
`lcm(poly1,poly2)`	Bestimmung des Kleinsten gemeinsamen Vielfachen
`remainder(poly1,poly2)`	Bestimmung des Restes bei der Division mit Rest (bzgl. der internen Variablenordnung)
`resultant(poly1,poly2,var)`	Bestimmung der Resultante der beiden Polynome bzgl. der Hauptvariablen *var*

Tabelle 12 : Weitere Polynomoperationen

Rationale Ausdrücke sind Quotienten von Polynomen. Auf Zähler und Nenner kann man mit folgenden Selektoren zugreifen :

`den(r)`	Nenner von r
`num(r)`	Zähler von r

3.4.5 Matrizen

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 die folgenden Operationen definiert (m steht dabei für eine Matrix) :

`cofactor(m,i,j)`	Adjunkte (Minor mit Vorzeichen) des Elements m_{ij} in m
`det m`	Determinante von m
`length m`	Dimension von m (als Zweierliste)
`mateigen(m,x)`	Eigenwerte und Eigenvektoren von m.
`nullspace m`	Basis des Lösungsraums $m \cdot x = 0$
`rank m`	Rang der Matrix m
`tp m`	transponiert m
`trace m`	Spur von m

Tabelle 13 : Operatoren auf Matrizen

Beispiele :

```
tp m;
```

```
[A  D]
[    ]
[B  E]
[    ]
[C  F]
```

```
h^(-1);

        [ 72   -240  180 ]
        [                ]
        [-240  900   -720]
        [                ]
        [180   -720  600 ]

mateigen(h,x);

                  3          2
        {{43200*X  - 39600*X  + 1572*X - 1,

          1,


          [ 3*ARBCOMPLEX(3)*(60*X + 1) ]
          [----------------------------]
          [           2                ]
          [   10*(72*X  - 54*X + 1)    ]
          [                            ]
          [ 6*ARBCOMPLEX(3)*(12*X - 1) ]
          [----------------------------]
          [           2                ]
          [    5*(72*X  - 54*X + 1)    ]
          [                            ]
          [       ARBCOMPLEX(3)        ]

          }}
```

`mateigen(m,x)` gibt dabei als Ergebnis eine Liste aus Dreierlisten zurück, deren erste Elemente die quadratfreien Faktoren des charakteristischen Polynoms von m sind (ihr Produkt ist also gleich dem *Minimalpolynom* der Matrix m); die zweiten Elemente geben deren Vielfachheit und die dritten ein allgemeines Element aus dem zugehörigen *Eigenraum* an.

3.5 Algebraische Anwendungen

3.5.1 Differenzieren und Integrieren

`DF(EXPRN:algebraic[,VAR:kernel<,NUM:integer>]):algebraic.`

Beispiele :

`df(y,x)`	$= \partial y/\partial x$
`df(y,x,2)`	$= \partial^2 y/\partial x^2$
`df(y,x1,2,x2,x3,2)`	$= \partial^5 y/\partial x_1^2\ \partial x_2 \partial x_3^2$.

DF ist ein Operator im oben beschriebenen Sinne. Beim Auswerten werden also die intern bekannten und vom Nutzer eingegebenen Simplifikationsregeln für die Ableitungen der

verschiedenen Funktionen angewendet. Mit `depend f,x,y;` kann dabei die sonst unbekannte Funktion f als von x, y abhängig deklariert werden. Unbestimmtes Differenzieren führt stets auf einen algebraischen Ausdruck ohne DF, wenn die Ableitungen der Kerne dem System bekannt sind.

```
INT(EXPRN:algebraic,VAR:kernel):algebraic.
```

ist der REDUCE Operator für unbestimmtes Integrieren. Er verwendet eine Kombination des Risch-Norman-Algorithmus und Pattern Matching und liefert damit stets einen geschlossenen Ausdruck für Integranden, die Polynome, Logarithmen, Exponentialfunktionen, *tan* und *atan* enthalten. Mit dem Paket *algint* kann man auch kompliziertere Integrale in geschlossener Form auswerten.

REDUCE hat keine Operatoren zur Auswertung definiter Integrale, da sich die entsprechende Theorie noch in einem sehr heuristischen Stadium befindet.

3.5.2 Polynomfaktorisierung

Mit `on factor` kann man erreichen, daß die Ergebnisse in faktorisierter Form ausgegeben werden. Da Faktorisierung meist aufwendig ist, wird diese Option standardmäßig abgeschaltet.

Die Faktorzerlegung einzelner polynomialer Ausdrücke bekommt man mit

```
FACTORIZE(EXPRN:polynomial):list,
```

wobei eine Liste der Faktoren zurückgegeben wird.

Beispiele :

```
off nat;
factorize(x^105-1);

{X**6 + X**5 + X**4 + X**3 + X**2 + X + 1,
X**4 + X**3 + X**2 + X + 1,
X**24 - X**23 + X**19 - X**18 + X**17 - X**16 + X**14 - X**13 + X**
12 - X**11 + X**10 - X**8 + X**7 - X**6 + X**5 - X + 1,
X**2 + X + 1,
X**48 + X**47 + X**46 - X**43 - X**42 - 2*X**41 - X**40 - X**39 + X
**36 + X**35 + X**34 + X**33 + X**32 + X**31 - X**28 - X**26 - X**24
 - X**22 - X**20 + X**17 + X**16 + X**15 + X**14 + X**13 + X**12 - X
**9 - X**8 - 2*X**7 - X**6 - X**5 + X**2 + X + 1,
X**8 - X**7 + X**5 - X**4 + X**3 - X + 1,
X**12 - X**11 + X**9 - X**8 + X**6 - X**4 + X**3 - X + 1,
X - 1}$

factorize(12x^2-12);

{12,X - 1,X + 1}
```

Im letzten Beispiel wurde der ganzzahlige Faktor nicht mit zerlegt. Dies kann man durch `on ifactor` erreichen :

```
on ifactor; factorize(12x^2-12);

{2,2,3,X - 1,X + 1}
```

Eine Faktorzerlegung ist auch über verschiedenen anderen Zahlbereichen möglich, so z.B. (für univariate Polynome) über einem modularen Zahlbereich :

```
setmod 7; on modular;
factorize(x^4+1);

  2             2
{X  + 3*X + 1,X  + 4*X + 1}
```

3.5.3 Der SOLVE-Operator

SOLVE ist ein Operator, der eine Vielzahl von Einzelalgorithmen zum Lösen von Gleichungssystemen aus verschiedenen mathematischen Gebieten in einer gemeinsamen Oberfläche zusammenfaßt.

```
SOLVE(EXPRN:algebraic[,VAR:kernel|,VARLIST:list of kernels]):list.
```

EXPRN ist dabei ein einzelner Ausdruck oder eine Liste von Ausdrücken (im weiteren Gleichungssystem). Jeder Ausdruck ist dabei eine Gleichung oder die Differenz der beiden Seiten einer Gleichung. Das zweite Argument ist ein Kern oder eine Liste von Kernen, nach denen das Gleichungssystem aufgelöst werden soll. Fehlt es, so sucht der SOLVE-Operator selbst die Kerne des Gleichungssystems heraus und löst nach ihnen auf.

Zum Lösen einer Gleichung versucht SOLVE rekursiv zu faktorisieren, zu zerlegen und die bekannten Inversen der elementaren Funktionen einzusetzen.

Lineare Gleichungen werden standardmäßig nach dem Bareissalgorithmus gelöst. `on cramer` schaltet zur Cramerschen Regel um.

Nichtlineare Gleichungssysteme werden mit der Gröbnerbasismethode bearbeitet.

Beispiele :

```
solve(log(sin(x+3))^5 = 8,x);
solve(a*log(sin(x+3))^5 - b, sin(x+3));
solve({a*x+y=3,y=-2},{x,y});
```

SOLVE gibt eine Liste von Ergebnissen zurück. Diese enthalten entweder Gleichungen für den Wert der Unbekannten oder aber einen Ausdruck mit `root_of`, wenn der Solver das Gleichungssystem nicht vollständig auflösen konnte.

Beispiel :

```
solve(x^2=1,x);

        {X=-1,X=1}

solve(x^7-x^6+x^2=1,x);
```

```
        {X=ROOT_OF(X_  + X_ + 1,X_),X=1}

solve({x+3y=7,y-x=1},{x,y});

        {{X=1,Y=2}}
```

Ein Ausdruck mit einem `root_of` als Kopf ist implizit eine Liste mit einer unbestimmten Anzahl von Elementen. Durch Substitutionen oder Umschalten in andere Zahlbereiche kann ein solcher Ausdruck auflösbar werden. In diesem Fall wird `root_of` durch `one_of` ersetzt. `expand_cases` wandelt diese Form in die Standardausgabeform des SOLVE-Operators um.

Beispiel :

```
solve(-a*x^3+a*x^2+x^4-x^3-4*x^2+4,x);
                          2     3
        {X=ROOT_OF(A*X_  - X_  + 4*X_ + 4,X_),X=1}

sub(a=-1,ws);

        {X=ONE_OF(2,-1,-2),X=1}

expand_cases ws;

        {X=2,X=-1,X=-2,X=1}
```

Für Gleichungen 3. und 4. Grades stehen spezielle Schalter bereit, die deren Auswertung in geschlossener Form ermöglichen. Standardmäßig sind diese ausgeschaltet, weil dabei normalerweise sehr komplizierte Ausdrücke entstehen.

`on fullroots`	Angabe der (komplexen) Lösungen in ihrer trigonometrischen Form
zusätzlich `off trigform`	Angabe der Lösungen als geschachtelte Wurzelausdrücke

Tabelle 14 : Gleichungen 3. und 4. Grads in geschlossener Form lösen

Beispiel :

```
on fullroots; off nat,trigform;
solve(x^3+x+1,x);

{X=((SQRT(31) - 3*SQRT(3))**(2/3)*2**(1/3)*( - (SQRT(31) - 3*SQRT(3))
**(2/3)*2**(1/3)*SQRT(93) - 3*(SQRT(31) - 3*SQRT(3))**(2/3)*2
**(1/3)*SQRT(31)*I - 9*(SQRT(31) - 3*SQRT(3))**(2/3)*2**(1/3)*
SQRT(3)*I - 9*(SQRT(31) - 3*SQRT(3))**(2/3)*2**(1/3) + 2*SQRT(
93) - 6*SQRT(31)*I - 18*SQRT(3)*I + 18))/48,
X=((SQRT(31) - 3*SQRT(3))**(2/3)*2**(1/3)*( - (SQRT(31) - 3*SQRT(3))
**(2/3)*2**(1/3)*SQRT(93) + 3*(SQRT(31) - 3*SQRT(3))**(2/3)*2
**(1/3)*SQRT(31)*I + 9*(SQRT(31) - 3*SQRT(3))**(2/3)*2**(1/3)*
SQRT(3)*I - 9*(SQRT(31) - 3*SQRT(3))**(2/3)*2**(1/3) + 2*SQRT(
```

```
93) + 6*SQRT(31)*I + 18*SQRT(3)*I + 18))/48,
X=((SQRT(31) - 3*SQRT(3))**(2/3)*2**(1/3)*((SQRT(31) - 3*SQRT(3))**(
2/3)*2**(1/3)*SQRT(93) + 9*(SQRT(31) - 3*SQRT(3))**(2/3)*2
**(1/3) - 2*SQRT(93) - 18))/24}$

% Die Probe :
for each y in ws product x-rhs y;

X**3 + X + 1$
```

Für Gleichungssysteme mit unendlich vielen Lösungen liefert SOLVE Ausdrücke mit Parametern zurück. Zu Details konsultiere man das Manual.

3.6 Simplifikationsregeln

3.6.1 Substitutionen

Die einfachste Form der Veränderung eines symbolischen Ausdrucks besteht darin, für einige der vorkommenden Kerne andere Ausdrücke zu substituieren. Dies ist über das folgende Kommando möglich :

```
SUB(<substitution_list>,exprn)
```

wobei `<substitution_list>` eine Liste von Gleichungen

```
var_k=exprn_k
```

ist. Zur Auswertung werden in `exprn` die Variablen `var1, var2, ...` durch (die Werte von) `exprn1, exprn2, ...` ersetzt und dann der entstehende Ausdruck ausgewertet. Eine rekursive Substitution findet *nicht* statt.

Die Klammern um die Liste der Substituenten kann weggelassen werden.

Beispiele :

```
sub({x=a+y,y=y+1},x^2+y^2) ;
s := {x=a+y,y=y+1};
sub(s,x^2+y^2);
sub(x=a+y,y=y+1,x^2+y^2);
```

```
             2               2
            A  + 2*A*Y + 2*Y  + 2*Y + 1
```

3.6.2 Eigene globale Simplifikationsregeln

In REDUCE können zusätzlich zu den internen Simplifikationsregeln eigene Regeln eingeführt werden :

```
LET <substitution list>
```

wobei `<substitution list>` eine Liste von Gleichungen der Form

```
<pattern> = <expression>
```

ist. Beispiel :

```
let {x = y^2,
     h(u,v) = u - v,
     cos(pi/3) = 1/2,
     a*b = c,
     l+m = n,
     w^3 = 2*z - 3,
     z^10 = 0}
```

Die geschweiften Klammern kann man wieder weglassen. Die angegebenen Regeln hätte man auch durch sieben `LET` Kommandos eingeben können.

Man beachte den Unterschied zwischen Regeln und Zuweisungen : Die Zuweisung `x:=y**2` bewirkt eine *sofortige* Bindung des Werts von `y**2` an `x`, wonach x und y je eigene Wege gehen. Die Regel `let x=y**2` dagegen bewirkt eine *späte* Bindung. `x` wird bei jedem Auftreten durch den *aktuellen* Wert von `y**2` ersetzt.

Regelanwendungen in REDUCE stellen kein syntaktisches, sondern ein algebraisches Patternmatching dar. So bedingt die Regel `let a*b=c`, daß überall, wo `A` und `B` beide als Faktoren auftreten, ihr Produkt durch `C` ersetzt wird. So wird z.B. `a^5 * b^7*w` ersetzt durch `c^5*b^2*w`. Die Regel `l+m=n` ersetzt nicht nur `L+M` durch `N`, sondern auch `L` durch `N-M`, aber nicht `M` durch `N-L`. Die Regel `z^10 = 0` ersetzt alle Potenzen von `z` größer gleich 10 durch 0.

Muster, die mit einem arithmetischen Operator beginnen, werden also, grob gesprochen, durch Umstellen der Gleichung bzgl. einer (der in der internen Ordnung höchsten) Variablen in eine "normale" Regel verwandelt. Die Details sind komplizierter, vgl. [2, 10.2.5]. Derartige Muster sollte man deshalb sparsam verwenden.

REDUCE stellt auch die Möglichkeit zur Verfügung, parametrisierte Regeln zu definieren :

```
FOR ALL <variable>,...,<variable> <LET statement> <terminator>
```

z.B.,

```
for all x,y let h(x,y) = x-y;
for all x let k(x,y) = x^y;
```

Die Laufvariablen sind wiederum gebundene (lokale selbstdefinierende) Variablen (und verdecken evtl. vorhandene gleichnamige Variablen aus der Umgebung).

Die erste Regel führt zur Substitution

```
h(a,b) -> A-B
h(u+v,u+w) -> V-W       usw.
```

Die zweite Regel ersetzt `k(a,y)` durch `a^y`, aber hat auf `k(a,z)` keine Wirkung, weil nicht `FOR ALL Y ...` vereinbart ist.

Die Anwendung von parametrisierten Regeln kann mit Bedingungen an die Parameter verknüpft werden :

```
for all <var> such that <boolean expression> let <rule>
```

Beispiel :

```
for all x such that numberp x and x<0 let h(x)=0;
```

Mit Hilfe des `CLEAR` Befehls können Regeln wieder gelöscht werden. Für parametrisierte Regeln muß aber dabei derselbe Regelkopf (mit denselben Namen der Laufvariablen und denselben booleschen Bedingungen) angegeben werden :

```
clear x, h(x,y);
for all x such that numberp x and x<0 clear h(x);
```

Genauso überschreiben neue Regeln mit derselben oder algebraisch stärkeren linken Seite frühere Regeln. Eine Regel für `x**4` etwa überschreibt eine früher vereinbarte Regel für `x**5`.

3.6.3 Regellisten und lokale Simplifikation

Die modernere Form der Behandlung von Regeln steht seit REDUCE 3.4 mit *Regellisten* zur Verfügung. Sie vereinigen in sich die Möglichkeit der Definition von globalen Regeln und der Ausführung von (lokalen) Substitutionen.

Eine Regelliste ist eine Liste von Ausdrücken der Form

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

Zum Beispiel,

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

Die Tilde vor den Variablen auf der linken Seite hat eine ähnliche Bedeutung wie `FOR ALL X,Y ...` in den globalen Regeldefinitionen. Jede so markierte Variable wird als (lokale selbstdefinierende) Variable betrachtet (die eine gleichnamige Umgebungsvariable überdeckt).

Eine Regelliste kann einem Bezeichner zugewiesen werden :

```
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};
```

Regellisten können zur Installation globaler Regeln verwendet werden :

```
let trig1;
```

setzt die oben angegebenen Regeln zur globalen Simplifikation von Winkelfunktionen,

```
clearrules trig1;
```

löscht die vereinbarten Regeln wieder.

Regellisten können auch zur Definition *lokaler Simplifikationsregeln* herangezogen werden :

```
cos(a)*cos(b+c)
   where {cos(~x)*cos(~y) => (cos(x+y)+cos(x-y))/2};
```

```
        COS(A - B - C) + COS(A + B + C)
       --------------------------------
                       2
```

oder

```
cos(a)*cos(b+c) where trig1;
```

wendet die nach dem Schlüsselwort `WHERE` angegebene Regel oder Regelliste (geschweifte Klammern kann man auch hier wieder weglassen) *lokal* auf den Ausdruck vor `WHERE` an. Dabei haben die lokalen Regeln Vorrang vor den global definierten.

Die allgemeine Syntax lautet

```
<expression> WHERE <rule>|<rule list>(,<rule>|<rule list> ...).
```

`WHERE` hat geringere Bindungskraft als alle anderen Infixoperatoren (insbesondere auch `:=`, bindet aber stärker als `ELSE`, `THEN`, `DO`, `REPEAT` und so weiter.

```
if a=2 then 3 else a+2 where a=3
```

ist also äquivalent zu

```
if a=2 then 3 else (a+2 where a=3).
```

Dagegen weist

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

a nicht 2 zu, sondern b, weil der Ausdruck als `(a:=b) where b=>2` parst.

3.6.4 Interne Simplifikationsregeln

REDUCE hat für jeden internen Operator, den der algebraische Modus zur Verfügung stellt, bereits einige globale Simplifikatonsregeln vordefiniert. Der Stand dieser Regeln kann (seit REDUCE 3.5) durch den Operator `SHOWRULES <op>;` abgefragt werden. Er gibt eine Liste der für den entsprechenden Operator vereinbarten globalen Simplifikationsregeln zurück.

Die Reihenfolge, in der Regeln mit überlappenden Geltungsbereichen angewendet werden, ist kompliziert. Man muß deshalb davon ausgehen, daß ihre Reihenfolge zufällig ist. Betrachtet man die Regeln für *einen* Operator, so gilt :

1. Regeln mit wenigstens einem freien Parameter werden vor den Regeln ohne freien Parameter angewendet.

2. Regeln aus späteren `LET` Vereinbarungen werden eher angewendet.

Außerdem gibt es eine Reihe interner Regeln, die nicht mit dem externen `LET` Mechanismus behandelt werden. Hierbei handelt es sich vor allem um die Art und Weise, wie polynomiale (Teil)Ausdrücke in eine REDUCE interne Form überführt werden. Durch Setzen verschiedener Schalter kann man die Auswahl dieser Simplifikationsregeln steuern :

Schalter	Wirkung	Standard
`on combinelogs`	$log(x) + log(y) \mapsto log(x * y)$	off
`on combineexpt`	$x^a * x^b \mapsto x^{a+b}$	off
`on exp`	Klammern werden aufgelöst	on
`on expandlogs`	$log(x * y) \mapsto log(x) + log(y)$	off
`on ezgcd`	verwendet den Extended Zassenhaus GCD Algorithmus zur GCD-Bestimmung	off
`on factor`	Ausdrücke werden in faktorisierter Form dargestellt	off
`on gcd`	aus Zähler und Nenner werden gemeinsame Faktoren herausgekürzt	off
`on mcd`	Ausdrücke werden auf einen gemeinsamen Hauptnenner gebracht	on
`on precise`	$sqrt(-8*a\hat{}2*b) \mapsto 2*abs(a)*sqrt(-2*b)$ anstatt $2 * a * sqrt(-2 * b)$	off

Tabelle 15 : Schalter zur Steuerung des Simplifikationsverhaltens

3.7 Steuerung der Ausgabeformen

Die Ausgabe von Ausdrücken erfolgt standardmäßig in einem zweidimensionalen stark distributiven Format mit Zeilenumbruch, wobei die vorkommenden Kerne nach internen Regeln geordnet sind. Die Ausgabeform kann man durch eine Reihe von Parametern und Schaltern beeinflußen :

Kommando	Wirkung	Standard
`on allfac`	klammere monomiale Faktoren aus	on
`on div`	Nenner lokal zu jedem Term erzeugen	off
`factor x,y`	Ordne die Ausgabe nach Potenzen von x und y	
`linelength(l)`	Setzen der Zeilenlänge	72
`on list`	Jeden Term auf eine eigene Zeile schreiben	off
`off nat`	eindimensionale (und damit wieder einlesbare) Ausgaben erzeugen	on
`on nero`	Nullen werden in der Ausgabe von Listen unterdrückt	off
`off nosplit`	Zeilenumbruch erfolgt an willkürlichen Stellen und ist damit weniger zeitaufwendig	on
`order x,y,z`	die Kerne x, y, z haben Priorität	`order nil`
`on rat`	ähnliche Wirkung wie `on div`	off
`on ratpri`	Brüche mit maximal zeilenlangem Zähler und Nenner werden zweidimensional ausgegeben	on
`remfac x,y`	hebt die `factor`-Deklaration wieder auf	
`on revpri`	Die normale Termordnung (von hohen zu niedrigen Graden) wird umgekehrt	off

Tabelle 16 : Schalter zur Ausgabeformatierung

3.8 Dateiein- und -ausgabe

REDUCE bietet die Möglichkeit, Ein- und Ausgaben auch über Dateien durchzuführen. Solche Dateien müssen mit `;end;` enden.

`out <file name>`	Öffne die entsprechende Datei und lenke alle weiteren Ausgaben von REDUCE auf diese Datei um
`shut <file name>`	Schließe die Datei und lenke die REDUCE Ausgabe wieder auf die Standardausgabe um
`in <file name>`	Lenke die Standardeingabe auf die angegebene Datei um, bis sie zu Ende eingelesen ist

`in <file>;` gibt den Inhalt der eingelesenen Datei auf die Ausgabe, während `in <file>$` die Datei still einliest und nur die Ergebnisse der einzelnen Kommandos anzeigt (sofern nicht auch in der einzulesenden Datei die Kommandos mit $ abgeschlossen sind).

Explizite Mitteilungen kann man auf das Ausgabefile mit

```
write text1,text2,...;
```

ausgeben. `text` sind dabei Strings oder reguläre REDUCE Ausdrücke, die vor ihrer Ausgabe dann natürlich ausgewertet werden.

Beispiel :

```
a:=5;

        A := 5

write "a ist gleich ",a;

        a ist gleich 5
```

Eingabedateien können im wesentlichen zwei unterschiedliche Zwecke verfolgen. Zum einen kann es sich um eine Demonstrations- oder Arbeitsdatei handeln, deren zeilenweise Abarbeitung man verfolgen möchte. Zum anderen kann es sich um zusätzlich zu initialisierende Teile der REDUCE Umgebung handeln. Für erstere gibt es eine Reihe vom Möglichkeiten, die Abarbeitung zu beobachten. Für letztere besteht die Möglichkeit, sie in ein FASL-File zu übersetzen :

```
faslout <fasl name>$
in <Quellfilename>$
faslend$
```

Ein übersetztes File ist explizit mit `load_package <fasl name>` zu laden, um es im Programm zu verwenden.

Zur Behandlung von Ein- und Ausgabespezifika stehen folgende Schalter zur Verfügung :

`on demo`	Bei zeilenweiser Abarbeitung der Eingabedatei wird nach jedem Kommando ein `<et>` abgewartet
`pause;`	Bei diesem Befehl in der Eingabedatei wird dessen Abarbeitung suspendiert und zur Standardeingabe umgeschaltet
`cont;`	Die Eingabe wird von der Eingabedatei fortgesetzt
`off nat`	produziere lineare Ausgaben, die von REDUCE wieder eingelesen werden können
`on echo`	gibt ein Echo jeder Eingabe auf das Ausgabefile aus

Tabelle 17 : Dateiein- und Ausgabesteuerung

Kapitel 4

REDUCE-Pakete

Im Unterschied zu den meisten anderen kommerziellen Computeralgebrasystemen ist REDUCE ein vollständig offenes System :

- REDUCE ist in derselben Sprache RLISP geschrieben wie alle seine Applikationen.
- REDUCE wird traditionell mit allen Quelltexten ausgeliefert.
- Die meisten REDUCE Installationen bieten einen LISP-Compiler an, mit dem der Nutzer selbst effizienten Code erzeugen und in das FASL-Paketsystem integrieren kann.
- REDUCE erbt von LISP die Fähigkeit, Programmteile dynamisch zu laden und Funktionen (selbst im Kernbereich) zu überschreiben.

REDUCE bietet damit potentiellen Nutzern einen interessanten Rahmen an, der es gestattet, algorithmisierbares mathematisches Wissen aufzunehmen.

Diese Ausstattung eines Rahmens zur symbolischen Formelmanipulation mit *konkretem mathematischem Wissen*, das anderen Nutzern als eine *black box* zur Verfügung steht, ist zugleich eine der Hauptkomponenten, die ein Computeralgebrasystem für den breiteren Einsatz interessant macht. Besonders effektive Pakete entstehen dabei vor allem im Zusammenwirken von Wissenschaftlern mit guten theoretischen Kenntnissen der Materie und Spezialisten der Programmierung des Hostsystems.

In der REDUCE Nutzergemeinde sind diese Kontakte über die Zeit der Entwicklung des Systems organisch gewachsen. Es gibt einen großen Kreis von Nutzern und Autoren, die Teile der mathematischen Software von REDUCE entworfen und implementiert haben und heute mit der Weiterentwicklung des Systems mehr oder weniger eng verbunden sind. Diese Software ist teils in das System unmittelbar integriert, teils in einer umfangreichen Nutzerbibliothek enthalten.

Einige der Pakete werden automatisch vom Kern geladen, während andere vor ihrem Einsatz in Rechnungen explizit mit

```
load_package <package>;
```

geladen werden müssen.

Fast alle diese Pakete enthalten eine separate Dokumentation und Testdateien, die zusammen mit dem Quelltext Teil der REDUCE Distribution sind.

4.1 REDUCE Pakete

Es folgt eine Zusammenstellung der mit REDUCE 3.5 ausgelieferten Pakete einschließlich einer Kurzbeschreibung der darin implementierten Fähigkeiten. Nutzerpakete aus dem Verzeichnis `LIB` sind neueren Datums und bestehen dabei meist aus 4 Dateien :

`*.red`	REDUCE Quelltext des Pakets
`*.tex`	LATEX-Dokumentation
`*.tst`	Testdatei für das Paket
`*.log`	Protokoll der Ausgaben der Testdatei (zu Vergleichszwecken)

Nutzerpakete aus dem Verzeichnis `SRC` sind meist älteren Datums und damit stärker in die interne REDUCE Struktur einbezogen. Ihre Dokumentation ist mit im Verzeichnis `DOC` enthalten.

Paket	Kurzbeschreibung
ALGINT	Integration von Ausdrücken, die Quadratwurzeln enthalten
ARNUM	Rechnen mit algebraischen Zahlen, die durch ihr Minimalpolynom gegeben sind
ASSIST	kleines Ergänzungspaket zum algebraischen Modus
AVECTOR	Paket zur Vektoralgebra
CALI	Paket zur konstruktiven nichtlinearen Algebra
CAMAL	eine REDUCE Version des bekannten CAMAL-Programms zur Himmelsmechanik
CHANGEVR	Paket zur Variablentransformation in Differentialgleichungen
COMPACT	Anwendung von Regeln auf polynomiale Ausdrücke
CRACK	Paket zur Untersuchung partieller Differentialgleichungen
CVIT	Programm zur schnellen Berechnung der Spur von Diracschen Gammamatrizen
DESIR	Lösung homogener gewöhnlicher Differentialgleichungen mit polynomialen Koeffizienten
EXCALC	Paket für Rechnungen in der Differentialgeometrie
FIDE	Paket zur Anwendung der Methode der finiten Elemente zur Lösung partieller Differentialgleichungen
GENTRAN	Paket zur Erzeugung von FORTRAN, RATFOR oder C Code aus symbolischen REDUCE Ausdrücken
GHYPER	Paket zur Vereinfachung verallgemeinerter hypergeometrischer Funktionen
GNUPLOT	Graphikpaket, das ein GNU PLOT Interface herstellt
GROEBNER	Paket zum Lösen polynomialer Gleichungssysteme
IDEALS	Paket zur Idealtheorie und kommutativen Algebra
LAPLACE	Laplacetransformationen (keine Dokumentation)
LIE	Paket zur Klassifizierung niederdimensionaler reeller und komplexer Liealgebren

LIMITS	Paket zur Berechnung von Grenzwerten
LININEQ	Lösen von Systemen linearer Gleichungen und Ungleichungen
MEIJERG	Paket zur Vereinfachung von Meijers G-Funktion
NUMERIC	Implementation numerischer Algorithmen auf der Basis der reellen Arithmetik von REDUCE
ODESOLVE	Paket zum Lösen gewöhnlicher Differentialgleichungen
ORTHOVEC	Paket zur Vektoranalysis
PHYSOP	Paket zum Operatorkalkül in der Quantentheorie
PM	an den Stil von MATHEMATICA angelehnter Pattern Matcher
REACTEQN	Unterstützung für Berechnungen chemischer Reaktionskinetiken
RLFI	Paket zur Erzeugung von LATEX-Output
ROOTS	Paket zum Auffinden reeller oder komplexer Nullstellen von Polynomen
SCOPE	Paket zur Quellcodeoptimierung
SPDE	Paket zum Auffinden von Symmetrien in partiellen Differentialgleichungssystemen
SPECFN	SApezielle analytische Funktionen
SUM	Paket zur Berechnungvon Summationsformeln
SYMMETRY	Paket zur Darstellungstheorie kleiner symmetrischer Gruppen
TAYLOR	Paket zur Berechnung von Taylorreihen
TPS	Paket zur Manipulation unendlicher Potenzreihen
TRI	Erzeugung von REDUCE Output in TEX-Format
WU	Implementation des Wu-Algorithmus zum Lösen polynomialer Gleichungssysteme

Tabelle 18 : REDUCE Nutzerpakete

Kapitel 5

RLISP und der symbolische Modus

Der algebraische Modus von REDUCE stellt eine Oberfläche dar, die auf der eigentlichen Programmiersprache RLISP, dem *symbolischen Modus* aufsetzt. Diese ist ein in "normale" Syntax umgesetzter LISP-Dialekt. Die wichtigsten Ideen und syntaktischen Konstruktionen von RLISP sowie die Struktur der wichtigsten in REDUCE verwendeten Datentypen sollen im folgenden kurz vorgestellt werden.

Der Nutzer kann auch den Sprachumfang dieser Hostsprache direkt nutzen, wenn er mit `symbolic;` in den symbolischen Modus umschaltet. Viele Berechnungen sind in diesem Modus effizienter und flexibler ausführbar, wenn man mit der Programmiersprache umgehen kann, weil die Daten nicht zwischen den beiden Modi umgesetzt werden müssen.

`algebraic;` schaltet wieder in den algebraischen Modus zurück. Mit `eval_mode;` oder `lisp !*mode;` kann man den gerade gesetzten Modus erkennen.

5.1 Grundideen von LISP

Symbolische Rechnungen sind Rechnungen mit Ausdrücken, deren Größe und Komplexität im Vorhinein meist nicht abschätzbar ist und benötigen deshalb eine dynamische Speicherverwaltung. Daß symbolische Objekte selbst oft nur schwer in Klassen einteilbar sind, erhöht die Anforderungen zusätzlich.

LISP löst beide Probleme, indem es Zeigerpaare (und Atome) als einzige elementare Datenstrukturen zuläßt, aus denen binäre Bäume aufgebaut werden, in denen die symbolische Information kodiert werden muß. Sieht man einmal von den in den Blättern lokalisierten Atomen ab, so erreicht man damit ein homogenes Speichermodell, das dynamisch im Rahmen der üblichen Speicherverwaltungsmodelle (Garbage Collection) bearbeitet werden kann. Eine auf einer solchen Idee beruhende Programmiersprache ist dem Ansatz nach ungetypt und ermöglicht zugleich eine einheitliche Sicht auf Daten und Programme. Dies ist ein zusätzlicher Vorteil für die symbolische Programmierung, da auch hier eine strikte Trennung zwischen Daten und Programmen oft nicht möglich ist.

Um ein solches Speichermodell effizient zu implementieren, folgt LISP dem funktionalen Programmierparadigma. Sieht man einmal von wenigen destruktiven Prozeduren ab (die aus Effizienzgründen zur Verfügung gestellt werden), transformieren Funktionsaufrufe dabei alte Datenstrukturen in neue, ohne Seiteneffekte auf bestehenden Strukturen auszulösen. Damit kann man einfach Zeiger statt ganzer binärer Bäume kopieren, als Parameter an Prozeduren übergeben usw. Derselbe Zeiger kann also in verschiedenen LISP-Objekten auftreten. Dies

bedeutet zugleich, daß destruktive Prozeduren oftmals unerwartete Seiteneffekte nach sich ziehen und Werte vollkommen unbeteiligter Variablen beeinflussen können.

5.1.1 Atomare Datentypen

Bei atomaren Datentypen unterscheiden wir zwischen symbolischen Atomen, die einen Wert besitzen können, und atomaren Objekten wie z.B. Zahlen, die sich selbst zum Wert haben.

Symbole sind vielseitig verwendbare Objekte mit folgenden Attributen :

1. **Symbolname** :

   Jedes Symbol besitzt einen eindeutig bestimmten Namen, mit dem es identifiziert werden kann. Für die Namen von Symbolen in RLISP gilt das unter 3.2.1 über Bezeichner gesagte.

2. **Wert** :

   Ein Symbol kann auf ein beliebiges LISP-Objekt verweisen. Dieser Verweis wird in die *Wertzelle* eingetragen. Ein Objekt kann auch auf sich selbst zeigen.

3. **Funktionszelle** :

   Ein Symbol kann auf eine funktionale Vorschrift verweisen, die ausgeführt wird, wenn das Symbol syntaktisch in funktionaler Stellung angetroffen wird.

   Ein Symbol kann stets nur auf einen Wert *oder* eine Funktion verweisen.

4. **Eigenschaftsliste** :

   Ein Symbol besitzt eine Eigenschaftsliste, in der das Symbol betreffende Indikatoren und Flags gespeichert werden können.

Symbole sind damit im Gegensatz zu anderen Programmiersprachen *mehr* als nur symbolische Gegenwerte von Speicheradressen.

RLISP stellt als atomare Datentypen weiterhin *Zahlen* zur Verfügung, und zwar ausschließlich ganze und reelle Zahlen in der unter 3.2.1 beschriebenen Syntax. Aus diesen, Symbolen und den Grundrechenarten lassen sich andere Zahlbereiche konstruieren.

Ein dritter elementarer Datentyp sind schließlich Strings, die man in RLSIP mit `explode` und `compress` (vgl. [4]) auf Listen abbilden und dann geeignet manipulieren kann.

Daneben gibt es in verschiedenen LISP-Dialekten weitere Datentypen wie Felder, Vektoren, Hash-Tabellen, Streams usw.

5.1.2 Listen

Der elementare Datentyp in LISP ist das *Zeigerpaar* `(A . B)`, wobei A und B selbst wieder LISP-Ausdrücke sind. Bezeichnen wir das Paar mit P, so gibt es Selektoren `A:=CAR P` und `B:=CDR P`. Zusammensetzungen der Form `CADAR (=CAR CDR CAR)` bis zu vier Ebenen als Kurzform entsprechender Selektorfolgen werden normalerweise unterstützt. Ein typischer Ausdruck dieser Sprache hat also etwa die Gestalt

```
(((x . 2) . 1) . (((x . 1) . (((y . 1) . 2) . NIL )) . NIL ))
```

`NIL` ist hierbei ein spezieller Zeiger, der *Nullzeiger*.

Eine spezielle binäre Struktur, die *Liste*, spielt in LISP eine zentrale Rolle. Listen werden rekursiv definiert und für ihre Notation eine abkürzende Syntax verwendet :

> `NIL` ist eine Liste, die leere Liste. Man schreibt für sie auch `()`.
>
> Ist `L` eine Liste und `A` ein beliebiges Objekt, so ist auch `(A . L)` eine Liste. Wenn `L:=(B C D ...)`, so schreibt man kurz für die entstehende Liste `(A B C D ...)`.

`CAR L` verweist also auf das erste Listenelement, `CDR L` auf die Restliste.

Unser obiger Ausdruck würde in dieser Listennotation folgendermaßen aussehen :

```
(((x . 2) . 1) ((x . 1) ((y . 1) . 2)))
```

Eine allgemeine Regel zur Überführung von Ausdrücken der ersten in die zweite Art ist die folgende : Ersetzt man überall `NIL` durch `()`, so kann man rekursiv ein Klammerpaar einer Ebene zusammen mit einem davorstehenden Punkt weglassen.

LISP sieht eine Vielzahl von Prozeduren vor, die auf Listen operieren. Konzeptionell wird dabei das erste Listenelement als Name eines Operators verstanden, die restlichen Elemente als dessen Parameter.

Eine spezielle Unterklasse von Listen sind die *eigentlichen Listen*, deren Elemente Atome sind. Andere Listen heißen auch *verschachtelte Listen*.

Indikatoren sind normalerweise als eine weitere spezielle Listenart gespeichert, als *Assoziationslisten*. Das sind Listen von Zeigerpaaren, wobei normalerweise die CAR-Teile einen Identifikator und die CDR-Teile die zugehörige Information enthalten.

5.2 Die Syntax von RLISP

Für Steuerstrukturen und Verbundanweisungen folgt RLISP denselben Syntaxregeln wie oben für den algebraischen Modus beschrieben. Es gibt ebenfalls ein *Prozedurkonzept*. In diesem Zusammenhang muß man allerdings unterscheiden, in welchem der Modi die definierte Prozedur zu interpretieren ist. Da auch der algebraische Modus als Interface viele symbolische Prozeduren verwendet, kann eine solche Unterscheidung nicht aus dem Kontext heraus erfolgen. Statt dessen verwendet REDUCE ein allgemeineres Prozedurkonzept als in 3.3.7 beschrieben :

```
<proctype> PROCEDURE <name>(<var_1>,...,<var_n>);<statement>;
```

`<proctype>` ist dabei entweder `symbolic` oder `algebraic`. Im entsprechenden Modus wird die jeweilige Erweiterung als Standardannahme gesetzt, wenn eine Prozedurtypspezifikation fehlt. Daneben gibt es noch andere (symbolische) Prozedurtypen, insbesondere vom Typ `MACRO`, auf die hier nicht weiter eingegangen werden soll, und vom Typ `SMACRO`. Letzterer entspricht dem aus anderen Programmiersprachen bekannten Makrokonstrukt, das während des Einlesens von Quelltext, d.h. noch vor der Auswertung, automatisch expandiert wird.

In beiden Modi werden die Argumente von links nach rechts ausgewertet (call by value) und dann der Prozedurkörper, in dem wie in 3.3.7 beschrieben auch lokale Variable verwendet werden können, abgearbeitet.

Im folgenden sind weitere wichtige syntaktische Elemente von RLISP zusammengestellt. Für eine vollständigere Einführung vgl. [4].

5.2.1 Datentypen und Auswertung

Datentyp	Bezeichnung
Atom	*atom*
beliebiger Ausdruck	*any*
boolescher Ausdruck	*boolean*
Bezeichner	*id*
Float-Zahl	*float*
Funktionszeiger	*fpointer*
ganze Zahl	*integer*
(t, nil)	*boolean*
Vektor	*vector*
Zeigerpaar	*pair*
Liste	*list*
eigentliche Liste	*p-list*
Assoziationsliste	*a-list*

Tabelle 19 : Datentypen in RLISP

Im Gegensatz zum algebraischen Modus werden im symbolischen Modus bei Zuweisungen, Funktionsaufrufen usw. die Argumente *genau* einmal ausgewertet, sofern sie nicht durch `quote` geschützt sind. Man beachte, daß an vielen Stellen die Variable selbst und nicht ihr Wert benötigt wird und deshalb quotierte Ausdrücke ein wichtiges Programmgestaltungsmittel sind. Durch `eval` kann man weitere Auswertungen bewirken.

```
a:=quote x;     % Weist A das Symbol X als Wert zu

        X

x:=2;           % Weist X den Wert 2 zu a;

        2

a;

        X

eval a;

        2
```

Prozedur	Wirkung	Abkürzung
eval*(x:any):any*	weitere Auswertung	
evlis*(u:list):list*	alle Elemente einer Liste auswerten	
quote*(x:any):any*	Auswertung verhindern	`'x`
setq*(x:id,y:any)*	Zuweisung	`x:=y`
set*(x:id,y:any)*	Zuweisung, die auch x auswertet	

Tabelle 20 : Steuerung der Auswertung

5.2.2 Listen

Operator	Wirkung
append*(l1,l2:list):list*	Listen aneinanderfügen
delete*(p:any,l:list):list*	Element p aus der Liste l herauslöschen
length*(l:list):integer*	Länge der (top level Elemente) der Liste l
list*(e1,e2,...):list* oder {*e1,e2,...*}	Liste aus Elementen zusammenstellen
nth*(l:list,n:integer):any*	n-tes Listenelement
pair*(l1,l2:list):list*	Zwei gleichlange Listen $l1$ und $l2$ werden zu einer Liste von Zeigerpaaren elementweise zusammengesetzt
reverse*(l:list):list*	Reihenfolge der Listenelemente umkehren

Tabelle 21 : Operatoren auf Listen

Beispiele :

```
l1:={1,2,3};

        (1 2 3)

l2:='(a b c);

        (A B C)

length l1;

        3

reverse l2;

        (C B A)

delete('b,l2);

        (A C)

u:=pair(l1,l2);

        ((1 . A) (2 . B) (3 . C))

assoc(2,u);

        (2 . B)
```

Bei diesen Operationen werden die Listen $l, l1, \ldots$ nicht verändert. Tritt als Ergebnis eine Liste auf, so ist dies eine neue Liste, aber unter Verwendung der Zeiger auf die alten Listenelemente. Stattdessen kann man (aus Effizienzgründen) dieselben Operationen auch auf der Originalliste

ausführen, womit diese modifiziert wird. Wegen der damit verbundenen Nebeneffekte sollte man diese Konstrukte allerdings sparsam verwenden und ihre Konsequenzen genau übersehen können.

car*(u:pair):=v:any*	Ersetzen des CAR-Teils von u durch v
cdr*(u:pair):=v:any*	Ersetzen des CDR-Teils von u durch v
nconc*(l1,l2:list):list*	physisches Verketten der Listen $l1$ und $l2$
reversip*(l:list):list*	destruktives Invertieren der Liste l

Tabelle 22 : Destruktive Listenoperatoren

5.2.3 Variablentypen

Variablen im algebraischen und symbolischen Modus gehören grundsätzlich verschiedenen Namensräumen an und können nicht ohne weiteres im jeweils anderen Modus verwendet werden.

Für den symbolischen Modus stehen in RLISP neben *lokalen Variablen* innerhalb eines Blocks oder der Parameterliste eines Prozeduraufrufs (die wie in 3.3.7 beschrieben durch `scalar, integer, real` deklariert werden können und entsprechend mit `NIL` oder 0 vorinitialisiert sind) verschiedene Klassen von *globalen Variablen* zur Verfügung :

- Die Typen `global` und `fluid` definieren beide global verwendbare Variable, die den üblichen Sichtbarkeitsregeln von Bezeichnern unterworfen sind.

  Einer allgemeinen Konvention folgend sind systeminterne Variablen vom Typ `global` meist mit Bezeichnern versehen, die auf `!*` enden.

- Der Typ `shared` definiert globale Variable, die auf gleiche Weise im symbolischen wie im algebraischen Modus zur Verfügung stehen. Damit verbunden ist allerdings ein in Details abweichendes Auswertungsverhalten im algebraischen Modus.

- Der Typ `switch` definiert Schalter. Mit jedem deklarierten Schalter `switch neu;` wird eine globale Variable `!*neu` verbunden, die den aktuellen Schalterwert (T oder NIL) speichert. Dies gilt ebenfalls für alle im System definierten Schalter.

Variablen jedes dieser Typen sind vor ihrer Verwendung zu deklarieren, womit sie auch mit NIL initialisiert werden :

```
fluid '(x y z);
global '(a b c);
switch sw1, sw2, sw3;
share s1, s2;
```

Nichtdeklarierte Variablen werden im Interpreterdialogmodus automatisch dem Typ `fluid` zugeordnet.

5.2.4 Funktionen als Prozedurparameter

Eine Besonderheit von LISP stellt die gleichberechtigte Behandlung von Daten und Funktionen dar. Entsprechend können ohne jede Schwierigkeit auch Funktionen als Parameter an Prozeduren übergeben werden. Die wichtigsten Sprachkonstrukte für eine solche Anwendung sind die aus der `map` und der `apply` Klasse.

apply*(fn:function,l:list)*

Wende die Funktion fn auf die Argumentliste l an (deren Länge der Arität von fn entsprechen muß)

apply1*(fn:function,arg:any)*

Berechne $fn(arg)$ für die einstellige Funktion fn

apply2*(fn:function,arg1,arg2:any)*

Berechne $fn(arg1, arg2)$ für die zweistellige Funktion fn

apply3*(fn:function,arg1,arg2,arg3:any)*

Berechne $fn(arg1, arg2, arg3)$ für die dreistellige Funktion fn

mapcar*(l:list,fn:function):list*

Wende auf jedes der Elemente der Liste l die (einstellige) Funktion fn an

Daneben gibt es noch eine ganze Reihe anderer `map...` Operatoren, vgl. [4].

Man beachte, daß Funktionen als Parameter stets über ihre Namen referenziert werden, d.h. beim Aufruf nicht ausgewertet werden dürfen. Eine spezielle Quotierung erreicht man hier mit dem Schlüsselwort `function` statt `quote`. Eine Anwendung auf durch Makros definierte Funktionen (dies sind die meisten RLISP Funktionen mit variabler Parameteranzahl wie `plus, times`) ist nicht möglich.

Beispiele :

```
l:='(1 3);

        (1 3)

apply(function plus2,l);

        4

mapcar(l,function add1);

        (2 4)
```

5.2.5 Verwaltung der Eigenschaftenliste

Die Eigenschaftenliste eines Symbols kann Flags und Indikatoren des Symbols sammeln. *Flags* können nur einen Wert annehmen und deshalb nur gesetzt oder nicht gesetzt sein. *Indikatoren* dagegen können mehrere verschiedene Werte annehmen und bestehen deshalb aus einem *Attribut* und einem Wert. Die Eigenschaftenliste eines Bezeichners `m` kann mit `prop 'm;` eingesehen werden. Mit einem Symbol verbundene Funktionsdefinitionen werden auf ähnliche Weise wie Eigenschaften zugewiesen, abgefragt und entfernt und deshalb im folgenden mit aufgeführt :

Operator	Wirkung
flag*(u:id-list,f:id)*	Setzen des Flags f für jedes Symbol der Liste u
flagp*(u:id,f:id):boolean*	Abfragen des Flags f auf dem Symbol u
get*(u:id,p:id)*	Rückgabe des Werts des Attributs p auf der Eigenschaftsliste von u (oder NIL)
getd*(u:id)*	`(<Typ> . <Code>)`, wenn u ein Funktionszeiger ist, sonst NIL
put*(u:id,p:id,v:any)*	Vermerke auf der Eigenschaftsliste von u unter dem Attribut p der Wert v
putd*(u:id,t:type,c:code)*	Verwandle das Symbol u in einen Funktionszeiger vom Typ t, der auf den Code c verweist
remd*(u:id)*	Entferne die Funktionsdefinition, auf die u verweist.
remflag*(u:id-list,f:id)*	Entferne das Flag f von jedem Element der Liste u
remprop*(u:id,p:id)*	Entferne das Attribut p von u

Tabelle 23 : Verwaltung der Liste der Eigenschaften und von Funktionsdefinitionen

Beispiele :

```
flag('(m n),'newflag)$
flagp('m,'newflag);

        T

u:=for i:=1:5 collect i;

        (1 2 3 4 5)

put('m,'indices,u);

        (1 2 3 4 5)

prop 'm;

        ((INDICES 1 2 3 4 5) NEWFLAG )

get('m,'indices);

        (1 2 3 4 5)

remflag('(m),'newflag)$
remprop('m,'indices)$
```

5.2.6 Boolesche Operatoren

Sämtliche im algebraischen Modus unter 3.2.4 und 3.2.5 beschriebenen booleschen Operatoren stehen ebenfalls im symbolischen Modus zur Verfügung und können (da ihre Werte T oder NIL legale symbolische Werte sind) auch außerhalb von Steuerkonstrukten verwendet werden. Darüberhinaus gibt es eine Reihe weiterer Operatoren zur Typbestimmung :

Operator	Wirkung
atom u	`true`, wenn u kein Paar ist
codep u	`true`, wenn u ein Funktionszeiger ist
fixp u	`true`, wenn u eine ganze Zahl ist
floatp u	`true`, wenn u eine Float-Zahl ist
fluidp u	`true`, wenn u eine Fluid-Variable ist
globalp u	`true`, wenn u eine Global-Variable ist
idp u	`true`, wenn u ein Bezeichner ist
listp u	`true`, wenn u eine Liste ist
minusp u	`true`, wenn u eine negative Zahl ist
null u	`true`, wenn `u eq NIL`
numberp u	`true`, wenn u eine ganze oder reelle Zahl ist
pairp u	`true`, wenn u ein Paar ist
stringp u	`true`, wenn u ein String ist
vectorp u	`true`, wenn u ein Vektor ist
zerop u	`true`, wenn $u = 0$

Tabelle 24 : Operatoren zum Testen von Datentypen

Während im algebraischen Modus im wesentlichen nur Zahlenwerte verglichen werden können, gibt es im symbolischen Modus auch die Möglichkeit, Strukturen auf Gleichheit zu testen. Dabei kann man, in der Hoffnung, auf gleiche Strukturen auch mit gleichen Zeigern zu verweisen, den (billigen) Vergleich der Zeiger vornehmen, aber auch den (teureren) Vergleich der Werte selbst. Ersteres ist insbesondere anwendbar, wenn man Atome vergleicht oder aber eine kanonische Darstellung der verwendeten LISP-Objekte vorliegt, d.h. auf jedes der untersuchten Objekte zeigt genau ein Zeiger.

Operator	Wirkung
assoc*(u:any,v:a-list):any*	erstes Element der Assoziationsliste mit `car equal u` oder NIL
atsoc*(u:any,v:a-list):any*	erstes Element der Assoziationsliste mit `car eq u` oder NIL
eq(a,b)	Test auf Gleichheit der Zeiger
eqn(a,b)	Vergleich auch von Zahlen
equal(a,b)	Test auf Gleichheit der Werte
memq*(a:any,b:list)*	`eq`-Test, ob a in der Liste b enthalten ist
member*(a:any,b:list)*	`equal`-Test, ob a in der Liste b enthalten ist

Tabelle 25 : Operatoren zum Strukturvergleich

5.2.7 Arithmetik

RLISP stellt im Gegensatz zum algebraischen Modus natürlich nur eine Arithmetik für ganze und Float-Zahlen zur Verfügung. Die bekannten Infixoperatoren werden dabei ebenfalls durch den Parser in Präfixoperatoren umgesetzt :

Operator	Zahl der Argumente	Infixnotation	Beschreibung
abs u	1		Absoluter Betrag
add1 u	1		$u+1$
difference(u,v)	2	u-v	Differenz
divide(u,v)	2		Paar `(quotient . remainder)`
expt(u,v)	2	u**v	Potenz (mit automatischer Typumwandlung $integer \mapsto float$, wenn erforderlich)
float u	1		Fließkommawert von u
fix u	1		ganzzahliger Anteil von u
max2(u,v)	2		Maximum
max(u1,u2,...)	bel.		Maximum (Makro)
min2(u,v)	2		Minimum
min(u1,u2,...)	bel.		Minimum (Makro)
plus2(u,v)	2	u+v	Summe
plus(u1,u2,...)	bel.	u1+u2+...	Summe (Makro)
quotient(u,v)	2	u/v	Ganzer Teil der Division u/v
remainder(u,v)	2		Rest bei der Division u/v
sub1 u	1		$u-1$
times2(u,v)	2	u*v	Produkt
times(u1,u2,...)	bel.	u1*u2*...	Produkt (Makro)

Tabelle 26 : Arithmetische Operatoren

5.2.8 Ausgabe

RLISP bietet neben den im Kapitel über den algebraischen Modus diskutierten Möglichkeiten der Ein- und Ausgabe von Dateien wie jede Programmiersprache die Möglichkeit, Ein- und Ausgabe über die Verwaltung von Dateihandles zu steuern. Darauf soll hier nicht eingegangen werden, vgl. [4].

Zur intermediären Textausgabe stehen folgende Operatoren zur Verfügung :

Operator	Wirkung
prin1 u	u ohne abschließenden Zeilenvorschub ausgeben
print u	u mit abschließendem Zeilenvorschub ausgeben
terpri()	Zeilenvorschub
write u1,u2,u3	$u1, u2, u3$ (ohne Zwischenräume) ausgeben

Tabelle 27 : Textausgabe in RLISP

5.2.9 Fehlerbehandlung in RLISP

Für einen Interpreter ist es wichtig, bei Fehlern den gerade laufenden Interpreterzyklus abzubrechen. Für eigene Prozeduren muß es deshalb Möglichkeiten des Fehlerabbruchs geben.

RLISP bietet dafür verschiedene Operatoren :

`rederr <string>`

: Abbruch der Prozedur mit einer (Fünf-Sterne-)Fehlermeldung `<string>`.

`rerror(<paketname>,nr,<string>)`

: Modernere Form von `rederr` für die Verwendung in Paketen. `nr` ist eine fortlaufende Nummer, die (später einmal) Debugging erleichtern soll.

`typerr(id,<string>)`

: Abbruch der Prozedur mit einer (Fünf-Sterne-)Fehlermeldung `<id> invalid as <string>`.

Zur Behandlung von Fehlern im Prozeß der Fehlersuche vgl. 3.2.10.

5.2.10 Betriebsmittelverwaltung

RLISP stellt einige Befehle zur Verwaltung von Betriebsmitteln wie Stackgröße, Heapgröße usw. zur Verfügung. Mit `system()` kann sogar eine Shell eröffnet werden, in der andere Programme zwischenzeitlich ausgeführt werden können. Weiterhin sind Befehle etwa zum Verzeichniswechsel möglich, die in derselben Shell ausgeführt werden müssen, um innerhalb von REDUCE wirksam zu sein.

Die folgenden Operatoren sind allerdings nicht auf allen Plattformen verfügbar bzw. haben wegen unterschiedlicher Konzepte der darunterliegenden Betriebssysteme unterschiedliche Bedeutung (so erfolgt Zeitmessung unter UNIX als CPU-Zeitmessung, unter DOS dagegen als Echtzeitmessung)

Operator	Wirkung
`system(<prog>)`	Start einer Shell mit dem Programm *prog*. Fehlt das Programm, wird eine interaktive Shell gestartet (die mit `exit` zu verlassen ist)
`cd <string>`	Wechsel des Verzeichnisses
`pwd()`	Ausgabe des aktuellen Verzeichnisses
`getenv`*(s:string):string*	Wert der Environmentvariable s
`setenv`*(s1,s2:string)*	der Environmentvariablen $s1$ wird der Wert $s2$ zugewiesen

Tabelle 28 : UNIX-Befehle in REDUCE

Operator	Wirkung
`time()`:*integer*	kumulative Rechenzeit (in ms.)
`gctime()`:*integer*	kumulative Zeit (in ms.) für garbage collection
`date!-and!-time()`:*string*	Ausgabe von Datum und Uhrzeit
`set_heap_size <nnn>`	Heapgröße neu setzen
`set_heap_size nil`	aktuelle Heapgröße ausgeben
`set_bndstk_size <nnn>`	Bindungsstackgröße setzen

Tabelle 29 : Verwaltung von Systemressourcen

Heap- und Stackgröße werden dabei in der Anzahl von Zellen angegeben, die je nach Betriebssystem entweder 4 oder 8 Bytes umfassen.

5.3 Grundlegende Konzepte und Datentypen für das symbolische Rechnen

Im vorhergehenden Abschnitt wurden die wichtigsten syntaktischen Konstrukte von RLISP besprochen. Sie bilden die Grundlage für die Darstellung symbolischer Information, die, wie nicht anders zu erwarten, in Form von geschachtelten Listen erfolgt. Obwohl LISP eine ungetypte Sprache ist, kann man deshalb verschiedene Datentypen auch auf der symbolischen Ebene durch die Verschiedenheit der ihnen zugeordneten Struktur der binären Bäume (wenigstens auf einer semantischen Ebene) unterscheiden. Dies sind

die algebraische Präfixform
: als eine universelle Form der symbolischen Darstellung mathematischer Formeln als geschachtelte Listen,

Standardformen
: als die interne kanonische Standarddarstellung polynomialer Ausdrücke und

Standardquotienten
: als der interne Datentyp, in den alle rationalen Ausdrücke transformiert werden

Da es aus Effizienzgründen nicht geraten ist, jeden intermediären Ausdruck wieder in die algebraische Präfixform zu verwandeln, spielen die Standardquotienten und als ihr wichtigster Bestandteil die Standardformen in REDUCE eine zentrale Rolle bei der Konstituierung zusammengesetzter Datentypen. Sie dienen dazu, Teile derselben quasi in *kompilierter* Form zu speichern.

REDUCE versucht dabei, diese kompakte Darstellung von rationalen Ausdrücken so lange als möglich in einer speziellen alegbraischen Datenform, der *SQ-Form* zu halten. Diese hat die Syntax

```
SQ-Form ::= (!*SQ <standard quotient> <ind>)
```

`!*SQ` ist dabei ein Typidentifikator und `<ind>` eine Variable, die den Wert T oder NIL hat. Der Standardquotient u in diesem Ausdruck liegt dann bereits in "kompilierter" Form vor und kann unmittelbar für Rechnungen in der internen Struktur verwendet werden. Allerdings könnte es sein, daß seit der Erzeugung von u die Variablenordnung oder andere die interne Form beeinflussende Parameter gewechselt wurden und die gespeicherte Form von u nicht mehr der aktuellen Form entspricht. Um dies zu überwachen wird das dritte Element der Liste benötigt. Genauer gesagt enthält der CDDR einer SQ-Form eine globale Variable `!*sqvar!*`, die als Wert die Liste (T) hat. Mit einer signifikanten Änderung der Umgebung, die eine Neubewertung der kompilierten Formen notwendig macht, wird diese Variable, die *allen* SQ-Formen gemeinsam ist, durch den Aufruf der destruktiven Listenoperation `rmsubs()` auf (NIL) gesetzt. Damit sind alle bisherigen SQ-Formen als reevaluierungsbedürftig angezeigt.

Für neue SQ-Formen wird im weiteren eine neue gemeinsame Variable `!*sqvar!*` verwendet. Dies ist zugleich ein Beispiel, wie eine destruktive Listenoperation, die Seiteneffekte auf verschiedene Zeiger hat, gezielt eingesetzt werden kann.

Der Konstruktor `mk!*sq <sq>` überführt Standardquotienten in SQ-Formen.

Auf der Basis der Standardformen ist die gesamte interne Polynomarithmetik implementiert. Dabei werden (unter `off factor`) alle Polynome in eine kanonische expandierte Form überführt, so daß zwei Polynome genau dann gleich sind, wenn sie durch gleichartige binäre Bäume dargestellt werden. Ein solches Simplifikationsverfahren für dieses zentrale Teilgebiet symbolischer Rechnungen, das nicht auf Patternmatching beruht, sichert die nötige Effizienz symbolischer Formelmanipulationen von Ausdrücken, die polynomiale Bestandteile haben.

5.3.1 Die algebraische Präfixform

Die algebraische Präfixform ist eine LISP-konforme Darstellung symbolischer Ausdrücke. Wandelt man alle in einem symbolischen Ausdruck vorkommenden Infixoperatoren in ihre Präfixäquivalente um, so hat ein symbolischer Ausdruck stets die Form

```
<operator>(<Liste der Argumente>)
```

wobei die Argumente Variable, Zahlen oder selbst wieder symbolische Ausdrücke sein können. Dieser Ausdruck wird in die Form

```
(<operator> . <Liste der Argumente>)
```

umgewandelt. Ist `f` ein Bezeichner eines Objekts im *algebraischen Modus*, so bekommt man mit `reval 'f;` dessen algebraisches Präfixäquivalent.

Beispiele :

```
algebraic$
f:=(x+y+z)^2;

      2                     2            2
F := X  + 2*X*Y + 2*X*Z + Y  + 2*Y*Z + Z

g:={x^2+y^2,x^2+z^2,y^2+z^2};

       2    2  2    2  2    2
G := {X  + Y ,X  + Z ,Y  + Z }

symbolic$
u:=reval 'f;

(PLUS (EXPT X 2) (TIMES 2 X Y) (TIMES 2 X Z) (EXPT Y 2) (TIMES 2 Y Z)
(EXPT Z 2))

reval 'g;

(LIST (PLUS (EXPT X 2) (EXPT Y 2)) (PLUS (EXPT X 2) (EXPT Z 2)) (PLUS
(EXPT Y 2) (EXPT Z 2)))
```

```
prop 'g;

((AVALUE LIST (LIST (!*SQ ((((X . 2) . 1) ((Y . 2) . 1)) . 1) T) (!*SQ
((((X . 2) . 1) ((Z . 2) . 1)) . 1) T) (!*SQ ((((Y . 2) . 1)
((Z . 2) . 1)) . 1) T))) (RTYPE . LIST))
```

An diesem Beipiel sieht man zugleich, daß Ausdrücke intern nicht in ihrer Präfixform, sondern in einer teilsimplifizierten Form gespeichert werden.

Die algebraische Präfixform stellt ein universelles Interface auch zwischen verschiedenen REDUCE Paketen dar. Insbesondere kann man sie als Interface zur Formelausgabe verwenden. Ein (syntaktisch sinnvoller) Ausdruck u in Präfixform kann mit `mathprint u;` diesem Paket übergeben werden.

```
mathprint u; % vgl. Zuweisung im vorigen Beispiel

 2                    2           2
X  + 2*X*Y + 2*X*Z + Y  + 2*Y*Z + Z
```

5.3.2 Kerne

Symbolische Rechnungen produzieren oft polynomiale Ausdrücke nicht in Variablen, sondern in allgemeineren Größen.

```
algebraic$
operator x;
f:=(sin(x)^2+tan(x)^2+cot(x)^2);

           2         2         2
F := SIN(X)  + TAN(X)  + COT(X)

lisp$
reval 'f;

(PLUS (EXPT (SIN X) 2) (EXPT (TAN X) 2) (EXPT (COT X) 2))

prop 'f;

((AVALUE SCALAR (!*SQ (((((SIN X) . 2) . 1) (((TAN X) . 2) . 1)
(((COT X) . 2) . 1)) . 1) T)))
```

Solche verallgemeinerte Variablen heißen *Kerne*. Sie haben die RLISP-Gestalt

```
Kern ::= Bezeichner | (Bezeichner . Liste von Argumenten)
```

Allerdings erfüllen Kerne darüberhinaus eine Eindeutigkeitseigenschaft, die im folgenden erklärt werden soll :

```
algebraic$
g:=x(1);
```

```
lisp;
u:=reval 'g; v:='(x 1);

        (X 1)

        (X 1)

equal(u,v);

        T

eq(u,v);

        NIL
```

Dieses Beispiel zeigt, daß Ausdrücke gleicher Gestalt unter verschiedenen Zeigern abgelegt sein können. Damit muß der aufwendigere `equal`-Test für ihren Vergleich ausgeführt werden. Da Kerne oft identifiziert werden müssen, ist es deshalb wichtig, sie eindeutig adressieren zu können. Dies wird durch das Anlegen einer Liste der verwendeten Kerne erreicht. Diese wird beim Erzeugen eines neuen Kerns untersucht, ob ein solcher Kern bereits existiert. Je nachdem, ob dies der Fall ist oder nicht, wird der neue Kern an die Liste angefügt oder aber die bereits bekannte Adresse des alten Kerns erneut verwendet. Damit kann man so definierte Kerne mit `eq, memq, atsoc` statt `equal, member, assoc` vergleichen.

Kerne sind linear bzgl. einer internen Ordnung geordnet, die sich grob an der lexikographischen Ordnung orientiert. Ähnlich dem algebraischen `korder` kann man diese etwa durch

```
setkorder '(x y z);
```

dahingehend modifizieren, daß x, y, z superior zu allen anderen Kernen angeordnet werden.

Für die Verarbeitung von Kernen stehen folgende Operatoren und globale Objekte zur Verfügung :

!*a2k*(k:kernel-expression):kernel*	gibt k als Kern zurück
setkorder*(l:list of kernel):list of kernel*	setzt die Kerne aus der Liste l als superiore Kerne und gibt die Liste der vorher gesetzten Kerne zurück
depl!**:list list kernel*	Liste der mit `depend` vereinbarten Abhängigkeiten zwischen Kernen
kord!**:list kernel*	Liste der superior angeordneten Kerne

Tabelle 30 : Kerne – Operatoren und globale Variable

5.3.3 Standardformen

Ein integraler Bestandteil der teilsimplifizierten Form von Ausdrücken sind die Standardformen, in denen polynomiale Ausdrücke gespeichert sind, wobei die Rolle von Variablen allgemeiner durch *Kerne* wahrgenommen wird.

REDUCE verwendet dafür eine rekursive Polynomdarstellung, die sich an der Ordnung der Kerne orientiert, d.h. der Polynomring $\mathbf{Z}[x_1, \ldots, x_n]$ wird als $(\mathbf{Z}[x_1, \ldots, x_{n-1}])[x_n]$ dargestellt. Ein Polynom ist also eine Linearkombination von Potenzen der höchsten Variablen in der gegebenen Kernordnung, der *Hauptvariablen*, mit Koeffizienten aus einem (rekursiv bereits darstellbaren) Polynomring in einer geringeren Zahl von Variablen oder aber den ganzen Zahlen als *Grundring*. Auch andere Zahlbereiche sind als Grundbereich möglich, siehe 5.4.1. Ein entsprechendes Element bezeichnet man als *Domainelement*.

Eine solche Linearkombination ist als Liste von *Standardtermen* abgelegt, die nach fallender Potenz der Hauptvariablen geordnet ist. Jeder Standardterm besteht aus einer Standardpotenz und einem Koeffizienten, der wieder eine Standardform (den Fall Grundring eingeschlossen) darstellt. Eine Standardpotenz schließlich besteht aus einem Kern, der Hauptvariablen, und einer natürlichen Zahl, dem Exponenten. Die formale Syntax ist damit die folgende :

```
<standard form> ::=
        NIL | domain element | ( <standard term> . <standard form> )
<standard term> ::= ( <standard power> . <standard form> )
<standard power> ::= ( <kernel> . <natural number> )
```

Ein Polynom

```
<lc> * <mvar>**<ldeg> + <red>
```

ist also intern als `(((mvar . ldeg) . lc ) . red )` dargestellt.

Auf die einzelnen Teile einer Standardform $p := \sum_{i=0}^{d} a_i x^i$ kann man mit den folgenden Selektoren zugreifen :

lt p	Leitterm $a_d x^d$ von p
lc p	Leitkoeffizient a_d von p
ldeg p	Leitgrad d von p
mvar p	Hauptvariable x von p
red p	Reduktum $p - lt\ p$

Tabelle 31 : Selektoren für Standardformen

Diese Selektoren sind aber nur für Standardformen definiert, die kein Domainelement sind.

Auf den Standardformen wird die gesamte Polynomarithmetik ausgeführt. Dabei wird stets vorausgesetzt, daß Standardformen als Argumente bzgl. der Kernordnung aufgebaut sind.

addf*(f1,f2:sf):sf*	$f1 + f2$
difff*(f1:sf,x:kernel):sf*	Ableitung von f nach x
domainp*(f:sf):sf*	Test, ob f ein Domainelement ist
exptf*(f:sf,n:integer):sf*	f^n
fctrf*(f:sf):factorlist*	Faktorzerlegung von f
gcdf*(f1,f2:sf):sf*	größter gemeinsamer Teiler
!*k2f*(k:kernel):sf*	Kern in eine Standardform verwandeln
multf*(f1,f2:sf):sf*	$f1 * f2$
negf*(f:sf):sf*	$-f$
null*(f:sf):sf*	Test, ob $f = 0$ (genauer, ob $f = NIL$)
numr simp*(a:pf):sf*	Präfixform in Standardform verwandeln
prepf*(f:sf):pf*	Standardform in Präfixgorm verwandeln
prim!-part*(f:sf):sf*	kürze aus f den gcd der Koeffizienten heraus
printf*(f:sf):sf*	Standardform ausdrucken
quotf*(f1,f2:sf):sf*	Exakter Quotient $f1/f2$ oder NIL
qremf*(f1,f2:sf):sf*	(quotient . remainder) der Division f1:f2
reorder*(f:sf):sf*	Neuordnung einer Standardform nach Wechsel der Kernordnung

Tabelle 32 : Operatoren auf Standardformen

`fctrf f` liefert dabei eine Liste der Form

```
factorlist ::= ( <integer factor> . <list of polynomial factors> )
```

zurück, wobei die Liste polynomialer Faktoren aus Elementen der Form

```
( <polynomial factor> . <multiplicity> )
```

besteht.

5.3.4 Standardquotienten

Ein Standardquotient ist ein Paar von Standardformen `(z . n)`, das den rationalen Ausdruck $\frac{z}{n}$ repräsentiert. Standardquotienten sind der in REDUCE am häufigsten verwendete interne Datentyp. In der folgenden Tabelle sind die wichtigsten Selektoren und Operatoren für Standardquotienten zusammengefaßt :

numr*(q:sq):sq*	Zähler von q
denr*(q:sq):sq*	Nenner von q
!*f2q*(f:sf):sq*	Standardform f in Standardquotienten verwandeln
simp*(a:pf):sq*	algebraische Präfixform a in Standardquotienten verwandeln
prepsq*(q:sq):pf*	Standardquotienten in algebraische Präfixform verwandeln
addsq*(q1,q2:sq):sq*	$q1 + q2$
negsq*(q:sq):sq*	$-q1$
sqprint*(q:sq)*	Ausgabe in math. Notation (nur unter `off nat`)
subtrsq*(q1,q2:sq):sq*	$q1 - q2$
multsq*(q1,q2:sq):sq*	$q1 * q2$
quotsq*(q1,q2:sq):sq*	$q1/q2$

Tabelle 33 : Selektoren und Operatoren für Standardquotienten

5.3.5 Substitutionsoperatoren

Substitutionen in allgemeinen Ausdrücken sind über den Operator `subeval1` möglich, der Argumente in algebraischer Präfixform erwartet. Eine *Substitutionsliste* ist dabei eine Assoziationsliste, deren CAR-Teil aus einem Kern besteht und deren CDR-Teil den für diesen Kern zu substituierenden Ausdruck in algebraischer Präfixform enthält.

subeval1*(a:sub-list,e:pf):pf*	substituiere a in der algebraischen Präfixform e
subf*(f:sf,a:sub-list):sq*	substituiere a in der Standardform f
subsq*(q:sq,a:sub-list):sq*	substituiere a im Standardquotienten q

Tabelle 34 : Substitutionsoperatoren

Man beachte, daß sowohl `subf` als auch `subsq` einen Standard*quotienten* zurückgeben, da bei einer Substitution auch in eine Standardform das Entstehen von Nennern nicht auszuschließen ist.

5.3.6 Typkonversionsoperatoren

Die folgende Tabelle enthält eine Zusammenstellung der Konvertierungsoperatoren zwischen den verschiedenen Datentypen. Wenn die entsprechende Umwandlung nicht möglich ist, wird mit einem Fehler abgebrochen.

`!*A2F`	algebraische Präfixform in Standardform verwandeln
`!*A2K`	algebraische Präfixform in Kern verwandeln
`!*F2A`	Standardform in algebraische Präfixform verwandeln
`!*F2Q`	Standardform in Standardquotienten verwandeln
`!*K2F`	Kern in Standardform verwandeln
`!*K2Q`	Kern in Standardquotienten verwandeln
`!*Q2F`	Standardquotienten in Standardform verwandeln (sofern dessen Nenner gleich 1 ist)
`!*Q2K`	Standardquotienten in Kern verwandeln
`mk!*sq`	Standardquotienten in SQ-Form verwandeln
`prepf`	algebraische Präfixform oder SQ-Form in Standardform verwandeln
`prepsq`	algebraische Präfixform oder SQ-Form in Standardquotienten verwandeln

Tabelle 35 : Konvertierungsoperatoren

5.4 Spezielle Datentypen

5.4.1 Zahlbereiche

Außer den von RLISP direkt bereitgestellten Zahlbereichen können in Standardformen und Standardquotienten auch allgemeinere Konstrukte, *Domainelemente*, verwendet werden. Das sind Paare, deren CAR-Teil einen *Typidentifikator* enthält und deren CDR-Teil auf das Element in einer für den jeweiligen Typ spezifischen Syntax verweist.

Zahlbereich	tag	interne Struktur
algebraische Zahlen	`!:ar!:`	`<Standardform>`
Floatzahlen	`!:rd!:`	`ft:`*float*
ganze Zahlen	–	`<integer>`
ganze komplexe Zahlen	`!:gi!:`	`( real . imag )`
modulare Zahlen	`!:mod!:`	`<integer>`
komplexe Floatzahlen	`!:gr!:`	`( real:`*float* `. imag:`*float* *)*
rationale Zahlen	`!:rn!:`	`( z . n )`

Tabelle 36 : Zahlbereichstypen in RLISP

Hierbei ist `float` je nach der Genauigkeit entweder eine Float-Maschinenzahl oder aber ein Paar `( mant . exp )` aus Mantisse und Exponent, beide ganzzahlig.

Der Aufrufkonflikt arithmetischer Operationen, wo je nach dem entsprechenden Bereichstyp zu verschiedenen Prozeduren zu verzweigen ist, wird über die Eigenschaftenliste des Typidentifikators aufgelöst.

Die Elemente *zero* und *one* sind gleich für alle Bereiche und werden durch die Elemente *nil* und 1 dargestellt.

5.4.2 Matrizen

Matrizen sind in Form einer zweistufigen Liste mit einem vorangestellten Typidentifikator `'mat` dargestellt.

Beispiel :

```
symbolic$
l:='((1 2 3) (4 5 6));

        ((1 2 3) (4 5 6))

m:='mat . l;

        (MAT (1 2 3) (4 5 6))

mathprint m$

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

Kapitel 6

REDUCE : (DOS-)Installation und Service

6.1 REDUCE Umgebung

Auf allen Plattformen wird REDUCE in einem eigenen Verzeichnis installiert, das in der Umgebungsvariablen $REDUCE abgelegt werden kann. Dieses Verzeichnis enthält eine Reihe weiterer Unterverzeichnisse. Diese Verzeichnisstruktur ist nicht einheitlich für alle Plattformen. Die REDUCE Version 3.5 enthält die folgenden Verzeichnisse :

bin	Binärfiles, insbesondere `bpsl` und `reduce.img`
doc	Dokumentation, meist im LATEX-Format
fasl	präkompilierte Moduln
help	Hilfesystem
lib	von Nutzern erstellte Erweiterungsbibliotheken
log	verschiedene bei der Systeminstallation erzeugte Protokollfiles
plot	Graphikausgabe
psl	Portable Standard Lisp
src	Quelltexte der Moduln des Kernbereichs von REDUCE
util	verschiedene weitere Dienstprogramme
xmpl	interaktive Lektionen und Beispiele für den Einsatz verschiedener Pakete

Tabelle 37 : REDUCE DOS-Verzeichnisbaum

Die DOS-Distribution von REDUCE umfaßt Versionen für

"pures" DOS, eine auf dem ERGO DOS-Extender aufsetzende protected mode Version, die damit nicht der Restriktion auf die klassischen 640 kB unterworfen ist (und damit nur auf einem 80386 und besser lauffähig ist),

Windows 3.1, die den Windows DPMI-Extender benutzt und auch virtuellen Speicher verwendet, wenn der physisch vorhandene zu gering ist (was allerdings die Rechenzeiten erheblich erhöht),

OS/2 und

Windows NT

Diese vier Versionen unterscheiden sich im wesentlichen nur in der verwendeten ausführbaren PSL-Binärdatei. Sowohl das eingefrorene `reduce.img` File als auch die vorkomplilierten Moduln werden von allen vier Versionen gemeinsam verwendet, so daß man REDUCE auch auf einem Rechner mit mehreren Betriebssystemen einsetzen kann.

Die DOS-Version benötigt einen Hauptspeicher von mindestens 2.5 MB, für umfangreichere Anwendungen sind wenigstens 4 MB notwendig. Da sie im protected mode läuft, dürfen keine anderen Programme (`emm386` oder `himem` etc.) aktiv sein, die auf den Erweiterungsspeicher zugreifen. Für die Windows-Version gelten diese Restriktionen nicht, da sie über dar DPMI-Interface konsistenter mit der restlichen MS-Software kooperiert.

Ein REDUCE Aufruf kann außerdem die Größe des zu reservierenden Speichers angegeben werden. Je nach Plattform hat ein solcher Aufruf die Gestalt

```
$REDUCE\dos386\psll $REDUCE\dos386\reduce.img <size> <stacksize>

oder

$REDUCE\bpsl -f $REDUCE\reduce.img -td <size>
```

`<size>` ist die Größe des zu Prozeßbeginn alloziierten Speichers in Byte.

Literaturverzeichnis

[1] Computeralgebra in Deutschland. Bestandsaufnahme, Möglichkeiten, Perspektiven. Herausgegeben von der Fachgruppe Computeralgebra der GI, DMV, GAMM (1993).

[2] A.C. Hearn : REDUCE User's Manual, v. 3.5. The Rand Corporation, Santa Monica 1993.

[3] M. MacCallum, F. Wright : Algebraic Computing with REDUCE . Clarendon Press, Oxford 1991.

[4] J. Marti, Hearn, A. C., Griss, M. L. and Griss, C. : Standard LISP Report, SIGPLAN Notices, ACM, New York, 14, No. 10 (1979) 48-68.

(Vgl. auch `DOC`-Verzeichnis der REDUCE Distribution)

[5] H. Melenk : REDUCE User's Guide for Personal Computers. Konrad-Zuse-Zentrum Berlin 1993.

[6] H. Melenk : REDUCE Symbolic Mode Primer. Draft version. Konrad-Zuse-Zentrum Berlin 1993.

[7] R. Pavelle, M. Rothstein, J. Fitch : Computer algebra. *Scientific American* , Dec. 1981.

[8] G. Rayna : REDUCE – Software for Algebraic Computation. Springer Series Symbolic Computation. Springer, New York 1987.

[9] J. Ueberberg : Einführung in die Computeralgebra mit REDUCE . BI Wissenschaftsverlag Mannheim, 1992.

[10] W. Steeb, D. Lewien : Algorithms and Computing with REDUCE . BI Wissenschaftsverlag Mannheim, 1992.