Skript zum Kurs
Einführung in das symbolische Rechnen
Sommersemester 2003

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

18. Juli 2003

Inhaltsverzeichnis

Kapitel 0

Einleitung

0.1 Das Anliegen dieses Kurses

Nach dem Siegeszug von Taschenrechner und (in klassischen Programmiersprachen geschriebenen) Numerik-Paketen spielen heute Computeralgebra-Systeme (CAS) mit ihren Möglichkeiten, auch stärker formalisiertes mathematisches Wissen in algorithmisch aufbereiteter Form über eine einheitliche Schnittstelle zur Verfügung zu stellen, eine zunehmend wichtige Rolle. Solche Systeme, die im Gegensatz zu klassischen Anwendungen des Computers auch symbolische Kalküle beherrschen, werden zugleich in absehbarer Zeit wenigstens im naturwissenschaftlich-technischen Bereich den Kern umfassenderer Wissensrepräsentationssysteme bilden. Damit gehört der souveräne Umgang mit solchen Systemen zu einer der Grundfertigkeiten, die von Hochschul-Absolventen wenigstens des mathematisch-naturwissenschaftlichen Bereichs in Zukunft erwartet werden.

Wie für das Programmieren in klassischen Programmiersprachen ist hierbei ein ausgewogenes Verhältnis zwischen Erwerb von eigenen Erfahrungen und methodischer Systematisierung dieser Kenntnisse angezeigt. Hier gibt es viele Parallelen zum Einsatz des Taschenrechners im Schulunterricht. Obwohl Schüler bereits frühzeitig eigene Erfahrungen im Umgang mit diesem „Werkzeug geistiger Arbeit“, etwa im Fachunterricht, sammeln, wird auf dessen *systematische* Einführung ab Klasse 8 (sächsischer Lehrplan) nicht verzichtet. Schließlich gehört der qualifizierte Umgang mit dem Taschenrechner, insbesondere die Kenntnis seiner Eigenarten, Möglichkeiten und Grenzen, zu den elementaren Fertigkeiten, die jeder Schüler mit Verlassen der Schule beherrschen sollte. Dasselbe gilt in noch viel größerem Maße für die Fähigkeiten von Hochschulabsolventen im Umgang mit CAS, deren Möglichkeiten, Eigenarten und Grenzen ob der Komplexität dieses Instruments viel schwieriger auszuloten sind.

Ich gehe davon aus, dass die Kursteilnehmer in der überwiegenden Zahl bereits eigene Erfahrungen mit einem der großen CAS gesammelt haben, und werde ich mich deshalb auf eine systematische Darstellung der Möglichkeiten, Grenzen, Formen und Methoden des Einsatzes von Computern zur Ausführung symbolischer Rechnungen konzentrieren. Im Gegensatz zur einschlägigen Literatur soll dabei nicht eines der großen Systeme im Mittelpunkt stehen, sondern deren Gesamtheit Berücksichtigung finden. Eine zentrale Rolle werden die an unserer Universität weit verbreiteten Systeme MAPLE, MUPADund MATHEMATICA spielen.

Ähnlich wie sich übergreifende Aspekte von Programmiersprachen nur aus deren vergleichender Betrachtung erschließen, ist ein solches Herangehen besser geeignet, die grundlegenden Techniken und Begriffe, die mit der Anwendung des Computers für symbolische Rechnungen verbunden sind, abzuheben.

Ein solcher Zugang erscheint mir auch deshalb sowohl gerechtfertigt als auch wünschenswert, weil es mittlerweile für jedes der großen Systeme eine Fülle von einführender Literatur höchst unterschiedlicher Qualität und verschiedenen Anspruchsniveaus gibt. Mehr noch, die Entwicklerteams

der großen Systeme haben in den letzten Jahren viel Energie darauf verwendet, ihre Produkte auch von der äußeren Gestalt her nach modernen softwaretechnischen und -ergonomischen Gesichtspunkten aufzubereiten[1], so dass selbst ein ungeübter Nutzer in der Lage sein sollte, mit wenig Aufwand wenigstens die Grundfunktionalitäten des von ihm favorisierten Systems eigenständig zu erschließen. Bei Kenntnis grundlegender Techniken, die von all diesen Systemen verwendet werden, sollte diese Einarbeitung noch schneller gelingen.

Für den Nutzer von Computeralgebrasystemen wird es andererseits zunehmend schwieriger, die wirkliche Leistungsfähigkeit der Systeme hinter ihrer gleichermaßen glitzernden Fassade zu erkennen. Auch hierfür soll dieser Kurs ein gewisses Testmaterial zur Verfügung stellen, obwohl bei einem globalen Vergleich der Leistungsfähigkeit der verschiedenen Systeme durchaus Vorsicht geboten ist. Unterschiedliche Herangehensweisen im Design zusammen mit der jeweils spezifischen Entstehungsgeschichte führten dazu, dass die Leistungsfähigkeit unterschiedlicher Systeme in unterschiedlichen Bereichen sehr differiert und, mehr noch, sich verschiedene „mathematische Rigorosa“ wie etwa in einem Teil der Wester-Liste [22, ch. 3] oder in Bernardins Übersicht [22, ch. 7] enthalten, mit sehr unterschiedlichem Aufwand im Nachhinein integrieren lassen. Diese Feinheiten führen an einigen Stellen zu unterschiedlichem Verhalten der verschiedenen Systeme bis hin zu sehr unterschiedlichen Antworten. Wir werden uns dabei auf die Frage beschränken, wie konsistent die einzelnen Systeme ihre eigenen Konzepte verfolgen, da gerade Übersichten wie die zitierten die Systementwickler manchmal dazu verleiten, bisher nicht beachtete Problemstellungen als „quick hack“ und damit nur halbherzig in ihre Systeme zu integrieren.

Diese Konsistenzfrage steht als generelle Einsatzvoraussetzung für den Nutzer eines Computeralgebrasystems an zentraler Stelle. Leider ist sie nicht eindeutig zu beantworten, so lange *ein* Werkzeug *verschiedene* Nutzergruppen adressiert. Aber selbst die Einführung global einstellbarer „Nutzerprofile“, wie in [22, ch. 3] vorgeschlagen, würde das Problem kaum lösen. Es ist deshalb in diesem Gebiet noch wichtiger als beim Taschenrechnereinsatz, sich kritisch mit dem Sinn bzw. Unsinn gegebener Antworten auseinanderzusetzen und sich über Art und Umfang möglicher Rechnungen und Antworten bereits im Voraus in groben Zügen im Klaren zu sein.

Allerdings begegnet man diesem „Zauberlehrlingseffekt“, den Weizenbaum in seinem inzwischen klassischen Werk [20] in Bezug auf den Computereinsatz erstmals umfassend artikulierte, im Zusammenhang mit dem Einsatz moderner Technik immer wieder. Dieser als „Janusköpfigkeit“ bezeichnete Effekt, dass der unqualifizierte und insbesondere nicht genügend reflektierte Einsatz mächtiger technischer Mittel zu unkontrollierten, unkontrollierbaren und in ihrer Wirkung unbeherrschbaren Folgen führen kann, wissen wir nicht erst seit Tschernobyl. Daraus resultierende Fragen sind inzwischen Gegenstand eines eigenen Wissensgebiets, des „technology assessment“, geworden, dessen deutsche Übertragung „Technologiefolgenabschätzung“ die zu thematisierenden Inhalte nur unvollkommen wiedergibt. Bezogen auf den Computer kann die Beschäftigung mit Computeralgebra auch hier wertvolle Einsichten vermitteln und helfen, ein am Werkzeugcharakter orientiertes kritisches Verhältnis zum Computereinsatz gegenüber heute oft anzutreffender Fetischisierung (wieder) mehr in den Vordergrund zu rücken.

[1] Dies ist, nach dem Vorreiter MATHEMATICA, mit einer stärkeren Kommerzialisierung auch der anderen großen Systeme verbunden. Eine wichtige Erkenntnis scheint mir dabei zu sein, dass sich im Gegensatz zur unmittelbaren Forschung an Algorithmen hierfür keine öffentlichen Mittel allokieren lassen. Auch scheinen die subtilen Mechanismen, die zum Erfolg freier Softwareprojekte führen, hier nur langsam zu greifen, da das Verhältnis zwischen Größe der Nutzergemeinde und erforderlichem Programmieraufwand deutlich ungünstiger aussieht. Andererseits stehen die geforderten Lizenzpreise gerade der Studentenversionen in keinem Verhältnis zur Leistungsfähigkeit der Systeme. Mit diesen Mitteln kann man im Wesentlichen wohl nur Supportleistungen, nicht aber tieferliegende algorithmische Untersuchungen refinanzieren.

Die Diskussion der Chancen und Risiken des Zusamenfließens öffentlich geförderter wissenschaftlicher Arbeit und privatwirtschaftlicher Supportleistung an diesem Beispiel erscheint mir auch deshalb wünschenswert, weil ähnliche Entwicklungen in anderen Bereichen der Informatik (Stichwort: LINUX-Distributionen und freie Software) ebenfalls stattfinden.

0.2 Die Voraussetzungen

Symbolische Rechnungen sind das wohl grundlegendste methodische Handwerkszeug in der Mathematik und durchdringen alle ihrer Gebiete und Anwendungen. Die Spanne symbolischer Kalküle reicht dabei von allgemein bekannten Anwendungen wie Termvereinfachung, Faktorisierung, Differential- und Integralrechnung, über mächtige mathematische Kalküle mit weit verbreitetem Einsatzgebiet (etwa Analysis spezieller Funktionen, Differentialgleichungen) bis hin zu Kalkülen einzelner Fachgebiete (Gruppentheorie, Tensorrechnung, Differentialgeometrie), die vor allem für Spezialisten der entsprechenden Fachgebiete interessant sind.

Für jeden dieser Kalküle gilt, dass dessen qualifizierter Einsatz den mathematisch entsprechend qualifizierten Nutzer voraussetzt. Moderne CAS bündeln in diesem Sinne einen großen Teil des heute algorithmisch verfügbaren mathematischen Wissens und sind damit als Universalwerkzeuge geistiger Arbeit ein zentraler Baustein einer sich herausbildenden „Wissensgesellschaft".

Dabei stellt sich heraus, dass zur Implementierung und Nutzung der Vielfalt unterschiedlicher Kalküle ein kleines, wiederkehrendes programmiertechnisches Grundinstrumentarium in den verschiedensten Situationen variierend zum Einsatz kommt. Dies mag nicht überraschen, ist doch ein ähnliches Universalitätsprinzip programmiersprachlicher Mittel zur Beschreibung von Algorithmen und Datenstrukturen gut bekannt und kann sogar theoretisch begründet werden.

Für einen Einführungskurs ergibt sich aus diesen Überlegungen eine doppelte Schwierigkeit, da einerseits die zu demonstrierenden Prinzipien erst in nichttrivialen Anwendungen zu überzeugender Entfaltung kommen, andererseits solche Anwendungen ohne Kenntnis ihres Kontextes nur schwer nachvollziehbar sind. Dies verlangt, eine wohlüberlegte Auswahl zu treffen.

Im folgenden werden wir uns deshalb, neben einer ausführlichen Diskussion der Stellung des symbolischen Rechnens im Wissenschaftsgebäude, beschränken auf die Darstellung wichtiger Designaspekte eines CAS der zweiten Generation und die Diskussion theoretischer und praktischer Aspekte der Simplifikationsproblematik als besonderem Charakteristikum des symbolischen Rechnens. Im letzten Teil des Kurses werden wir schließlich am Beispiel der symbolischen Behandlung algebraischer Zahlen diese Konzepte in ihrem komplexen Zusammenspiel erleben, Anforderungen und Lösungen gegenüberstellen und dabei einige auf den ersten Blick merkwürdige Ansätze besser verstehen. Diesen Teil beschließt ein Überblick über die Algorithmen zur Integration rationaler Funktionen, in denen algebraische Zahlen auf natürliche Weise auftreten.

Nicht berühren werden wir das Feld der *Differentialgleichungen*, da die Schwierigkeiten der damit verbundenen Mathematik in keinem Verhältnis zu den dabei gewinnbaren neuen Einsichten in Zusammenhänge des symbolischen Rechnens steht. Gleichfalls ausgeklammert bleiben Fragen der graphischen Möglichkeiten der CAS, die ein wichtiges Mittel der Visualisierung wissenschaftlicher Zusammenhänge sind und für viele Nutzer den eigentlichen Reiz eines großen CAS darstellen, aber nicht zum engeren Gebiet des symbolischen Rechnens gehören. Dasselbe gilt für die mittlerweile ausgezeichneten Präsentationsmöglichkeiten der integrierten graphischen Oberflächen der meisten der betrachteten Systeme.

Kapitel 1

Einsatzvarianten von Computeralgebrasystemen

In diesem Kapitel wollen wir Computeralgebrasysteme in verschiedenen Situationen als Rechenhilfsmittel kennen lernen und dabei erste Besonderheiten eines solchen Systems gegenüber rein numerisch basierten Rechenhilfsmitteln heraus arbeiten. Die folgenden Rechnungen basieren auf MuPAD 2.5, hätten aber mit jedem der großen CAS in ähnlicher Weise ausgeführt werden können.

1.1 Computeralgebrasysteme als Taschenrechner für Zahlen

Mit einem CAS kann man natürlich zunächst alle Rechnungen ausführen, die ein klassischer Taschenrechner beherrscht.

Das umfasst neben den Grundrechenarten auch eine Reihe mathematischer Funktionen:

```
1.23+2.25;
```

$$3.48$$

```
12+23;
```

$$35$$

```
10!;
```

$$3628800$$

```
100!;
```

$$\begin{aligned}&93326215443944152681699238856266700490715968264381621468592963895217599993\\&22991560894146397615651828625369792082722375825118521091686400000000000000\\&0000000000\end{aligned}$$

Am letzten Beispiel erkennen wir eine erste Besonderheit: CAS rechnen im Gegensatz zu Taschenrechnern mit ganzen Zahlen mit *voller* Genauigkeit. Die in ihnen implementierte *Langzahlarithmetik* erlaubt es, die entsprechenden Umformungen *exakt* auszuführen. Dies wird auch bei der Behandlung rationaler Zahlen

```
1/2+2/3;
```

$$7/6$$

```
sum(1/i,i=1..50);
```

$$13943237577224054960759/3099044504245996706400$$

```
float(%);
```

$$4.499205338$$

sowie irrationaler Zahlen

```
sqrt(2);
```

$$\sqrt{2}$$

```
PI;
```

$$\pi$$

deutlich. Diese werden nicht durch numerische Näherungswerte ersetzt, sondern bleiben als *symbolische Ausdrücke* in einer Form stehen, die uns aus einem streng mathematischen Kalkül wohlbekannt ist. Im Gegensatz zu numerischen Approximationen, bei denen sich die Zahl `3.1415926535` qualitativ kaum von der Zahl `3.1426968052` unterscheidet (letzteres ist $\frac{20}{9}\cdot\sqrt{2}$, auf 10 Stellen nach dem Komma genau ausgerechnet), verbirgt sich hinter einem solchen Symbol eine wohldefinierte *mathematische Semantik*. So ist etwa

> $\sqrt{2}$ diejenige (eindeutig bestimmte) positive reelle Zahl, deren Quadrat gleich 2 ist.

Ein CAS verfügt über Mittel, diese Semantik (bis zu einem gewissen Grade) darzustellen. So „weiss“ MuPAD einiges über die Zahl π

```
sin(PI);
```

$$0$$

```
sin(PI/4);
```

$$\frac{\sqrt{2}}{2}$$

und sogar

```
sin(PI/5);
```

$$\frac{\sqrt{2}\sqrt{-\sqrt{5}+5}}{4}$$

Ein Programm, das nur einen (noch so guten) Näherungswert von π kennt, kann die Information $\sin(\pi) = 0$ prinzipiell nicht *exakt* ableiten:

```
DIGITS:=20: p:=float(PI); sin(p); delete DIGITS:
```

$$3.1415926535897932384$$

$$1.0097419586828951109e-28$$

Ohne hier auf Details der inneren Darstellung einzugehen, halten wir fest, dass CAS symbolischen Ausdrücken *Eigenschaften* zuordnen, die deren mathematischen Gehalt widerspiegeln.

Wie eben gesehen, kann eine dieser Eigenschaften insbesondere darin bestehen, dass dem CAS auch ein Verfahren zur Bestimmung eines *numerischen Näherungswerts* bekannt ist. Dieser kann mit einer beliebig vorgegebenen Präzision berechnet werden, die softwaremäßig auf der Basis der Langzahlarithmetik operiert. Damit ist die Numerik zwar oft deutlich langsamer als die auf Hardware-Operationen zurückgeführte Numerik klassischer Programmiersprachen, erlaubt aber genauere und insbesondere adaptive Approximationen.

Ähnlich wie auf dem Taschenrechner stehen auch wichtige mathematische Funktionen und Operationen zur Verfügung, allerdings in einem wesentlich größeren Umfang als dort. So wissen CAS, wie man von ganzen Zahlen den größten gemeinsamen Teiler, die Primfaktorzerlegung oder die Teiler bestimmt, die Primzahleigenschaft testet, nächstgelegene Primzahlen ermittelt und vieles mehr.

```
gcd(2^30-1,3^20-1);
```

$$11$$

```
ifactor(12!);
```

$$2^{10} \cdot 3^5 \cdot 5^2 \cdot 7 \cdot 11$$

```
isprime(12!-1);
```

$$TRUE$$

1.2 Computeralgebrasysteme als Taschenrechner für Formeln und symbolische Ausdrücke

Die wichtigste Besonderheit eines Computeralgebrasystems gegenüber Taschenrechner und klassischen Programmiersprachen ist ihre

Fähigkeit zur Manipulation symbolischer Ausdrücke.

Dies wollen wir zunächst an symbolischen Ausdrücken demonstrieren, die gewöhnliche Variablen enthalten, also Literale wie x, y, z ohne weitergehende mathematische Semantik. Aus solchen Symbolen kann man mit Hilfe der 4 Grundrechenarten *rationale Funktionen* zusammenstellen. Betrachten wir dazu einige Beispiele:

```
u:= a/((a-b)*(a-c)) + b/((b-c)*(b-a)) + c/((c-a)*(c-b));
```

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

Es ergibt sich die Frage, ob man diesen Ausdruck weiter vereinfachen kann. Am besten wäre es, wenn man einen gemeinsamen Zähler und Nenner bildet, welche zueinander teilerfremd sind, und diese in ihrer expandierten Form darstellt. Das ermöglicht die Funktion `normal`.

```
normal(u);
```

$$0$$

Das Ergebnis mag verblüffen; jedenfalls sieht man das dem ursprünglichen Ausdruck nicht ohne weiteres an.

Eine solche normalisierte Darstellung ist nicht immer zweckmäßig, weshalb diese Umformung nicht automatisch vorgenommen wurde. So erhalten wir etwa

```
u:=(x^15-1)/(x-1);
```

$$\frac{x^{15}-1}{x-1}$$

```
normal(u);
```

$$x+x^2+x^3+x^4+x^5+x^6+x^7+x^8+x^9+x^{10}+x^{11}+x^{12}+x^{13}+x^{14}+1$$

Betrachten wir allgemein Ausdrücke der Form

$$u_n := \frac{a^n}{(a-b)\,(a-c)} + \frac{b^n}{(b-c)\,(b-a)} + \frac{c^n}{(c-a)\,(c-b)}$$

und untersuchen, wie sie sich unter Normalformbildung verhalten. Wir verwenden dazu die Funktion `subs`, die lokal eine Variable durch einen anderen Ausdruck, in diesem Fall eine Zahl, ersetzt.

```
u:= a^n/((a-b)*(a-c)) + b^n/((b-c)*(b-a)) + c^n/((c-a)*(c-b)):

normal(subs(u,n=2));
```

$$1$$

```
normal(subs(u,n=3));
```

$$a+b+c$$

```
normal(subs(u,n=4));
```

$$ab+ac+bc+a^2+b^2+c^2$$

```
normal(subs(u,n=5));
```

$$abc+a^3+b^3+c^3+ab^2+a^2b+ac^2+a^2c+bc^2+b^2c$$

Wir sehen, dass sich in jedem der betrachteten Fälle die rationale Funktion u_n zu einem Polynom vereinfacht.

Tragen die in die Formeln eingehenden Symbole weitere semantische Information, so sind auch komplexere Vereinfachungen möglich. So werden etwa Ausdrücke, die die imaginäre Einheit I enthalten, wie erwartet vereinfacht:

```
(1+I)^n $ n=1..10;
```

$$1+I, 2I, -2+2I, -4, -4-4I, -8I, 8-8I, 16, 16+16I, 32I$$

```
(3+I)/(2-I);
```

$$1+I$$

Oftmals müssen solche Umformungen durch entsprechende Funktionsaufrufe gezielt angestoßen werden.

```
(sqrt(2)+sqrt(3))^n $ n=2..5;
```

$$\left(\sqrt{2}+\sqrt{3}\right)^2, \left(\sqrt{2}+\sqrt{3}\right)^3, \left(\sqrt{2}+\sqrt{3}\right)^4, \left(\sqrt{2}+\sqrt{3}\right)^5$$

```
expand((sqrt(2)+sqrt(3))^n) $n=2..5;
```

$$2\sqrt{2}\sqrt{3}+5, 11\sqrt{2}+9\sqrt{3}, 20\sqrt{2}\sqrt{3}+49, 109\sqrt{2}+89\sqrt{3}$$

Manchmal ist es nicht einfach, das System zu „überreden“, genau das zu tun, was man will. So liefern weder `expand` noch `normal` oder `simplify` eine Form von $(\sqrt{2}+\sqrt{3})^{-1}$ mit rationalem Nenner. Erst die offensichtlich aufwändige Funktion (Laufzeit 1.4 s.)

```
radsimp(1/(sqrt(2)+sqrt(3)));
```

$$\sqrt{3}-\sqrt{2}$$

liefert das erwartete Ergebnis, das man auch schnell im Kopf ausrechnen kann. Der Grund liegt darin, dass es aus Effizienzgründen nicht klug ist, Nenner rational zu machen. Wir kommen auf diese Frage später zurück.

1.3 CAS als Problemlösungs-Umgebungen

Wir haben in obigen Beispielen bereits in bescheidenem Umfang programmiersprachliche Mittel eingesetzt, um unsere speziellen Wünsche zu formulieren.

> Das Vorhandensein einer voll ausgebauten Programmiersprache, mit der man den Interpreter des jeweiligen CAS gut steuern kann, ist ein weiteres Charakteristikum der betrachteten Systeme (nur Derive macht hier bisher eine Ausnahme und orientiert sich stärker an einer Drag-and-Drop-Philosophie). Dabei werden alle gängigen Sprachkonstrukte einer imperativen Programmiersprache unterstützt und noch um einige Spezifika erweitert, die aus der Natur des symbolischen Rechnens folgen und über die weiter unter zu sprechen sein wird.

Mit diesem Instrumentarium und dem eingebauten mathematischen Wissen werden CAS so zu einer vollwertigen Problemlösungs-Umgebung, in der man neue Fragestellungen und Vermutungen (mathematischer Natur) ausgiebig testen und untersuchen kann. Selbst für zahlentheoretische Fragestellungen reichen die Möglichkeiten dabei weit über die des Taschenrechners hinaus.

Betrachten wir etwa Aufgaben der Art:

Bestimmen Sie die letzte Ziffer der Zahl 2^{100}*,*

wie sie in Schülerarbeitsgemeinschaften vor Einführung von CAS zum Kennenlernen des Rechnens mit Resten gern gestellt wurden. Die Originalaufgabe ist für ein CAS gegenstandslos, da man die ganze Zahl leicht ausrechnen kann.

```
2^100;
```

$$1267650600228229401496703205376$$

Erst im Bereich von Exponenten in der Größenordnung 1000000 wird der Verbrauch von Rechenzeit ernsthaft spürbar und die dann über 300000-stelligen Zahlen zunehmend unübersichtlich.

Wirkliche Probleme bekommt der Nutzer von MuPAD, wenn er die dem Computer „angemessene“ Frage nach letzten Ziffern der Zahl $2^{10^{10}}$ stellt. Zunächst ist zu beachten, dass MuPAD `2^10^10` als 2^{100} berechnet, der Operator $\wedge$ also linksassoziativ statt wie sonst üblich rechtsassoziativ wirkt[1]. Bei der korrekten Eingabe `2^(10^10)` gibt MuPAD nach endlicher Zeit auf mit der Information

[1] Zum Vergleich: Maple lässt solche Ausdrücke gar nicht zu, Mathematica folgt der allgemeinen – und mit Blick auf die Potenzgesetze auch sinnvollen – Konvention).

```
Error: Overflow/underflow in arithmetical operation
```

Der ursprüngliche Sinn der Aufgabe bestand darin, zu erkennen, dass man zur Berechnung der letzten Ziffer nicht die gesamte Potenz, sondern nur deren Rest (*mod* 10) berechnen muss und dass Potenzreste 2^k (*mod* 10) eine periodische Folge bilden. Mit unserem CAS kann man den Rechner für eine wesentlich weitergehende Untersuchung derselben Problematik einsetzen. Da dieses in der Lage ist, die ermüdende Berechnung der Potenzreste zu übernehmen[2], können wir nach Regelmäßigkeiten für die Potenzreste für beliebige Moduln fragen.

```
2^k mod 10 $ k=1..20;
```

$$2, 4, 8, 6, 2, 4, 8, 6, 2, 4, 8, 6, 2, 4, 8, 6, 2, 4, 8, 6$$

```
2^k mod 3 $ k=1..20;
```

$$2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1$$

```
2^k mod 11 $ k=1..20;
```

$$2, 4, 8, 5, 10, 9, 7, 3, 6, 1, 2, 4, 8, 5, 10, 9, 7, 3, 6, 1$$

```
2^k mod 17 $ k=1..20;
```

$$2, 4, 8, 16, 15, 13, 9, 1, 2, 4, 8, 16, 15, 13, 9, 1, 2, 4, 8, 16$$

Diese wenigen Kommandos liefern eine Fülle von Material, das uns schnell verschiedene Regelmäßigkeiten vermuten lässt, die man dann gezielter experimentell untersuchen kann. So taucht für primen Modul in der Folge stets eine 1 auf, wonach sich die Potenzreste offensichtlich wiederholen. Geht man dieser Eigenschaft auf den Grund, so erkennt man die Bedeutung primer Restklassen. Auch die Länge der Folge zwischen dem Auftreten der 1 folgt gewissen Gesetzmäßigkeiten, die ihre Verallgemeinerung in elementaren Aussagen über die Ordnung von Elementen in der Gruppentheorie finden.

Auch die modifizierte Aufgabe zur Bestimmung letzter Ziffern von $2^{10^{10}}$ können wir nun lösen. Der erste Versuch

```
2^(10^10) mod 10000;
```

schlägt allerdings fehl: Die Auswertungsregeln der Informatik führen dazu, dass *vor* Auswertung der mod-Funktion die inneren Argumente ausgewertet werden, d.h. zunächst der Versuch unternommen wird, die Potenz vollständig auszurechnen. Wir brauchen statt dessen eine spezielle Potenzfunktion, die bereits in den Zwischenrechnungen die Reste reduziert und damit Zwischenergebnisse überschaubarer Größe produziert. MuPAD verfolgt einen objektorientierten Ansatz, der funktionale Polymorphie (hier der Potenzfunktion) an Hand der Argumenttypen auflöst. Die korrekte Potenzfunktion wird also ausgewählt, wenn bereits die Zahl 2 als Restklasse erzeugt wird:

```
R:=Dom::IntegerMod(10^20);
R(2)^(10^10);
```

$$\begin{aligned}&\text{Dom :: IntegerMod}(10000000000000000000)\\ &4637454968178710937 6 \bmod 10000000000000000000\end{aligned}$$

In Maple erreichen wir denselben Effekt über den *inerten Operator* `&^` (auf solche inerten Funktionen wird später noch einzugehen sein),

[2] Notwendige Nebenbemerkung für den Einsatz in der Schule: Nachdem dieser Gegenstand genügend geübt worden ist.

```
2&^(10^10) mod 10^20;
```

4637454968178710937 6

in MATHEMATICA müssen wir die spezielle Funktion `PowerMod` verwenden:

```
PowerMod[2,10^10,10^20]
```

4637454968178710937 6

Wir wollen die auch aus didaktischen Gesichtspunkten interessante Möglichkeit, CAS als Problemlösungs- und Experimentierumgebung einzusetzen, zur Erforschung von Eigenschaften ganzer Zahlen an einem weiteren Beispiel verdeutlichen:

Eine Zahl n heisst *perfekt*, wenn sie mit der Summe ihrer echten Teiler übereinstimmt, d.h. die Summe *aller* ihrer Teiler gerade $2\,n$ ergibt. Die Untersuchung solcher Zahlen kann man bis in die Antike zurückverfolgen. Bereits in der Pythagoräischen Schule im 6. Jh. vor Christi werden solche Zahlen betrachtet. EUKLID kannte eine Formel, nach der man alle gerade perfekte Zahlen findet, die erstmals von EULER exakt bewiesen wurde.

Wir wollen nun ebenfalls versuchen, uns einen Überblick über die perfekten Zahlen zu verschaffen. Das MUPAD-Paket `numlib` enthält die Funktion `sigma(n)`, mit der wir die Teilersumme der natürlichen Zahl n berechnen können. Mit der folgenden Schleife verschaffen wir uns erst einmal einen Überblick über die perfekten Zahlen bis 800:

```
export(numlib):
for i from 2 to 800 do
        if 2*i=sigma(i) then print(i) end_if
end_for:
```

6
28
496

Betrachten wir die gefundenen Zahlen näher:

$$6 = 2 \cdot 3, \quad 28 = 4 \cdot 7, \quad 496 = 16 \cdot 31.$$

Alle diese Zahlen haben die Gestalt $2^{k-1}(2^k - 1)$. Wir wollen deshalb versuchen, perfekte Zahlen dieser Gestalt zu finden:

```
for k from 2 to 40 do n:=2^(k-1)*(2^k-1):
        if 2*n=sigma(n) then print(k,n,yes)
        else print(k,n,no)
        end_if
end_for:
```

k	n	perfekt ?
2	6	yes
3	28	yes
4	120	no
5	496	yes
6	2016	no
7	8128	yes
8	32640	no
9	130816	no
10	523776	no
11	2096128	no
12	8386560	no
13	33550336	yes
14	134209536	no
15	536854528	no
16	2147450880	no
17	8589869056	yes
18	34359607296	no
19	137438691328	yes
20	549755289600	no
...		

Die Ausgabe ist als Tabelle zusammengestellt und wegen ihrer Länge gekürzt. Sie bietet umfangreiches experimentelles Material, auf dessen Basis sich qualifizierte Vermutungen über bestehende Gesetzmäßigkeiten aufstellen lassen. Es scheint insbesondere so, als ob perfekte Zahlen nur für prime k auftreten. Jedoch liefert nicht jede Primzahl k eine perfekte Zahl n.

Derartige Beobachtung kann man nun versuchen, mathematisch zu untermauern, was in diesem Fall nur einfache kombinatorische Überlegungen erfordert: Die Teiler der Zahl $n = p_1^{a_1} \cdot \ldots \cdot p_m^{a_m}$ haben offensichtlich genau die Gestalt $t = p_1^{b_1} \cdot \ldots \cdot p_m^{b_m}$ mit $0 \leq b_i \leq a_i$. Ihre Summe beträgt

$$\sigma(n) = (1 + p_1 + \ldots + p_1^{a_1}) \cdot \ldots \cdot (1 + p_m + \ldots + p_m^{a_m}).$$

In der Tat, multipliziert man den Ausdruck aus, so hat man aus jeder Klammer einen Summanden zu nehmen und zu einem Produkt zusammenzufügen. Alle möglichen Auswahlen ergeben genau die beschriebenen Teiler von n.

Die Teilersumme $\sigma(n)$ ergibt sich also unmittelbar aus der Kenntnis der Primfaktorzerlegung der Zahl n. Ist insbesondere $n = a \cdot b$ eine zusammengesetzte Zahl mit zueinander teilerfremden Faktoren a, b , so gilt offensichtlich $\sigma(n) = \sigma(a) \cdot \sigma(b)$.

Wenden wir das auf die gerade perfekte Zahl $n = 2^{k-1}b$ ($k > 1, b$ ungerade) an, so gilt wegen $\sigma(2^{k-1}) = 1 + 2 + 2^2 \ldots + 2^{k-1} = 2^k - 1$

$$2n = 2^k\, b = \sigma(n) = (2^k - 1)\sigma(b).$$

Also ist $2^k - 1$ ein Teiler von $2^k\, b$ und als ungerade Zahl damit auch von b. Es gilt folglich $b = (2^k - 1)c$ und somit $\sigma(b) = 2^k\, c = b + c$. Da b und c Teiler von b sind und $\sigma(b)$ *alle* Teiler von b aufsummiert, hat b nur die Teiler b und $c = 1$, muss also eine Primzahl sein. Da auch umgekehrt jede solche Zahl eine perfekte Zahl ist haben wir damit folgenden Satz bewiesen:

Satz 1 *Jede gerade perfekte Zahl hat die Gestalt* $n = 2^{k-1}(2^k - 1)$*, wobei* $P := 2^k - 1$ *eine Primzahl sein muss.*

Ungerade perfekte Zahlen sind bis heute nicht bekannt, ein Beweis, dass es solche Zahlen nicht gibt, aber auch nicht gefunden worden.

1.4 Computeralgebrasysteme als Expertensysteme

Neben den bisher betrachteten Rechenfertigkeiten und der Programmierbarkeit, die ein CAS auszeichnen, spielt das

in ihnen gespeicherte mathematische Wissen

für Anwender die entscheidende Rolle. Dieses deckt, wenigstens insofern wir uns auf die dort vermittelten algorithmischen Fertigkeiten beschränken, weit mehr als den Stoff der gymnasialen Oberstufe ab und umfasst die wichtigsten symbolischen Kalküle der Analysis (Differenzieren und Integrieren, Taylorreihen, Grenzwertberechnung, Trigonometrie, Rechnen mit speziellen Funktionen), der Algebra (Matrizen- und Vektorrechnung, Lösen von linearen und polynomialen Gleichungssystemen, Rechnen in Restklassenbereichen), der Kombinatorik (Summenformeln, Lösen von Rekursionsgleichungen, kombinatorische Zahlenfolgen) und vieler anderer Gebiete der Mathematik.

Für viele Kalküle aus mathematischen Teildisziplinen, aber zunehmend auch solche aus verwandten Bereichen anderer Natur- und Ingenieurwissenschaften, ja selbst der Finanzmathematik, gibt es darüber hinaus spezielle Pakete, auf die man bei Bedarf zugreifen kann. Dieses sprunghaft wachsende Expertenwissen ist der eigentliche Kern eines CAS als Expertensystem. Man kann mit gutem Gewissen behaupten, dass heute bereits große Teile der algorithmischen Mathematik in dieser Form verfügbar sind und auch algorithmisches Wissen aus anderen Wissensgebieten zunehmend erschlossen wird. In dieser Richtung spielt das System MATHEMATICA seit Jahren eine Vorreiterrolle.

In dem Zusammenhang ist die im deutschen Sprachraum verbreitete Bezeichnung „Computeralgebra“ irreführend, denn es geht durchaus nicht nur um algebraische Algorithmen, sondern auch um solche aus der Analysis und anderen Gebieten der Mathematik. Entscheidend ist allein der *algebraische Charakter* der entsprechenden Manipulationen als endliche Kette von Umformungen symbolischer Ausdrücke. Und genau so geht ja etwa der Kalkül der Differentialrechnung vor: die konkrete Berechnung von Ableitungen elementarer Funktionen erfolgt nicht nach der (auf dem Grenzwertbegriff aufbauenden) Definition, sondern wird durch geschicktes Kombinieren verschiedener *Regeln*, wie der Ketten-, der Produkt- und der Quotientenregel, auf entsprechende Grundableitungen zurückgeführt.

Hier einige Beispielrechnungen mit MUPAD:

```
diff(x^n,x);
```

$$\frac{n x^n}{x}$$

```
diff(x^2*sin(x)*ln(x+a),x$2); # Zweite Ableitung #
```

$$2\sin(x)\ln(a+x)+4x\cos(x)\ln(a+x)+\frac{4x\sin(x)}{(a+x)}-x^2\sin(x)\ln(a+x)+\frac{2x^2\cos(x)}{(a+x)}-\frac{x^2\sin(x)}{(a+x)^2}$$

Beim Integrieren kommen darüber hinaus auch prozedurale Verfahren zum Einsatz, wie etwa die Partialbruchzerlegung, die ihrerseits die Faktorzerlegung des Nennerpolynoms und das Lösen von (linearen) Gleichungssystemen verwendet, die aus einem Ansatz mit unbestimmten Koeffizienten gewonnen werden. Auf diese Weise, und das ist ein weiteres wichtiges Moment des Einsatzes von CAS,

spielen algorithmische Fertigkeiten aus sehr verschiedenen Bereichen der Mathematik nahtlos miteinander zusammen.

```
p:=(x^3+x^2+x-2)/(x^4+x^2+1);
```

$$\frac{x^3+x^2+x-2}{x^4+x^2+1}$$

```
int(p,x);
```

$$\ln\left(x^2-x+1\right)-\frac{\ln\left(\left(x+1/2\right)^2+3/4\right)}{2}-\frac{\sqrt{3}\arctan\left(\frac{2\sqrt{3}\left(x+1/2\right)}{3}\right)}{3}$$

```
diff(%,x);
```

$$\frac{(2x+-1)}{(x^2-x+1)}-\frac{2}{3\left(\frac{4(x+1/2)^2}{3}+1\right)}-\frac{(2x+1)}{2\left(\left(x+1/2\right)^2+3/4\right)}$$

```
normal(%-p);
```

$$0$$

```
int(1/(x^4-1),x);
```

$$\frac{\ln\left(x-1\right)}{4}-\frac{\arctan\left(x\right)}{2}-\frac{\ln\left(x+1\right)}{4}$$

Expertenwissen wird zum Lösen von verschiedenen Problemklassen benötigt, wobei der Nutzer in den seltensten Fällen wissen (und wissen wollen) wird, welche konkreten Algorithmen im jeweiligen Fall genau anzuwenden sind. Deshalb bündeln die CAS die algorithmischen Fähigkeiten zum Auflösen gewisser Sachverhalte in einem oder mehreren

`solve`-Operatoren,

die den Typ eines Problems erkennen und geeignete Lösungsalgorithmen aufrufen.

So bündelt der `solve`-Operator von MuPAD Wissen über Algorithmen zum Auffinden der Nullstellen univariater Polynome, von linearen und polynomialen Gleichungssystemen und selbst Differentialgleichungen können damit gelöst werden.

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

$$\left\{-1/2i\sqrt{3}-1/2,1/2i\sqrt{3}-1/2\right\}$$

```
solve({x+y=2,x-y=1},{x,y});
```

$$\left\{\left[y=1/2,x=3/2\right]\right\}$$

```
solve({x^2+y=2,3*x-y^2=2},{x,y});
```

$$\left\{\left[x=1,y=1\right],\left[x=2,y=-2\right],\left[x=-\frac{i}{2}\sqrt{3}-3/2,y=1/2-\frac{3\,i}{2}\sqrt{3}\right],\left[x=\frac{i}{2}\sqrt{3}-3/2,y=\frac{3\,i}{2}\sqrt{3}+1/2\right]\right\}$$

Selbst für kompliziertere Gleichungen, die trigonometrische Funktionen enthalten, werden Lösungen in mittlerweile schöner Form angegeben:

```
s:=solve(sin(x)=1/2);
```

$$x \in \{\frac{\pi}{6} + 2\pi \mathrm{X17}; \mathrm{X17} \in \mathbf{Z}\} \cup \{\frac{5\pi}{6} + 2\pi \mathrm{X19}; \mathrm{X19} \in \mathbf{Z}\}$$

Genau so erfolgt die vollständige Angabe der Lösungsmenge in einer exakten mathematischen Darstellung. Allerdings sieht das Ergebnis netter aus als es ist; es bedarf schon einiger Kenntnisse der Interna von MuPAD, um s weiter zu verarbeiten. Wir wollen daraus alle Lösungen im Intervall $-10 < x < 10$ extrahieren, also die angegebene Vereinigungsmenge mit dem Intervall $[-10, 10]$ schneiden. In MuPAD 2.0 (Windows-Version) verbarg sich hinter der schönen Darstellung von s noch die logische Verknüpfung $(x \in M_1) \vee (x \in M_2)$, aus der zunächst die Mengenverknüpfung $M_1 \cup M_2$ zu extrahieren war. In MuPAD 2.5 ist diese Vereinigungsmenge $M = M_1 \cup M_2$ bereits als zweites Argument in s (intern dargestellt als _in(x, M)) enthalten, kann also mit `op(s,2)` extrahiert und dann mit dem relevanten Intervall geschnitten werden[3]:

```
op(s,2) intersect Dom::Interval(-10,10);
```

$$\left\{\frac{13\,\pi}{6}, \frac{\pi}{6}, -\frac{11\,\pi}{6}, \frac{17\,\pi}{6}, \frac{5\,\pi}{6}, -\frac{7\,\pi}{6}, -\frac{19\,\pi}{6}\right\}$$

Hätten wir die Aufgabe in der Form

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

angeschrieben, denn dann wäre das Ergebnis gleich in der Mengenform ausgegeben worden.

$$\{\frac{\pi}{6} + 2\pi \mathrm{X17}; \mathrm{X17} \in \mathbf{Z}\} \cup \{\frac{5\pi}{6} + 2\pi \mathrm{X19}; \mathrm{X19} \in \mathbf{Z}\}$$

Allerdings geht dabei die Information verloren, dass die Werte mit der Variablen x zu tun haben.

Diese Information wird von folgender Variante des `solve`-Operators weitergegeben, dessen Ausgabe mit der im ersten Fall übereinstimmt.

```
s:=solve(sin(x)=1/2,{x});
```

Maples Lösung derselben Aufgabe lautet

```
solve(sin(x)=1/2);
```

$$1/6\,\pi$$

Wenn wir uns erinnern, dass die Lösungsmenge trigonometrischer Gleichungen periodisch ist, also mit x auch $x + 2\pi$ die Gleichung erfüllt, so haben wir schon die Hälfte der tatsächlichen Lösungsmenge gefunden. Warum kommt Maple nicht selbst auf diese Idee? Auch hier hilft erst ein Blick in die (Online-)Dokumentation weiter; setzt man eine spezielle Systemvariable richtig, so erhalten wir das erwartete Ergebnis, allerdings in einer recht verklausulierten Form:

```
_EnvAllSolutions:=true;
```

$$\texttt{\_EnvAllSolutions} := \mathit{true}$$

```
solve(sin(x)=1/2);
```

$$1/6\,\pi + 2/3\,\pi\,\texttt{\_B1\~{}} + 2\,\pi\,\texttt{\_Z1\~{}}$$

Hier sind `_B1` und `_Z1` Hilfsvariablen mit den Wertebereichen `_B1` $\in \{0, 1\}$ (B wie boolean) und `_Z1` $\in \mathbf{Z}$.

Mathematica antwortet so:

[3] Die Reihenfolge der Elemente in der Ausgabe auf dem Bildschirm und der Printausgabe stimmen nicht überein – aber das Ergebnis ist ja auch eine *Menge*.

```
In[1]:= Solve[Sin[x]==1/2,x]

Solve::ifun: Inverse functions are being used by Solve, so some
     solutions may not be found.

                Pi
Out[1]= {{x -> --}}
                6
```

Es gibt keine Möglichkeit, MATHEMATICA zu bewegen, die Lösungsschar in parametrisierter Form auszugeben. Im Handbuch wird dies wie folgt begründet ([24, p. 794]):

> *Obwohl sich alle Lösungen dieser speziellen Gleichung einfach parametrisieren lassen, führen die meisten derartigen Gleichungen auf schwierig darzustellende Lösungsmengen. Betrachtet man Systeme trigonometrischer Gleichungen, so sind für eine Parameterdarstellung diophantische Gleichungssysteme zu lösen, was im allgemeinen Fall als algorithmisch nicht lösbar bekannt ist.*

Wir stoßen mit unserer einfachen Frage also unvermutet an prinzipielle Grenzen der Mathematik, die im betrachteten Beispiel allerdings noch nicht erreicht sind.

Oft haben wir nicht nur mit der Frage der Vollständigkeit der angegebenen Lösungsmenge zu kämpfen, sondern auch mit unterschiedlichen Darstellungen ein und des selben Ergebnisses, die sich aus unterschiedlichen Lösungswegen ergeben. Betrachten wir etwa die Aufgabe

```
solve(sin(x)+cos(x)=1/2,x);
```

Eine mögliche Umformung, die uns die vollständige Lösungsmenge liefert, wäre

$$\sin(x)+\cos(x)=\sin(x)+\sin(\frac{\pi}{2}-x)=2\sin(\frac{\pi}{4})\cos(x-\frac{\pi}{4}),$$

womit die Gleichung zu $\cos(x-\frac{\pi}{4})=\frac{1}{2\sqrt{2}}$ umgeformt wurde.

Als Ergebnis erhalten wir (wegen $\cos(x)=u \;\Rightarrow\; x=\pm\arccos(u)+2\,k\,\pi$)

$$L=\left\{\frac{\pi}{4}+\arccos(\frac{1}{2\sqrt{2}})+2k\pi,\ \frac{\pi}{4}-\arccos(\frac{1}{2\sqrt{2}})+2k\pi \;\middle|\; k\in\mathbf{Z}\right\}.$$

MUPADs Antwort lautet

```
s:=solve(sin(x)+cos(x)=1/2,x);
```

$$\{2\pi \mathrm{X110}+2\arctan\left(2/3-\tfrac{\sqrt{7}}{3}\right);\mathrm{X110}\in\mathbf{Z}\}\cup\{2\pi \mathrm{X113}+2\arctan\left(\tfrac{\sqrt{7}}{3}+2/3\right);\mathrm{X113}\in\mathbf{Z}\}$$

während MAPLE folgende Antwort liefert:

```
_EnvAllSolutions:=true:
s:=[solve(sin(x)+cos(x)=1/2,x)];
```

$$\left[\arctan\left(\tfrac{1/4-1/4\sqrt{7}}{1/4+1/4\sqrt{7}}\right)+2\,\pi\,\_Z3,\arctan\left(\tfrac{1/4+1/4\sqrt{7}}{1/4-1/4\sqrt{7}}\right)+\pi+2\,\pi\,\_Z3\right]$$

MATHEMATICA schließlich:

```
s:=Solve[Sin[x]+Cos[x]==1/2,{x}]

Solve::ifun: Inverse functions are being used by Solve, so some
     solutions may not be found.
```

$$\{\{x \to \arccos(\tfrac{1-\sqrt{7}}{4})\}, \{x \to -\arccos(\tfrac{1+\sqrt{7}}{4})\}\}$$

Die Lösungen werden in sehr unterschiedlichen Formen präsentiert, die sich auch nicht durch einfache (im Sinne von „offensichtliche“) Umformungen ineinander überführen lassen. Versuchen wir wenigstens die näherungsweise Auswertung einiger Elemente beider Lösungsmengen:

```
float(s intersect Dom::Interval(-10,10)); # MuPAD #
```

$$\{-6.707216347, -4.288357941, -0.4240310395, 1.994827366, 5.859154268, 8.278012674\}$$

```
seq(op(subs(_Z3=i,evalf(s))), i=-1..1); # Maple #
```

$$-6.707216348, -4.288357941, -0.4240310395, 1.994827367, 5.859154268, 8.278012675$$

```
s//N (* Mathematica *)
```

$$\{\{x \to 1.99483\}, \{x \to -0.424031\}\}$$

An dieser Stelle wird bereits deutlich, dass die Komplexität eines CAS es oftmals notwendig macht, Ergebnisse in einem gegenüber dem Taschenrechnereinsatz vollkommen neuen Umfang nach- und umzuinterpretieren, um sie an die eigenen Erwartungen an deren Gestalt anzupassen.

1.5 Erweiterbarkeit von Computeralgebrasystemen

Ein weiterer Aspekt, der für den Einsatz eines CAS als Expertensystem wichtig ist, besteht in der

Möglichkeit der Erweiterung seiner mathematischen Fähigkeiten.

Die zur Verfügung stehende Programmiersprache kann nicht nur zur Ablaufsteuerung des Interpreters verwendet werden, sondern auch (in unterschiedlichem Umfang), um eigene Funktionen und ganze Pakete zu definieren.

Betrachten wir etwa die Vereinfachungen, die sich bei der Untersuchung des rationalen Ausdrucks u_n ergeben haben. Die dabei entstandenen Polynome, wie übrigens u_n auch, ändern sich bei Vertauschungen der Variablen nicht. Solche Ausdrücke nennt man deshalb *symmetrisch*. Insbesondere spielen symmetrische Polynome in vielen Anwendungen eine wichtige Rolle. Nehmen wir an, dass wir solche symmetrischen Polynome untersuchen wollen, diese Funktionalität aber vom System nicht gegeben wird.

Dazu erst einige Definitionen. Sei $X = (x_1, \ldots, x_n)$ eine endliche Menge von Variablen. Die bekanntesten symmetrischen Polynome sind die *elementarsymmetrischen Polynome* $e_k(X)$, die alle verschiedenen Terme aus k paarweise verschiedenen Faktoren x_i aufsummieren, die *Potenzsummen* $p_k(X) = \sum_i x_i^k$ und die *vollen symmetrischen Polynome* $h_k(X)$, die *alle* Terme in den gegebenen Variablen vom Grad k aufsummieren. So gilt etwa für $n = 4$

$$\begin{aligned} e_2 &= x_1x_2 + x_1x_3 + x_1x_4 + x_2x_3 + x_2x_4 + x_3x_4 \\ p_2 &= {x_1}^2 + {x_2}^2 + {x_3}^2 + {x_4}^2 \\ h_2 &= x_1x_2 + x_1x_3 + x_1x_4 + x_2x_3 + x_2x_4 + x_3x_4 + {x_1}^2 + {x_2}^2 + {x_3}^2 + {x_4}^2 \end{aligned}$$

Wir wollen das System MuPAD so erweitern, dass durch Funktionen `e(d,vars)`, `p(d,vars)` und `h(d,vars)` zu einer natürlichen Zahl d und einer Liste *vars* von Variablen diese Polynome erzeugt werden können. Statt der angegebenen Definition wollen wir dazu die rekursiven Relationen

$$e(d, (x_1, \ldots, x_n)) = e(d, (x_2, \ldots, x_n)) + x_1 \cdot e(d-1, (x_2, \ldots, x_n))$$

und

$$h(d, (x_1, \ldots, x_n)) = h(d, (x_2, \ldots, x_n)) + x_1 \cdot h(d-1, (x_1, \ldots, x_n))$$

verwenden. Von der Richtigkeit dieser Formeln überzeugt man sich schnell, wenn man die in der Summe auftretenden Terme in zwei Gruppen einteilt, wobei die erste Gruppe x_1 nicht enthält, die Teilsumme also gerade das entsprechende symmetrische Polynom in $(x_2, \ldots, x_n)$ ist. In der zweiten Gruppe kann man x_1 ausklammern.

Die entsprechenden Funktionsdefinitionen in MuPAD lauten (ohne Typprüfungen der Eingabeparameter)

```
e:=proc(d,vars)
  local u;
begin
  if nops(vars)<d then 0
  elif d=0 then 1
  else u:=[op(vars,2..nops(vars))];
      expand(e(d,u)+op(vars,1)*e(d-1,u))
  end_if
end_proc;

h:=proc(d,vars)
  local u;
begin
  if d=0 then 1
  elif nops(vars)=1 then op(vars,1)^d
  else u:=[op(vars,2..nops(vars))];
      expand(h(d,u)+op(vars,1)*h(d-1,vars))
  end_if
end_proc;

p:=proc(d,vars)
begin
  _plus(op(map(vars,x->x^d,x)))
end_proc;
```

Wir sehen an diesem kleinen Beispiel bereits, dass die komplexen Datenstrukturen, die in symbolischen Rechnungen auf natürliche Weise auftreten, den Einsatz verschiedener Programmierparadigmen und -praktiken ermöglichen.

Die ersten beiden Blöcke ergänzen die jeweilige Rekursionsrelation durch geeignete Abbruchbedingungen zu einer korrekten Funktionsdefinition für e_d und h_d. Im dritten Block werden intensiv verschiedene Operationen auf Listen in einem funktionalen Programmierstil verwendet. Für

```
vars:=[x,y,z];
```

erhalten wir damit

```
h(2,vars);
```

$$xy + xz + yz + x^2 + y^2 + z^2$$

```
e(3,vars);
```

$$xyz$$

```
h(3,vars);
```

$$xyz + x^3 + y^3 + z^3 + xy^2 + x^2y + xz^2 + x^2z + yz^2 + y^2z$$

Ein Vergleich mit unseren weiter oben vorgenommenen Vereinfachungen zeigt, dass die rationale Funktion u_d offensichtlich gerade mit dem vollen symmetrischen Polynom h_{d-2} zusammenfällt.

Kapitel 2

Die Stellung des symbolischen Rechnens im Wissenschaftsgebäude

In diesem Kapitel wollen wir die Konsequenzen für Wissenschaft und Gesellschaft näher beleuchten, die sich aus der Möglichkeit ergeben, verschiedene Kalküle der Wissenschaft in Computerprogramme zu gießen und sie auf diese Weise einem weiten Anwenderkreis als Black- oder Grey-Box-Verfahren zur Verfügung zu stellen.

Vorab sei bemerkt, dass im deutschsprachigen Raum die Begriffe „symbolisches Rechnen“ und „Computeralgebra“ gewöhnlich alternativ zur Kennzeichnung des Fachgebiets, um das es in diesem Kurs geht, verwendet werden. In diesem Kapitel werde ich den Begriff „symbolisches Rechnen“ (wie auch bei der Bezeichnung des Kurses insgesamt) verwenden, wenn ich stärker auf die Potenzen des Fachgebiets fokussiere, den Begriff „Computeralgebra“ dagegen, wenn ich den aktuellen Entwicklungsstand des Fachgebiets stärker ins Auge fasse.

2.1 Zur Genese von Wissenschaft im Industriezeitalter

Ein Blick in die Geschichte lehrt uns, dass es den heute geläufigen Wissenschaftsbegriff mit seinen mannigfachen Verzweigungen und Verästelungen noch gar nicht so lange gibt. Bis hinein ins Mittelalter wurde Wissenschaft ganzheitlich und mit dem Anspruch betrieben, die Welt in ihrer gesamten Komplexität zu begreifen. Für Goethes Faust galt es noch, Philosophie, Medizin, Juristerei und Theologie, die vier Zweige eines klassischen wissenschaftlichen Studiums jener Zeit, nicht alternativ, sondern gemeinsam und in ihrer gegenseitigen Wechselbeziehung zu studieren. Zugleich war das Wissenschaftlerdasein elitär geprägt und „das Privileg meist wohlhabender, oft adliger Privatgelehrter“ ([1, S. 278]). Im Alltag spielten wissenschaftliche Kenntnisse eine absolut untergeordnete Rolle, ja selbst (aus heutiger Sicht elementare) Grundfertigkeiten wie Lesen, Schreiben und Rechnen waren kaum verbreitet.

Das ändert sich grundlegend mit dem Aufbruch ins Industriezeitalter. Neben einem paradigmatischen Bruch in der Wissenschaft selbst (siehe ebenda, S. 279) beginnt Wissenschaft auch im Alltag eine wichtigere Rolle einzunehmen; abzulesen etwa in der Einrichtung von Volksschulen, welche die Fertigkeiten des Lesens, Schreibens und Rechnens verbreiten.

Ursache für diese veränderte Stellung von Wissenschaft sind zweifelsohne die gewachsenen Anforderungen, die ein industriell organisierter Arbeitsprozess sowohl an die beteiligten Akteure als auch an die geistige Durchdringung der Prozesse selbst stellt. Wissenschaftliche Bemühungen werden nach [1] nunmehr stärker auf die Fragen des „Wie?“ und „Wodurch?“, also auf funktiona-

le und kausale Erklärungen der Phänomene ausgerichtet. Ein solches Verständnis ermöglicht erst das „Eingreifenkönnen und Beherrschen natürlicher Prozesse und Dinge“ (ebenda, S.278). *Wissenschaftliche Rationalität* wird damit zum beherrschenden Wissenstypus, wenigstens im Bereich der Natur- und Technikwissenschaften, denen wir uns im Weiteren ausschließlich zuwenden werden.

Ein solcher Rationalitätsbegriff prägt denn auch das heutige Selbstverständnis der einzelnen Naturwissenschaften (Physik, Chemie, Biologie, ...) als Fachwissenschaften: sie haben als Ziel, in der Natur ablaufende Prozesse adäquat zu beschreiben und damit Modellvorstellungen zu entwickeln, auf deren Basis man Vorhersagen über diese Prozesse treffen oder sie sogar bewusst ausnutzen oder beeinflussen kann. Letzteres ist mit leicht anderer Schwerpunktsetzung auch Gegenstand der Technikwissenschaften.

Die wissenschaftliche Strenge, die für eine solche Rationalität an den Tag zu legen ist, unterliegt fachübergreifenden Standards. Die Existenz derartiger Standards hat ihre Ursache nur zum Teil im gemeinsamen Ursprung der Einzelwissenschaften. Eine wesentlich wichtigere Quelle liegt in der gemeinsamen Methodologie und dem dabei verwendeten erkenntnistheoretischen Instrumentarium, das in der folgenden Erkenntnisspirale angeordnet werden kann:

- aus einer Fülle von experimentell gewonnenem Datenmaterial werden *Regelmäßigkeiten* herausgefiltert und in *Hypothesen* mit dem bisherigen Kenntnisstand verbunden (hierfür wird Intuition benötigt);
- diese werden durch wissenschaftlich strenge Beweise zu *Gesetzmäßigkeiten* verdichtet (hierbei wächst der Abstaktionsgrad);
- Bündel von zusammengehörigen Gesetzmäßigkeiten werden im Zuge weiterer experimenteller Verfikation zu neuen *Theorien* zusammengefasst.

Für die praktische Anwendung dieser neuen Theorien ist schließlich deren

- Aufbereitung in einem handhabbaren *Kalkül* ausschlaggebend[1], der zugleich
- die Basis für die Gewinnung neuen Datenmaterials auf der nächst höheren Abstraktionsebene bildet.

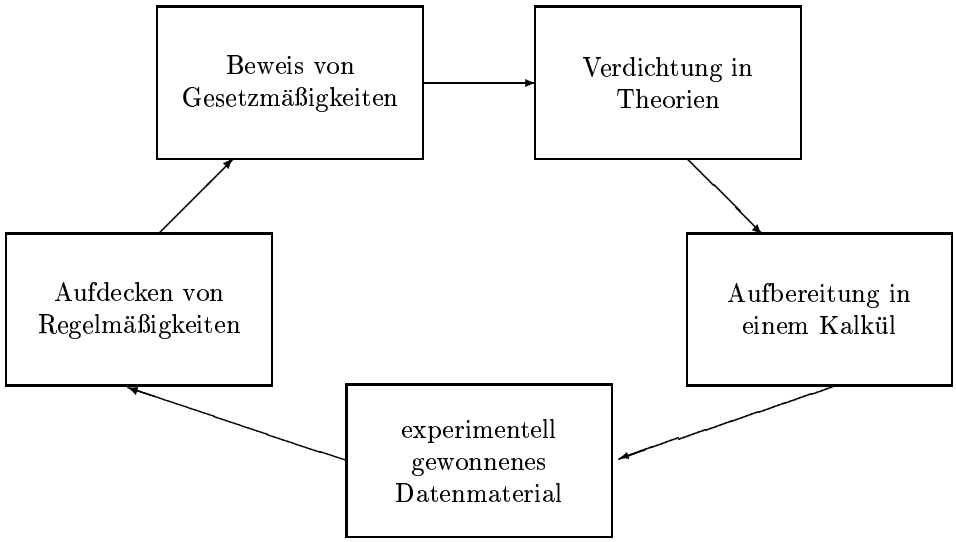
Diese Spirale wurde und wird im Erkenntnisprozess ständig durchlaufen, wobei der Gang durch jede aktuelle Windung alle vorherigen subsumiert und voraussetzt. Auch die Ontogenese von Wissenschaft, die Heranführung junger Nachwuchskräfte an die vorderste Front ihres Faches, folgt einer solchen Spirale zunehmender Abstraktion.

Aus der Sicht des symbolischen Rechnens ist dabei die Rolle von Kalkülen besonders interessant. Während in den Phasen des Datensammelns und Entdeckens von Regelmäßigkeiten die aktive Beherrschung des jeweiligen Kalküls notwendig ist, wobei der Computer eine wichtige Hilfe sein kann, rückt in der Phase des Formulierens von Gesetzmäßigkeiten und Theorien die Fähigkeit, *über* den aktuellen Kalkül zu räsonieren und sich damit auf ein höheres Abstraktionsniveau zu begeben, in den Mittelpunkt. Hierbei ist der Computer nur von sehr beschränktem Nutzen, wenigstens seine *speziellen* Kalkülfähigkeiten betreffend.

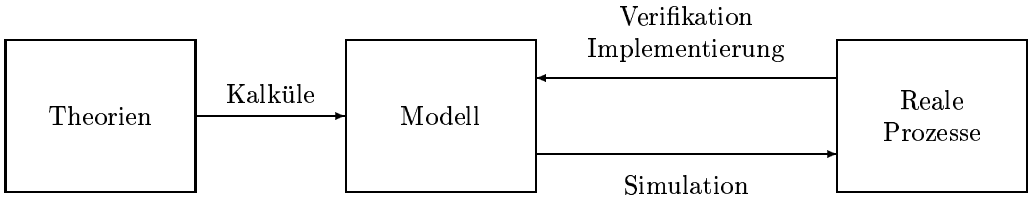
Eine solche in Richtung zunehmender Abstraktion weisende Erkenntnisspirale ist typisch für die „reinen“ Wissenschaften. Um Wissenschaften im Zuge zunehmender Industrialisierung produktiv werden zu lassen, spielt die Anwendbarkeit und Anwendung theoretischen Wissens auf die gesellschaftliche Praxis eine ebenso wichtige Rolle. Diese Domäne der „angewandten“ und Technik- oder Ingenieurwissenschaften folgt einem anderen erkenntnistheoretischen Paradigma:

- Reale Prozesse werden mit Hilfe eines geeigneten Kalküls *simuliert*.

[1] Buchberger [2, S. 808] spricht in diesem Zusammenhang von der „Trivialisierung“ einer Problemklasse (der symbolischen Mathematik).



Die Erkenntnisspirale der „reinen“ Wissenschaften



Paradigma der „angewandten“ Wissenschaften

- Die Simulation wird auf dem Hintergrund der verwendeten Theorie durch Analyse zu einem *Modell* verdichtet.
- Das Modell wird experimentell überprüft (und gegebenenfalls weiter verfeinert)
- Die gewonnenen Erkenntnisse werden in die Praxis implementiert.

In diesem Kreislauf spielen fertige Theorien und konkrete, bereits entwickelte Kalküle (nicht nur der Mathematik) und damit die Kalkülfähigkeiten des Computers eine zentrale Rolle.

Übergreifende Gesetzmäßigkeiten dieser Erkenntnisprozesse sind Gegenstand von *Querschnittswissenschaften*, von denen hier vor allem Philosophie, Mathematik und inzwischen auch die Informatik zu nennen sind.

Während die Philosophie die Denk- und Abstraktionsprozesse in ihrer Allgemeinheit zum Gegenstand hat, befasst sich die Mathematik mit übergreifenden Gesetzmäßigkeiten, welche beim Quantifizieren von Phänomenen auftreten. Quelle und Target dieser Bemühungen sind die ent-

sprechenden logischen Strukturen der Einzelwissenschaften, die oft erst durch die Anstrengungen der Mathematik eine streng deduktiven Ansprüchen genügende Konsistenz erhalten.

Die Mathematik leistet so einen unverzichtbaren und eigenständigen Beitrag für die methodische Fundierung der Einzelwissenschaften, ohne welchen letztere nur wenig über ein empirisches Verständnis ihres Gegenstands hinauskommen würden. Mathematik und mathematischen Methoden kommt damit besonders in der Phase der Hypothesen- und Theoriebildung, aber auch bei der Modellierung und Analyse realer Prozesse, ein wichtiger Platz für die Leistungsfähigkeit und argumentative Tiefe einzelwissenschaftlicher Erkenntnisprozesse zu. Sie ist außerdem die Grundlage einzelwissenschaftlicher Kalküle, egal, ob diese Quantenphysik, Elektronik, Statik oder Reaktionskinetik heißen. Mathematik ist in diesem Sinne die „lingua franca“ der Wissenschaft, was MARX zu der Bemerkung veranlasste, dass „sich eine Wissenschaft erst dann als entwickelt betrachten könne, wenn sie dahin gelangt sei, sich der Mathematik zu bedienen“.

Im Gegensatz zu spezielleren Kenntnissen aus einzelnen Bereichen der Natur- oder Ingenieurwissenschaften sind mathematische Kenntnisse und Fertigkeiten damit in unserer technisierten Welt nicht nur in breiterem Umfang notwendig, sondern werden auch an verschiedenen Stellen des (Berufs-)Lebens selbst bei Facharbeitern oder vergleichbaren Qualifikationen schlichtweg vorausgesetzt. Eine gewisse „mathematische Kultur“, die über einfache Rechenfertigkeiten hinausgeht, ist damit heute für eine qualifizierte Teilhabe am sozialen Leben unumgänglich.

Jedoch ist nicht nur der Einzelne auf solche Kenntnisse angewiesen, sondern auch die Gesellschaft als Ganzes. Denn erst eine solche „Kultur des Denkens“ sichert die Fähigkeit, innerhalb der Gesellschaft auf einem Niveau zu kommunizieren, wie es für die Beherrschung der sozialen Prozesse notwendig ist, die sich aus der immer komplexeren technologischen Basis ergeben. Unter diesem Blickwinkel mag es nicht weiter verwundern, dass der Teil des durch die Mathematik entwickelten methodischen und begrifflichen Rüstzeugs, der inzwischen in die Allgemeinbildung Einzug gehalten hat, stetig wächst.

Obwohl es immer wieder Diskussionen über die Angemessenheit solcher Elemente im Schulunterricht gibt, zeigt sich im Lichte der TIMMS-Studien der letzten Jahre, die die mathematischen Fertigkeiten von Schülern in verschiedenen Ländern vergleicht, dass die allgemeine mathematische Kultur, die die Schule in Deutschland derzeit vermittelt, eher als mittelmäßig einzustufen ist.

Mit der allgegenwärtigen Verfügbarkeit leistungsfähiger Rechentechnik wird diese „Verwissenschaftlichung“ gesellschaftlicher Zusammenhänge auf eine qualitativ neue Stufe gehoben. Viele, auch umfangreichere Kalküle können nun mechanisiert oder sogar automatisiert werden und stehen damit für einen breiteren Einsatz zur Verfügung, womit sich zugleich die Reichweite wissenschaftlicher Gedankenführung für einen weiten Kreis von Anwendungen deutlich erhöht. Neben Pflege, Weiterentwicklung und Vermittlung entsprechender *Denk-Kalküle*, dem traditionellen Gegenstand *mathematischer* Bildung, tritt damit eine weitere Querschnittswissenschaft, die die Erstellung, Pflege, Nutzungsunterweisung und Einbettung für solche technikbasierte Hilfsmittel geistiger Arbeit, kurz, eine sich neu herausbildende *technologische Seite des Denkens*, zum Gegenstand hat. Ein solches Verständnis von Informatik[2] lässt Raum für eine

weitergehende Symbiose von Kalkül und Technologie als Gegenstand eines Faches zwischen Mathematik und Informatik,

dem Grabmeier in [6] den provisorischen Namen „Computermathematik“ gegeben hat. Das symbolische Rechnen ist ein wesentlicher Teil dieses Gebiets.

[2] Gängige Definitionen des Gegenstands der Informatik fokussieren stärker auf eine einzelwissenschaftliche Betrachtung, etwa als „Wissenschaft von der systematischen Verarbeitung von Informationen, besonders der automatischen Verarbeitung mit Digitalrechnern“ ([4]).

2.2 Die Erkenntnisspirale der Mathematik

Auch die Mathematik entwickelt sich in ähnlichen Erkenntnisspiralen wie oben für die Naturwissenschaften beschrieben. Das findet seinen Niederschlag nicht zuletzt im Schulcurriculum, wo der Abstraktionsgrad der vermittelten mathematischen Begriffe, Denk- und Arbeitsweisen von Schuljahr zu Schuljahr wächst. Das Abstraktionsniveau klettert dabei von der Beherrschung der Grundrechenarten im kleinen und großen Einmaleins über Variablenansätze in Textaufgaben und Lösungsverfahren für einfache lineare und nichtlineare Probleme bis hin zu einem ersten Kontakt mit Begriffen der Differential- und Integralrechnung. Dieses Curriculum zeichnet damit in groben Zügen die wichtigsten Abstraktionsschritte nach, welche die Mathematik als Wissenschaft in den letzten Jahrhunderten selbst gegangen ist und die der Logik ihres inneren Aufbaus entsprechen.

Eine solche Zunahme des Abstraktionsniveaus fällt allerdings nicht, wie in der Regel im Schulunterricht, vom Himmel, sondern ist das Resultat eben dieser Erkenntnisspirale, welche die Mathematik wie jede andere Wissenschaft durchläuft: In einer Fülle von *Experimenten* werden *Regelmäßigkeiten* aufgedeckt, die im Weiteren zu *Gesetzmäßigkeiten* verdichtet werden und schließlich in weiter ausgebauter und systematisierter Form als *Kalkül* für neue Experimente zur Verfügung stehen.

Die Durchführung von Experimenten setzt die Beherrschung des aktuellen Kalküls voraus, die Aufdeckung von Regelmäßigkeiten im so gewonnenen Erfahrungsmaterial zusätzlich eine geschulte Beobachtungsgabe, gepaart mit einem guten Schuss Intuition. Will man die erkannten Regelmäßigkeiten allerdings zu Gesetzmäßigkeiten verdichten, reichen Fertigkeiten im Umgang mit dem aktuellen Kalkül nicht mehr aus, sondern man muss in der Lage sein, *über diesen Kalkül* selbst zu reflektieren. Eine solche Zunahme des Abstraktionsniveaus ist oft damit verbunden, dass man einzelne Elemente des aktuellen Kalküls in Sinneinheiten zusammenfasst und mit diesen als Ganzes operiert, wofür sich symbolische Methoden anbieten.

Erkannte Gesetzmäßigkeiten geben einen Ursache-Wirkungs-Zusammenhang wieder, der im Weiteren natürlich dahingehend produktiv werden soll, dass man gezielt Wirkungen durch Erzeugung der entsprechenden Ursachen hervorrufen möchte. Hierfür ist es notwendig, die gefundenen Gesetzmäßigkeiten und die Argumente zu deren Fundierung zu einem neuen Kalkül zu verdichten, womit schließlich ein voller Zyklus der Erkenntnisspirale durchlaufen ist.

Wir wollen die hier beschriebene Erkenntnisspirale der Mathematik einige Windungen weit verfolgen. Grundstock aller mathematischer Fertigkeiten sind zuerst einmal *Rechenfertigkeiten*, wie sie seinerzeit von Rechenmeistern wie Adam Ries verbreitet wurden und heute Gegenstand des Grundschul-Unterrichts sind. Entsprechende Fertigkeiten, deren weite Verbreitung im 18. und 19. Jahrhundert die Herausbildung unserer modernen Gesellschaft erst ermöglichte, verlieren heute scheinbar an Bedeutung, denn jeder Taschenrechner und Computer beherrscht dieses Gebiet ebenfalls und sogar zuverlässiger und schneller als der Mensch.

So kann man etwa, statt dies per Hand zu tun, die Fertigkeiten eines Taschenrechners im Umgang mit Dezimalbrüchen zum Berechnen von Nullstellen eines Polynoms, etwa mit dem Newtonverfahren, nutzen. Wir wollen dies tun, um eine größere Menge von Daten zu erzeugen, in der wir eine Beziehung zwischen den Nullstellen und den Koeffizienten eines Polynoms aufspüren wollen. Je nach Komfort, den uns der jeweilige Taschenrechner bietet, müssen wir für die Berechnung jeder solchen Nullstelle mehr oder weniger Aufwand treiben. Auf moderneren Taschenrechnern reicht oft ein einziger Knopfdruck pro Nullstelle. Auf einem Computer könnten wir sogar noch komfortabler die ganze Problemstellung in ein Programm gießen und uns die entsprechenden Ergebnisse gleich als Tabelle ausgeben lassen. Abgesehen von diesem (für praktische Zwecke nicht unwesentlichen) Komfort sind die dafür notwendigen Rechenfertigkeiten allerdings keine anderen als bei einer Handrechnung. Wir wollen diese Rechnungen deshalb mit Maple simulieren, wobei wir die Koeffizienten des entsprechenden Polynoms zweiten Grades jeweils mit der Summe und dem Produkt seiner Nullstellen vergleichen:

```
for a from 2.0 to 4.0 do for b from -12.0 to -9.0 do
```

```
    u:=solve(x^2+a*x+b=0,x);
    print(u,a,b,u[1]+u[2],u[1]*u[2])
end_for end_for ;
```

Nullstellen	a	b	Summe	Produkt
[-4.605551275, 2.605551275]	2.0	-12.0	-2.000000000	-12.00000000
[-4.464101615, 2.464101615]	2.0	-11.0	-2.000000000	-11.00000000
[-4.316624790, 2.316624790]	2.0	-10.0	-2.000000000	-9.99999999
[-4.162277660, 2.162277660]	2.0	-9.0	-2.000000000	-8.99999999
[-5.274917218, 2.274917218]	3.0	-12.0	-3.000000000	-12.00000000
[-5.140054945, 2.140054945]	3.0	-11.0	-3.000000000	-11.00000000
[-5., 2.]	3.0	-10.0	-3.000000000	-10.00000000
[-4.854101966, 1.854101966]	3.0	-9.0	-3.000000000	-8.99999999
[-6., 2.]	4.0	-12.0	-4.000000000	-12.00000000
[-5.872983346, 1.872983346]	4.0	-11.0	-4.000000000	-11.00000000
[-5.741657387, 1.741657387]	4.0	-10.0	-4.000000000	-10.00000000
[-5.605551276, 1.605551276]	4.0	-9.0	-4.000000000	-9.00000000

Wir erkennen unschwer die bekannte Relation, dass für die Nullstellen x_1 und x_2 gerade $a = -(x_1 + x_2)$ und $b = x_1\, x_2$ gilt. Eine solche Liste von experimentellen Ergebnissen gibt eine gewisse Evidenz, dass ein solcher Zusammenhang besteht, liefert aber, so lang sie auch sein mag, noch keinen Beweis im streng deduktiven Sinn. Diesen wird man für den Fall eines Polynoms zweiten Grades erst erbringen können, wenn man in der Lage ist, die Nullstellen der *allgemeinen Gleichung 2. Grades*

$$x^2 + a\,x + b = 0$$

symbolisch durch die Koeffizienten a, b darzustellen, also die allgemeine Lösungsformel

$$x_1 = -1/2\,a + 1/2\,\sqrt{a^2 - 4\,b}, \quad x_2 = -1/2\,a - 1/2\,\sqrt{a^2 - 4\,b}$$

kennt. Eine solche Kenntnis geht natürlich weit über die Fähigkeit hinaus, die jeweiligen Nullstellen mit dem Newtonverfahren zu berechnen. Sie sind bereits Teil eines anderen Kalküls, mit dem man *über* die Nullstellen univariater Polynome sprechen kann.

Entsprechend einfach ist es auch, aus diesen Formeln einen *Beweis* für den behaupteten Zusammenhang herzuleiten, wenn man diesen neuen Kalkül, die *Variablenrechnung*, beherrscht. Mehr noch sind die entsprechenden Rechnungen mit einem Computerprogramm, das Formelmanipulationen beherrscht, *automatisch* ausführbar, wie in unserem Beispiel mit Maple:

```
delete a,b; u:=solve(x^2+a*x+b=0,x);
```

$$\left\{-\frac{a}{2} - \frac{\sqrt{a^2-4\,b}}{2}, -\frac{a}{2} + \frac{\sqrt{a^2-4\,b}}{2}\right\}$$

```
u[1]+u[2];
```

$$-a$$

```
expand(u[1]*u[2]);
```

$$b$$

Dieselben Untersuchungen für ein Polynom dritten Grades lassen einen ähnlichen Zusammenhang vermuten. Die entsprechenden numerischen Rechnungen sind wieder mit Maple simuliert, diesmal aber auf drei Stellen gekürzt:

```
DIGITS:=3;
for a from 2.0 to 3.0 do
   for b from -12.0 to -11.0 do
   for c from 3.0 to 5.0 do
     u:=solve(x^3+a*x^2+b*x+c=0,x);
     print(u, a,b,c, u[1]+u[2]+u[3], u[1]*u[2]*u[3] )
end_for end_for end_for ;
```

Nullstellen	a	b	c	Summe	Produkt
[-4.69, .263, 2.43]	2.0	-12.0	3.0	-2.00	-2.99
[-4.72, .359, 2.36]	2.0	-12.0	4.0	-2.00	-3.99
[-4.75, .460, 2.29]	2.0	-12.0	5.0	-2.00	-5.02
[-4.56, .290, 2.27]	2.0	-11.0	3.0	-2.00	-3.00
[-4.59, .398, 2.19]	2.0	-11.0	4.0	-2.00	-4.01
[-4.62, .515, 2.10]	2.0	-11.0	5.0	-2.01	-5.00
[-5.35, .270, 2.08]	3.0	-12.0	3.0	-3.00	-3.00
[2., -5.37, .373]	3.0	-12.0	4.0	-3.00	-3.99
[-5.40, .485, 1.91]	3.0	-12.0	5.0	-3.01	-5.00
[-5.22, .300, 1.92]	3.0	-11.0	3.0	-3.00	-3.01
[-5.24, .418, 1.83]	3.0	-11.0	4.0	-2.99	-4.01
[-5.27, .554, 1.71]	3.0	-11.0	5.0	-3.01	-4.99

Hier ist der Zusammenhang zur expliziten Lösungsformel zum Beweis der angetroffenen Regelmäßigkeit bereits schwieriger auszunutzen. Für Polynome höheren Grades, wo es solche Lösungsformeln (wenigstens in einer aus dem Schulunterricht gewohnten Form) gar nicht gibt, muss man schließlich tieferliegende Zusammenhänge zwischen den Nullstellen univariater Polynome und deren Koeffizienten ausnutzen, die sich nur in einer Sprache formulieren lassen, die *über Variablenrechnung* räsoniert: Ein Polynom

$$f := x^n + a_1\, x^{n-1} + a_2\, x^{n-2} + \ldots + a_{n-1}\, x + a_n$$

hat stets n komplexe Nullstellen $x_1, x_2, \ldots, x_n$, wenn man diese mit der entsprechenden Vielfachheit zählt, und es gilt

$$f = (x - x_1) \cdot (x - x_2) \cdot \ldots \cdot (x - x_n).$$

Nun ist es leicht, die Vietaschen Formeln allgemein herzuleiten. Man muss einfach diese Produktdarstellung ausmultiplizieren und die Koeffizienten mit denen in der Ausgangsdarstellung vergleichen.

Satz 2 *(Vietascher Wurzelsatz)*
Zwischen den Koeffizienten des Polynoms

$$f := x^n + a_1\, x^{n-1} + a_2\, x^{n-2} + \ldots + a_{n-1}\, x + a_n$$

und seinen Nullstellen $x_1, x_2, \ldots, x_n$ besteht folgender Zusammenhang:

$$a_i = (-1)^i\, e_i(x_1, x_2, \ldots, x_n), \quad i = 1, 2, \ldots, n,$$

wobei e_i das im letzten Kapitel eingeführte i-te elementarsymmetrische Polynom ist.

Evidenz für diesen Sachverhalt kann man sich wiederum mit einem Computerprogramm wie Maple verschaffen, das diesen Variablenkalkül, hier insbesondere das *Rechnen mit Polynomen in mehreren Variablen*, beherrscht:

```
for n from 2 to 5 do print(collect(expand(_mult(x-x.i $ i=1..n)),x)) end_for;
```

$$x^2 + (-x_2 - x_1)\,x + x_1\,x_2$$

$$x^3 + (-x_1 - x_3 - x_2)\,x^2 + (x_1\,x_3 + x_1\,x_2 + x_2\,x_3)\,x - x_1\,x_2\,x_3$$

$$\begin{aligned} x^4 &+ (-x_2 - x_1 - x_4 - x_3)\,x^3 + (x_1\,x_2 + x_2\,x_4 + x_2\,x_3 + x_1\,x_4 + x_1\,x_3 + x_3\,x_4)\,x^2 \\ &+ (-x_1\,x_2\,x_4 - x_1\,x_2\,x_3 - x_2\,x_3\,x_4 - x_1\,x_3\,x_4)\,x + x_1\,x_2\,x_3\,x_4 \end{aligned}$$

$$\begin{aligned} x^5 &+ (-x_1 - x_2 - x_5 - x_3 - x_4)\,x^4 \\ &+ (x_2\,x_5 + x_2\,x_4 + x_3\,x_4 + x_2\,x_3 + x_1\,x_2 + x_1\,x_4 + x_3\,x_5 + x_1\,x_5 + x_1\,x_3 + x_4\,x_5)\,x^3 \\ &+ \big(-x_1\,x_2\,x_3 - x_3\,x_4\,x_5 - x_2\,x_4\,x_5 - x_2\,x_3\,x_5 - x_1\,x_4\,x_5 - x_1\,x_3\,x_5 \\ &\qquad - x_1\,x_2\,x_5 - x_1\,x_3\,x_4 - x_1\,x_2\,x_4 - x_2\,x_3\,x_4\big)x^2 \\ &+ (x_2\,x_3\,x_4\,x_5 + x_1\,x_2\,x_4\,x_5 + x_1\,x_2\,x_3\,x_5 + x_1\,x_2\,x_3\,x_4 + x_1\,x_3\,x_4\,x_5)\,x - x_1\,x_2\,x_3\,x_4\,x_5 \end{aligned}$$

Auch diese (endliche) Liste von Ergebnissen ist natürlich noch kein Beweis des Vietaschen Wurzelsatzes. Um einen solchen zu erbringen gilt es, eine weitere Abstraktionsstufe erklimmen, auf der sich die (hoffentlich unendliche) Folge der beobachteten Regelmäßigkeiten einer (endlichen) mathematischen Argumentation erschließt. Die im letzten Kapitel hergeleitete rekursive Definition der elementarsymmetrischen Polynome zusammen mit einem Induktionsargument schließen die noch bestehende Lücke.

2.3 Symbolisches Rechnen und der Computer als Universalmaschine

Interessanterweise wiederholt sich die Entwicklung vom einfachen Kalkül der Arithmetik hin zu komplizierteren symbolischen Kalkülen in der Geschichte des Einsatzes des Computers als Hilfsmittel geistiger Arbeit. Historisch wurde das Wort Computer bekanntlich zuerst mit einer Maschine zur schnellen Ausführung numerischer Rechnungen verbunden. Nachdem dies in der Anfangszeit ebenfalls auf einfache arithmetische Operationen beschränkt war (und für Taschenrechner lange so beschränkt blieb), können auf entsprechend leistungsfähigen Maschinen heute auch kompliziertere Anwendungen wie das Berechnen numerischer Werte mathematischer Funktionen, die Approximation von Nullstellen gegebener Polynome oder von Eigenwerten gegebener Matrizen realisiert werden. Solche numerischen Verfahren spielen (derzeit) die zentrale Rolle in Anwendungen mathematischer Methoden auf Probleme aus Naturwissenschaft und Technik und bilden den Kern einer eigenen mathematischen Disziplin, des *Wissenschaftlichen Rechnens*[3].

All diesen Anwendungen ist gemein, dass sie zwar, unter Verwendung ausgefeilter Programmiersprachen, die Programmierfähigkeit eines Computers ausnutzen, sich letztlich aber allein auf das Rechnen mit (Computer)zahlen zurückführen lassen. Der Computer erscheint in ihnen stets als außerordentlich präzise und schnelle, im übrigen aber stupide *Rechen*maschine, als „number cruncher“.

Genau so kommt der Computer auch in vielen großen numerischen Simulationen praktischer Prozesse zum Einsatz, so dass ein Bild seiner Fähigkeiten entsteht, das sowohl aus innermathematischen als auch informatik-theoretischen Überlegungen heraus eher einer künstlichen Beschränkung seiner Einsatzmöglichkeiten gleichkommt. Zeigt uns doch die Berechenbarkeitstheorie in Gestalt der Church'schen These, dass der Computer eine *Universalmaschine* ist, die, mit einem geeigneten Programm versehen, prinzipiell in die Lage versetzt werden kann, jede nur denkbare algorithmische Tätigkeit auszuüben. Also insbesondere auch in der Lage sein sollte, Symbole und nicht nur Zahlen nach wohlbestimmten Regeln zu verarbeiten.

Dass ein Computer auch zu einer solchen Symbolverarbeitung fähig ist, war übrigens lange vor dem Bau des ersten „echten“ (von-Neumann-)Rechners bekannt. Bereits Charles Babbage (1792

[3] Allerdings definiert sich Wissenschaftliches Rechnen normalerweise nicht über die verwendeten Kalküle, sondern in Abgrenzung zur „reinen Mathematik“ über das zu Grunde liegende (und weiter oben bereits beschriebene) Anwendungsparadigma.

- 1838), der mit seiner „Analytical Engine“ 1838 ein dem heutigen Computer ähnliches Konzept entwickelte, ohne es aber je realisieren zu können, hatte diese Fähigkeiten einer solchen Maschine im Blick. Seine Assistentin, Freundin und Mäzenin, Lady Lovelace, schreibt (zitiert nach [9, S. 1]) :

> Viele Menschen, die nicht mit entsprechenden mathematischen Studien vertraut sind, glauben, dass mit dem *Ziel* von Babbage's Analytical Engine, Ergebnisse in Zahlennotation auszugeben, auch deren *Inneres* arithmetisch-numerischer Natur sein müsse statt algebraisch-analytischer. Das ist ein Irrtum. Die Maschine kann ihre numerischen Eingaben genau so anordnen und kombinieren, als wären es Buchstaben oder andere allgemeine Symbole; und könnte sie sogar *in einer solchen Form ausgeben*, wenn nur entsprechende Vorkehrungen getroffen würden.

Ein solches Computerverständnis ist uns, im Gegensatz zu den Pionieren des Computer-Zeitalters, im Lichte von ASCII-Code und Textverarbeitungssystemen heute allgemein geläufig, wenigstens was den Computer als intelligente Schreibmaschine betrifft. 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, *symbolische Information* verarbeiten zu können, bereits unmittelbar auch für den Umgang mit Daten zum Einsatz. Dabei verwischt sich, genau wie von Lady Lovelace vorausgesehen, durch die binäre Kodierung symbolischer Information die Grenze zwischen Zahlen und Zeichen, die zuerst so absolut schien.

Auf dieser Abstraktionsebene ist es auch möglich, die verschiedensten Nachschlagewerke und Formelsammlungen, also in symbolischer Form kodiertes Wissen, mit dem Computer aufzubereiten, in datenbankähnlichen Strukturen vorzuhalten und mit entsprechenden Textanalyseinstrumenten zu erschließen. Es wird sogar möglich, auf verschiedene Weise symbolisch kodierte Informationen in multimedialen Produkten zu verknüpfen, was das ungeheure innovative Potential dieser Entwicklungen verdeutlicht. In der Hand des Ingenieurs und Wissenschaftlers entwickelt sich damit der Computer zu einem sehr effektiven Instrument, das nicht nur den Rechenschieber, sondern auch zunehmend Formelsammlungen abzulösen in der Lage ist. Auf diesem Niveau handelt es sich allerdings noch immer um eine *syntaktische* Verarbeitung von Information, wo der Computer deren *Sinn* noch nicht in die Verarbeitung einzubeziehen vermag.

Aber auch der Einsatz des Computers zu *numerischen* Zwecken enthält bereits eine wichtige symbolische Komponente: Er hat das Programm für die Rechnungen in adäquater Form in seinen Speicher zu bringen und von dort wieder zu extrahieren. Diese Art symbolischer Information ist bereits *semantischer Art*, da die auf diese Weise dargestellten *Algorithmen* inhaltliche Aspekte der verarbeiteten Zahlengrößen erschließen.

Dass dies vom Nutzer nicht in gebührender Form wahrgenommen wird, hängt in erster Linie mit der strikten Trennung von (numerischen) Daten und (symbolischem) Programm sowie der Betrachtung des Computers als virtuelle Maschine („Das Programm macht der Programmierer, die Rechnung der Computer“) zusammen.

Bringt man beide Ebenen, die Daten und die Programme, zusammen, ermöglicht also algorithmische Operationen auch auf symbolischen Daten, geht man einen großen Schritt in die Richtung, semantische Aspekte auch symbolischer Information einer automatischen Verarbeitung zu erschließen. Dieser Gedanke wurde von den Gründervätern der künstlichen Intelligenz bereits Mitte der 60er Jahre beim Design von LISP umgesetzt. Denken lernt der Computer damit allerdings nicht, denn auch die Algorithmik symbolischer Informationsverarbeitung benötigt zunächst den in menschlicher Vorleistung erdachten *Kalkül*, welchen der Computer dann in der Regel schneller und präziser als der Mensch auszuführen vermag.

Im Schnittpunkt dieser modernen Entwicklungen befindet sich heute die *Computeralgebra*. Mit ihrem ausgeprägten Werkzeugcharakter und einer starken Anwendungsbezogenheit steht sie paradigmatisch dem (klassischen Gegenstand des) Wissenschaftlichen Rechnen nahe und wurde lange Zeit nur als Anhängsel dieser sich aus der Numerik heraus etablierten mathematischen Disziplin verstanden. Ihre Potenzen sind aber vielfältiger. Zunächst steht sie in einer Reihe mit anderen nichtnumerischen Applikationen einer „Mathematik mit dem Computer“ wie z.B. Anwendungen der diskreten Mathematik (Kombinatorik, Graphentheorie) oder der diskreten Optimierung, die *endliche* Strukturen untersuchen, die sich *exakt* im Computer reproduzieren lassen. „Mathematik mit dem Computer“ ist bereits damit mehr als Numerik.

Im Gegensatz zur diskreten Mathematik hat Computeralgebra mathematische Konstruktionen zum Gegenstand, die zwar syntaktisch endlich (und damit *exakt* im Computer darstellbar) sind, aber semantisch unendliche Strukturen repräsentieren können. Sie kommt damit mathematischen Arbeitstechniken näher als die anderen bisher genannten Gebiete. Siehe [2] für weitergehende Überlegungen zur Abgrenzung des Gegenstands des symbolischen Rechnens von numerischer und diskreter Mathematik.

Neben konkreten Implementierungen wichtiger mathematischer Verfahren reicht die Bedeutung der Computeralgebra aber über den Bereich der algorithmischen Mathematik hinaus. Die Vielzahl mathematischer Verfahren, die in einem modernen CAS unter einer *einheitlichen* Oberfläche verfügbar sind, machen dieses zu einem metamathematischen Werkzeug für Anwender, ähnlich den Numerikbibliotheken, die heute schon im Wissenschaftlichen Rechnen eine zentrale Rolle spielen.

In einer zu etablierenden „Computermathematik“ ([6]) als Symbiose dieser Entwicklungen werden computergestützte numerische, diskrete und symbolische Methoden gleichberechtigt nebeneinander stehen und in praktischen Applikationen sich gegenseitig befruchtend ineinander greifen sowie sich mit „denktechnologischen“ Fragen der Informatik verzahnen.

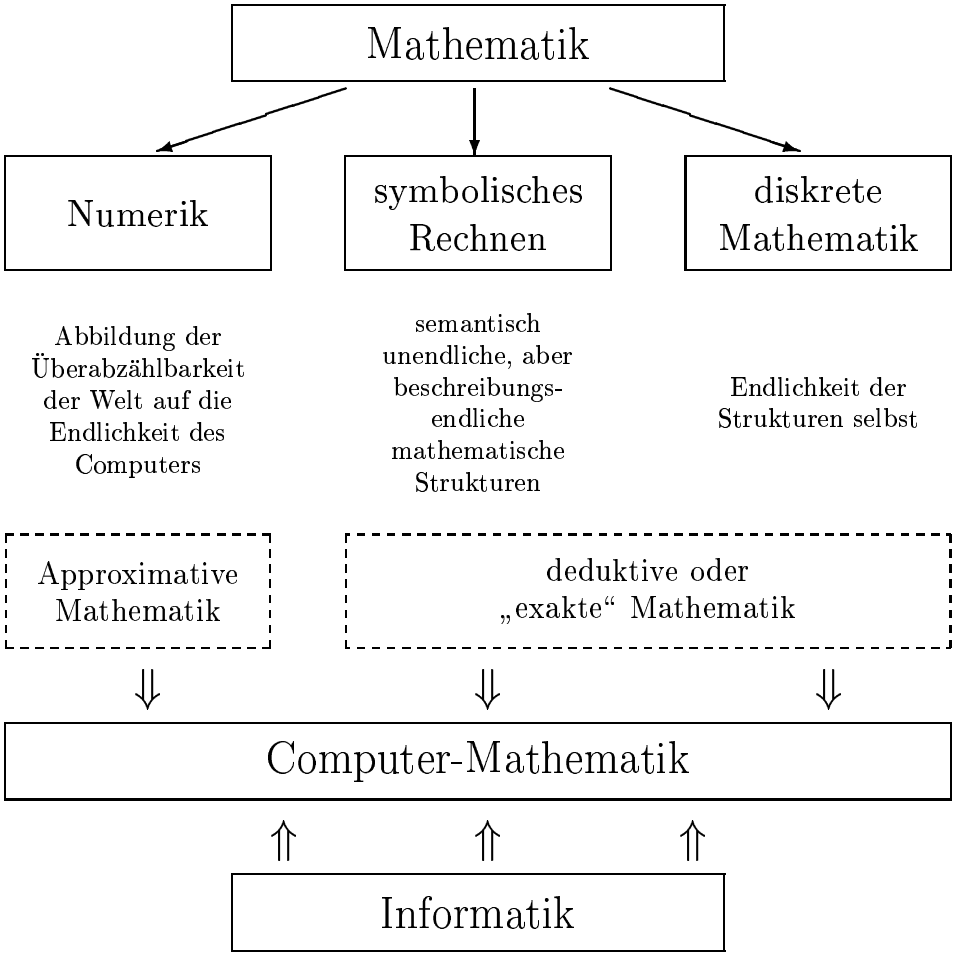
2.4 Numerische versus strukturelle Mathematik

Beim Versuch der Etablierung einer solchen „Computermathematik“ stoßen allerdings paradigmatisch unterschiedliche Welten aufeinander: Während die symbolischen Methoden der Computeralgebra, wenigstens im Prinzip, streng mathematisch deduktiver Natur sind, haben numerische Verfahren meist approximativen Charakter und sind damit zunächst auf ein quantitatives Bild der Realität ausgerichtet. Wir wollen deshalb einige wichtige Unterschiede zwischen numerischen und symbolischen Methoden herausarbeiten.

Eine solcher Vergleich ist auch deshalb interessant, weil die Erfahrungen, über die die meisten Leser dieses Skripts mit automatischen Rechnungen auf dem Computer verfügen, gerade aus dem Gebiet der Numerik kommen. Auch der größte Teil der Rechnungen, die heutzutage auf großen Mainframes und Rechnerverbänden ausgeführt werden, sind numerischer Natur und erfahren bestenfalls noch eine Ergänzung durch Visualisierungstools, die sich auf eben solche Grundlagen stützen.

Den typischen Unterschied zwischen beiden Sichtweisen beschreibt Pavelle in [12] wie folgt:

> Betrachten wir den einfachen Ausdruck $\frac{3\pi^2}{\pi}$. Jeder weiss, dass sich dieser Bruch zu 3π vereinfachen lässt, wenn man Zähler und Nenner durch π kürzt. Der numerische Wert von 3π kann interessieren, es kann aber auch sein, dass es ausreicht oder vielleicht sogar günstiger ist, diesen Ausdruck in seiner symbolischen, nichtnumerischen Form zu belassen. Mit einem nur auf numerische Operationen getrimmten Computer *muss* der Ausdruck $\frac{3\pi^2}{\pi}$ ausgewertet werden; wird dies mit einer Präzision von 10 Stellen ausgeführt, erhält man als Wert 9.424777958. Diese Zahl, abgesehen von der Tatsache, dass es sich dabei um eine eher uninformative Folge von Ziffern handelt, ist nicht



Die Genese der Computermathematik

dasselbe wie die Zahl, die man auf demselben Wege (d.h. durch Auswertung mit einer Genauigkeit von 10 Stellen) aus 3π bekommt. Letzteres liefert nämlich die Zahl 9.424777962, wobei die Differenz in den letzten beiden Stellen aus unumgänglichen Rundungsfehlern des Computers herrührt. Die Äquivalenz von $\frac{3\pi^2}{\pi}$ und 3π würde von solch einem Computer nicht einmal festgestellt werden.

Der Verlust struktureller Information, der im Rahmen numerischer Verfahren durch die Abbildung der Überabzählbarkeit der reellen Welt auf die Endlichkeit des Computers unvermeidlich ist, führt dazu, dass mit diesen Daten streng deduktiv basierte, also dem Rationalitätsanspruch einer „reinen" Wissenschaft genügende Argumentationen nicht mehr möglich sind. Symbolische Verfahren können dagegen ob der Endlichkeit ihrer Begriffswelt adäquat in die Endlichkeit eines Computers übertragen werden, ohne dadurch Teile ihrer strukturellen Aussagekraft einzubüßen.

Numerische Verfahren kann man einsetzen, um praktisch relevante Zahlenwerte *auszurechnen* (und mit geeigneten Visualisierungswerkzeugen darzustellen), symbolische Verfahren dagegen auch, um strukturelle mathematische Aussagen *zu beweisen.*

Numerische Verfahren gestatten es, eine große Zahl von Daten zu produzieren, in denen sich ein gewisser struktureller Zusammenhang widerspiegelt. Auf die *Allgemeingültigkeit* eines solchen Zusammenhangs (im Sinne wissenschaftlicher Rationalität) kann man aber aus noch so vielen Daten nicht schließen. Solche Datenmengen eignen sich auch nur beschränkt für die (nicht sensorische) Bearbeitung inverser Fragestellungen, die beim Steuern und Regeln von Prozessen auftreten.

Mit diesen Argumenten soll die vielfältig unter Beweis gestellte Bedeutung numerischer Verfahren und Ergebnisse nicht in Abrede gestellt werden. Allerdings ergeben sich im Vergleich von symbolischen und numerischen Methoden eine Reihe von Vorteilen, die eine strukturelle Sicht auf mathematische Fragestellungen gegenüber einer rein quantitativen bietet. Pavelle ([12]) listet die folgenden auf:

- *Erstens* gestattet eine präventive symbolische Analyse es oftmals, einen Ausdruck effektiver numerisch auszuwerten.

  Dieser einmalige vorherige Aufwand spielt besonders für Funktionen, die mit vielen verschiedenen Parameterbelegungen auszuwerten sind, eine entscheidende Rolle. Nach [13] werden moderne CAS zu 90 % gerade zu diesem Zweck, d.h. zur Generierung effizienten Codes, eingesetzt.

- *Zweitens* sind die so gewonnenen Aussagen durch das prinzipielle Vermeiden von numerischen Approximationen exakt in einem streng mathematisch-deduktiven Sinn. Damit werden Fehler- und Rundungsanalysen, die bei der Ergebnisverifikation numerischer Verfahren oftmals den Löwenanteil des Aufwands ausmachen, unnötig.

  Die Frage der Sicherung der Relevanz numerischer Simulationen ist eine zentrale Frage, die bei vielen praktischen Applikationen unbeantwortet im Raum stehen bleibt. Sie erfordern gewöhnlich umfangreiche Stabilitätsuntersuchungen, bei denen ihrerseits symbolische Methoden hilfreich sein können.

- Und *drittens* impliziert ein Ergebnis in einer symbolischen Form ein qualitativ tieferes Problemverständnis als es selbst durch eine große Menge rein numerischer Daten ausgedrückt werden kann.

  Zwar ermöglichen es (z.B. auf numerischen Verfahren basierende) Simulationen, größere Datenmengen zu erzeugen, die man nach *Regelmäßigkeiten* absuchen kann, jedoch erst eine (nur symbolisch mögliche) Analyse in einem streng deduktiven Verständnis von Mathematik vermag diese in *Gesetzmäßigkeiten* zu verwandeln, die unserem *Pool gesicherten Wissens* hinzugefügt werden können und die Basis für die Bearbeitung inverser Fragestellungen des Steuerns und Regelns bilden.

In *Theorien* fixierte symbolische Ergebnisse vermitteln einen wesentlich tieferen strukturellen Einblick in Zusammenhänge als Daten aus (rein numerischen) Simulationen, erfordern aber zugleich einen komplizierteren mathematischen Apparat. Sicher kann man längst nicht alles von praktischer Relevanz bis zu einer solchen strukturellen Sicht theoretisch aufarbeiten und verdichten. Sie ist und bleibt jedoch das eigentliche Ziel mathematischer (und nicht nur mathematischer) Erkenntnis. [12] zitiert in diesem Zusammenhang einen Ausspruch von R.W.HAMMING:

The purpose of computing is insight, not numbers.

Es gilt also, die Besonderheiten beider Gebiete ernst zu nehmen und zu einem neue Horizonte eröffnenden Zusammenspiel zu führen.

2.5 Was ist Computeralgebra ?

Was ist nun Computeralgebra? Oben hatten wir gesehen, dass sie eine spezielle Art von Symbolverarbeitung zum Gegenstand hat, in der, im Gegensatz zur Textverarbeitung, die Symbole mit Inhalten, also einer *Semantik*, verbunden sind. Auch geht es bei der Verarbeitung um Manipulationen eben dieser Inhalte und nicht primär der Symbole selbst.

Von der Natur der Inhalte und der Form der Manipulationen her können wir den Gegenstand der Computeralgebra also in erster Näherung als

symbolisch-algebraische Manipulationen mathematischer Inhalte

bezeichnen.

Gehen wir eher von der syntaktischen Form aus, in der uns diese Inhalte entgegentreten, so lässt sich der Gegenstand grob als

Rechnen mit Symbolen, die mathematische Objekte repräsentieren

umreißen. 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.

Das Adjektiv *symbolisch* bedeutet, dass das Ziel der mathematischen Problemstellung die Suche 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.

Der Begriff „algebraisch“ beinhaltet dabei keine Einschränkung auf spezielle Teilgebiete der Mathematik, sondern eine der verwendeten Methoden. Diese sind als mathematische Schlussweise weit verbreitet, denn auch Anwendungen aus der Analysis, die, wie z.B. Grenzwert- oder Integralbegriff, per definitionem Näherungsprozesse untersuchen, verwenden in ihrem eigenen Kalkül solche algebraischen Umformungen. Dies wird am Unterschied zwischen der Ableitungsdefinition und dem Vorgehen bei der praktischen Bestimmung einer solchen Ableitung deutlich, vgl. auch [10].

Computeralgebraische Werkzeuge beherrschen heute schon einen großen Teil der algorithmisch zugänglichen mathematischen und zunehmend auch naturwissenschaftlichen und ingenieurtechnischen Kalküle und bieten fach- und softwarekundigen Anwendern Zugang zu entsprechendem

Know-how auf Black-Box-Basis. Implementierungen fortgeschrittener Kalküle aus den Einzelwissenschaften sind ihrerseits nicht denkbar ohne Zugang zu effizienten Implementierungen der zentralen mathematischen Kalküle aus Algebra und Analysis.

Historisch stand und stehen dabei *Termumformungen*, also das Erkennen semantischer Gleichwertigkeit syntaktisch unterschiedlicher Ausdrücke, sowie das effiziente Rechnen mit polynomialen und rationalen Ausdrücken am Ausgangspunkt. Die hierbei verwendeten Methoden unterscheiden sich oftmals sehr von der durch „mathematische Intuition" gelenkten und stärker heuristisch geprägten Schlussweise des Menschen und greifen verstärkt die Entwicklungslinien der konstruktiven Mathematik mit ihrer vorerst letzten Blüte in den 1920er Jahren wieder auf, vgl. R.Loos in [10].

In diesem Sinne beschreibt J.Grabmeier in [6], auf R.Loos zurückgehend,

> *Computeralgebra als den Teil der Informatik und Mathematik, der algebraische Algorithmen entwickelt, analysiert, implementiert und anwendet.*

Das Beiwort „algebraisch" bezieht sich dabei, wie oben erläutert, auf die verwendeten Methoden. Buchberger verwendet deshalb in [2, S. 799] die genaueren Adjektive „exakt" und „abstrakt":

> *Symbolisches Rechnen* (dort als Synonym für Computeralgebra, vgl. ebenda S. 800 – HGG) *ist der Teil der algorithmischen Mathematik, der sich mit dem exakten algorithmischen Lösen von Problemen in abstrakten mathematischen Strukturen befasst.*

Im Weiteren unterstreicht Buchberger die Bedeutung der Algebraisierung und Algorithmisierung mathematischer Fragestellungen (der „Trivialisierung von Problemstellungen"), um sie einer computeralgebraischen Behandlung im engeren Sinne zugänglich zu machen, und schlägt diesen Aufwand dem symbolischen Rechnen zu. Allerdings werden so die Grenzen zu anderen mathematischen Teilgebieten verwischt, die sich zusammen mit der Computeralgebra im engeren Sinne arbeitsteilig an der Entwicklung und Implementierung der jeweiligen Kalküle beteiligen. Ein solches Verständnis blendet zugleich den technikorientierten Aspekt der Computeralgebra als Computerwissenschaft weitgehend aus.

Auch wenn die Übergänge zu anderen Gebieten der algorithmischen Mathematik fließend sind, werden wir im Folgenden den Gegenstand des symbolischen Rechnens stärker als Symbiose zwischen Mathematik und Computer verstehen und das Wort *Computeralgebra* in diesem Sinne verwenden. Mit Blick auf die zunehmende Kompliziertheit der entstehenden Werkzeuge sind dafür obige Definitionen noch zu erweitern um den Aspekt der

> *Entwicklung des zu Implementierung und Management solcher Systeme notwendigen informatik-theoretischen und -praktischen Instrumentariums.*

Die Computeralgebra befindet sich damit an der Schnittstelle zentraler Entwicklungen verschiedener Gebiete sowohl der Mathematik als auch der Informatik.

Im Computeralgebra-Handbuch [7], an dem weltweit über 200 bekannte Fachleute aus den verschiedensten Bereichen der Computeralgebra mitgearbeitet haben, wird das eigene Fachgebiet etwas ausführlicher wie folgt definiert:

> *Die Computeralgebra ist ein Wissenschaftsgebiet, das sich mit Methoden zum Lösen mathematisch formulierter Probleme durch symbolische Algorithmen und deren Umsetzung in Soft- und Hardware beschäftigt. Sie beruht auf der exakten endlichen Darstellung endlicher oder unendlicher mathematischer Objekte und Strukturen und ermöglicht deren symbolische und formelmäßige Behandlung durch eine Maschine. Strukturelles mathematisches Wissen wird dabei sowohl beim Entwurf als auch bei der Verifikation und Aufwandsanalyse der betreffenden Algorithmen verwendet. Die Computeralgebra kann damit wirkungsvoll eingesetzt werden bei der Lösung von mathematisch modellierten Fragestellungen in zum Teil sehr verschiedenen Gebieten der Informatik und Mathematik sowie in den Natur- und Ingenieurwissenschaften.*

In diesem Spannungsfeld zwischen Mathematik und Informatik findet die Computeralgebra zunehmend ihren eigenen Platz und nimmt wichtige Entwicklungsimpulse aus beiden Gebieten auf. So mag es nicht verwundern, dass die großen Durchbrüche der letzten Jahre sowohl in der Mathematik als auch in der Informatik die von der Computeralgebra produzierten Werkzeuge wesentlich beeinflusst haben und umgekehrt.

Mit der Computerisierung mathematischer Verfahren erweitert sich zugleich der Kognitionsbereich „durchschnittlicher" Mathematiker, so dass mathematische Ergebnisse, die noch vor einigen Jahrzehnten intellektuelle Vorposten darstellten, zum Allgemeingut werden. Eines der kuriosen Beispiele, das in diesem Zusammenhang immer wieder erwähnt wird, ist die Wiederholung der Mondbahnberechnungen, für die der französische Astronom C.DELAUNAY im 19. Jahrhundert 20 Jahre seines Lebens opferte und die mit modernen Computern in wenigen Stunden nachvollziehbar sind, vgl. [12]. Die mathematische Glaubwürdigkeit des Computerergebnisses ist zudem unvergleichlich höher, wie in der anerkennenden Feststellung zum Ausdruck kommt, dass sich DELAUNAY nur an drei Stellen in Termen untergeordneter Bedeutung verrechnet habe (und nicht umgekehrt der Computer).

Zusammenfassend lässt sich sagen, dass die Computeralgebra nicht eine weitere Computeranwendung schlechthin unter vielen anderen ist, sondern *in natürlicher Weise* Entwicklungen, die die Informatik als Ganzes hervorgebracht haben, weiterführt: Der Computereinsatz für symbolische Rechnungen eröffnet einen neuen Abschnitt auf dem Weg des Computers vom primitiven Bitknipser zu einem Universalwerkzeug für geistige Arbeit. Es beginnt damit eine neue Etappe auf dem Weg der praktischen Realisierung des theoretischen Anspruchs, den die Church'sche These impliziert.

Die von der Computeralgebra produzierten Werkzeuge spielen bereits heute eine stark zunehmende Rolle sowohl in den Natur- als auch in den Ingenieurwissenschaften. Dies dokumentiert sowohl die wachsende Zahl von Anwenderpaketen der verschiedenen großen Systeme als auch eine beeindruckende Zahl von Büchern zu dieser Thematik. Für einen vollständigeren Überblick über Tendenzen und Anwendungen der Computeralgebra sei auf das schon erwähnte Computeralgebra-Handbuch [7] verwiesen.

Ein pikantes Detail liegt in der Ignoranz dieser Entwicklungen durch Teile der etablierten Informatik selbst, denn Darlegungen zur Computeralgebra sucht man in verschiedenen Quellen, die sich eine umfassende Darstellung der Informatik vorgenommen haben, vergebens. So enthalten weder der „Duden Informatik" [4] noch das „Lexikon Informatik" [15] ein Stichwort *symbolisches Rechnen* oder *Computeralgebra*. Aber auch hier scheinen sich Gewichte zu verschieben, wie ein Blick in das neue „Handbuch Informatik" [14] belegt, in dem ein ganzer Abschnitt dem symbolischen Rechnen gewidmet ist (aus dem wir weiter oben bereits zitiert haben).

In der Tat kann man den Einfluss dieser neuen Arbeitsmittel auf den Umbruch unserer technisierten Arbeitswelt insbesondere im ingenieurtechnischen Bereich kaum überschätzen. Dort, wo vor einigen Jahren noch dicke Formelsammlungen und Tafelwerke das Berufsbild prägten, die trotz ihrer Dicke genau wie ein Berg numerischer Daten immer nur eine sehr beschränkte Sicht auf *Fakten* und keine *Einsichten* vermitteln konnten, halten zunehmend Computer-Werkzeuge Einzug, die auf symbolischen Fähigkeiten im beschriebenen Sinne aufsetzen.

Damit einher geht ein Umbruch dieser Bereiche geistiger Arbeit, der sie in vielleicht noch nachhaltigerer Weise revolutionieren wird als dies mit der Erfindung des Buchdrucks geschah. Schließlich eröffnen sich mit der Möglichkeit, nun auch algorithmisches Know How in großem Umfang zu vergegenständlichen, vollkommen neuen Dimensionen der Wissensrepräsentation. J. Grabmeier beschreibt die Perspektiven eines solchen

Übergangs von einer fakten- zu einer stärker algorithmenorientierten Wissensrepräsentation

in [6] wie folgt:

Viele Probleme aus der Ingenieurwelt, den Naturwissenschaften und den Wirtschaftswissenschaften sind heute ohne massiven Einsatz von Computern nicht lösbar. Die dahinterliegenden Probleme werden mit den Methoden des Wissenschaftlichen Rechnens angegangen. Dabei werden mehr und mehr die traditionellen numerischen Rechnungen durch symbolisches Rechnen mit dem Computer ersetzt bzw. ergänzt...

Der Siegeszug der Computeralgebra in den letzten Jahren ist eng gekoppelt mit der stürmischen Entwicklung von immer neuen Rechnergenerationen, die es erst möglich gemacht hat, die besonders rechen- und speicherintensiven Programme und Systeme zum symbolischen Rechnen zu realisieren.

Aber der Aufwand lohnt sich: Wenn man statt einer Zahl eine parameterabhängige Formel als Ergebnis erzielt, hat man nicht nur ein Problem gelöst, sondern eine Klasse von möglicherweise *unendlich vielen* Problemen erledigt. Dadurch wird ein Qualitätssprung möglich, denn die Formel erlaubt es nun z.B., die Parameter zu optimieren oder schnell auf Veränderungen zu reagieren. ...

Wie heute ein Taschenrechner zum Alltag gehört, wird künftig jeder Ingenieur und jeder, zu dessen Aufgaben das Lösen, Erlernen oder Lehren mathematischer Probleme gehört, Zugriff auf ein Computeralgebra-System haben. Die verschiedenen schon heute verfügbaren Komponenten für numerisches und symbolisches Rechnen, für Statistik und andere mathematische Gebiete, für Grafik und Animation, Textverarbeitung und Dokumentation mit Hypertext-Systemen und vieles mehr sehe ich in nicht allzu ferner Zukunft über entsprechende Schnittstellen zu individuell kombinierbaren Computermathematik-Systemen für das Wissenschaftliche Rechnen zusammenwachsen. Die Computeralgebra leistet damit einen wesenlichen Beitrag für eine der Schlüsseltechnologien unserer technikbestimmten Gesellschaft.

2.6 Computeralgebrasysteme (CAS) – Ein Überblick

2.6.1 Die Anfänge

Die ersten Anfänge der Entwicklung von Programmen der Computeralgebra reichen bis in die frühen 50er Jahre zurück. [19] nennt in diesem Zusammenhang Arbeiten aus dem Jahre 1953 von H.G.Kahrimanian (Temple University in Philadelphia) und von Norton (?) (Massachusetts Institute of Technology) zum analytischen Differenzieren. Kahrimanian schrieb in diesem Rahmen auch ein Assemblerprogramm für die Univac I, das damit als Urvater der CAS gelten kann.

Ende der 50er und Anfang der 60er Jahre wurden am MIT 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, weil in dessen Sprachkonzept die strenge Trennung von Daten- und Programmteil, deren Überwindung wir als für die Computeralgebra wesentlich herausgearbeitet hatten, bereits aufgehoben ist.

Die Wurzeln der ersten allgemeinen Systeme liegen in Versuchen verschiedener Fachwissenschaften, die im jeweiligen Kalkül anfallenden umfangreichen symbolischen Rechnungen einem Computer als Hifsmittel zu übertragen. Schwardmann [16] weist auf die Programme CAYLEY für Algorithmen in der Gruppentheorie und SAC zum Rechnen mit multivariaten Polynomen und rationalen Funktionen (Mathematik) sowie SHEEP für die Relativitätstheorie und MOA für die Himmelsmechanik (Physik) hin. [19] enthält einen detaillierteren Überblick über die Entwicklungen und Systeme jener Zeit.

Die ersten Systeme allgemeiner Ausrichtung, die nicht (nur) für spezielle Anforderungen eines Faches konzipiert wurden, die Systeme REDUCE und MACSYMA, basieren auf LISP und sind seit Mitte der 60er Jahre im Einsatz. REDUCE wurde von A. Hearn aus einem System für Berechnungen in der Hochenergiephysik (Feynman-Diagramme, Wirkungsquerschnitte) entwickelt, während

MACSYMA am MIT im Zusammenhang mit Untersuchungen zur künstlichen Intelligenz im Rahmen des DARPA-Programms entstand. GRABMEIER klassifiziert diese in [6] als *Allzwecksysteme der ersten Generation* und weist darauf hin, dass die programmiertechnischen Restriktionen jener Zeit (Lochstreifen, Stapelbetrieb) für symbolische Rechnungen, die aus noch darzulegenden Gründen meist stärker dialogorientiert sind, denkbar ungeeignete Bedingungen boten.

Mit der Entwicklung stärker dialogorientierter Rechentechnik in der 70er und 80er Jahren erhielten diese Entwicklungen neue Impulse. Sie beginnen ebenfalls, wie auch die des Computers als „number cruncher", bei der Arithmetik, hier allerdings der *Arithmetik symbolischer Ausdrücke*. Entsprechend bilden bei den Computeralgebrasystemen der zweiten Generation eine interaktiv zugängliche Polynomarithmetik zusammen mit einem regelbasierten Simplifikationssystem den Kern des Systemdesigns, um den herum mathematisches Wissen in Anwenderbibliotheken gegossen wurde und wird. Diese Systeme sind durch ein Zwei-Ebenen-Modell gekennzeichnet, in dem die Interpreterebene zwar gängige Programmablaufstrukturen unterstützt, jedoch keine Datentypen kennt, sondern es prinzipiell erlaubt, alle im Kernbereich, der *ersten Ebene*, implementierten symbolischen Algorithmen mit allen möglichen Daten zu kombinieren. Weitergehende programmiersprachliche Elemente wie insbesondere Typisierung werden nur rudimentär unterstützt.

Neben neuen Versionen der „alten" Systeme REDUCE und MACSYMA entstanden dabei eine Reihe neuer Systeme, von denen besonders MAPLE, MATHEMATICA und DERIVE zu nennen sind.

2.6.2 Mathematica

MATHEMATICA entwickelte sich aus dem von Steven Wolfram Ende der 70er Jahre entworfenen System SMP, das wiederum aus der Notwendigkeit geboren wurde, komplizierte algebraische Manipulationen in bestimmten Bereichen der theoretischen Physik effektiv und zuverlässig auszuführen. Es ist das erste System, das unmittelbar in der sich zu dieser Zeit im Compilerbereich durchsetzenden Sprache C geschrieben wurde. In der weiteren Entwicklung spielte dieses System eine Vorreiterrolle in der konsequenten Vermarktung von Software aus dem symbolischen Bereich, was bis heute in einer äußerst restriktiven Lizenzpolitik der inzwischen speziell für die Weiterentwicklung und den Vertrieb gegründeten Firma Wolfram Research, Inc. zum Ausdruck kommt. Der Versuch, schwerpunktmäßig über rein marktorientierte Mechanismen die Mittel für die Weiterentwicklung der algorithmischen und softwaretechnischen Seite eines solch komplexen Produkts aufzubringen, hat MATHEMATICA, besonders mit seinen Versionen 3.0 (Sommer 1996) und 4.0 (Frühjahr 1999), in die vorderste Front der „Großen" gebracht. Wolfram Research hat um sein Flaggschiff inzwischen eine ganze Infrastruktur mit Webportalen, Nutzerschulungen, Entwicklerkonferenzen sowie Büchern und Zeitschriften (zuletzt durch Gründung des Verlags Wolfram Media) aufgebaut, die MATHEMATICA über das eigentliche Softwareprodukt hinaus attraktiv machen. Eingeschlossen in dieses Engagement sind Plattformen wie MathSource (`http://library.wolfram.com/infocenter/MathSource`), über welche verschiedenste von Nutzern entwickelte und bereitgestellte MATHEMATICA-Pakete freizügig zugänglich sind, oder das Engagement für Eric Weissteins Online-Enzyklopädie MathWorld (`http://mathworld.wolfram.com`).

Trotz dieses Engagements im Geiste des Open-Source-Gedankens, der ein wichtiger Aspekt der Sicherung einer freizügig zugänglichen wissenschaftlichen Infrastruktur ist, werden die Aktivitäten von Steven Wolfram, ähnlich derer von Bill Gates, von der weltweiten Gemeinschaft der Computeralgebraiker teilweise mit großen Vorbehalten verfolgt.

Die aktuelle MATHEMATICA-Version 4.2 ist seit dem Frühjahr 2002 auf dem Markt.

2.6.3 Maple

Ähnliche Überlegungen des „Downsizing" der bis dahin nur auf Mainframes laufenden großen Systeme lagen der Entwicklung von MAPLE an der University of Waterloo zugrunde. In nur drei Wochen wurde Ende 1980 eine erste Version (mit beschänkten Fähigkeiten) entwickelt, die auf *B*, einer ressourcen- und laufzeitfreundlichen Sprache aus der BCPL-Familie aufsetzte, aus der sich

C als heutiger Standard entwickelt hat. Seit 1983 sind Versionen von MAPLE auch außerhalb der University of Waterloo in Gebrauch. Zur Gewährleistung einer effizienten Portabilität auf immer neue Computergenerationen wurde im Gegensatz zu den großen LISP-Systemen MACSYMA und REDUCE Wert auf einen kleinen, heute in C geschriebenen Systemkern gelegt, der die erforderlichen Elemente einer symbolischen Hochsprache implementiert, in der seinerseits die verschiedenen symbolischen Algorithmen geschrieben werden können.

Auch hier stellte sich schnell heraus, dass die Anforderungen, die der weltweite Vertrieb einer solchen Software, der Support einer Nutzergemeinde sowie die Portierung auf immer neue Rechnergenerationen stellen, die Möglichkeiten einer akademischen Anbindung sprengen und unter heutigen Bedingungen wohl nur über eine Software-Firma stabil zu gewährleisten ist. Diese Rolle spielt seit Ende 1987 Waterloo Maple Inc., die im Gegensatz zu Wolfram Research aber nach wie vor mit einer sehr engen Bindung an die universitäte Gruppe um K. Geddes und G. Labahn an der University of Waterloo, Ontario (Kanada), über ein ausgebautes akademisches Hinterland verfügt, das ein arbeitsteiliges Vorgehen in der softwaretechnischen und algorithmischen Weiterentwicklung des Systems ermöglicht. MAPLEs Internetportal ist unter `http://www.maplesoft.com` zu erreichen, so dass in Fachkreisen der Name „MapleSoft“ oft auch mit der Firma assoziiert wurde. Im Jahr 2002 gab deshalb Waterloo Maple das als den nunmehr offiziellen Firmennamen (its primary business name) bekannt.

Im Gegensatz zu MATHEMATICA verfolgt MAPLE eine stärker kooperative Politik auch mit anderen Softwarefirmen, um einerseits fremdes Know how für MAPLE verfügbar zu machen (etwa die Numerik-Bibliotheken von NAG, der Numerical Algorithms Group `http://www.nag.co.uk`, mit der Maple im Sommer 1998 eine strategische Allianz geschlossen hat) und andererseits in Softwareprodukte anderer Firmen Fähigkeiten zu symbolischen Rechnungen zu integrieren wie etwa in Mathcad (`http://www.mathcad.com`) oder Scientific Workplace, das „Word für Wissenschaftler“ (`http://www.mackichan.com`).

In den letzten zehn Jahren wurden MAPLE und MATHEMATICA, die bis dahin nur in mehr oder weniger experimentellen Fassungen vorlagen, mit größerem Aufwand unter marktorientierten Gesichtspunkten weiterentwickelt. Schwerpunkt waren dabei vor allem die Einbindung von bequemeren, fensterbasierten Ein- und Ausgabetechniken auf Notebook-Basis, hypertextbasierte Hilfesysteme und leistungsfähige Grafikmoduln. Sie besitzen damit heute in der Regel eine ausgereifte Benutzeroberfläche, sind gut dokumentiert und auf den verschiedensten Plattformen (bis hin zu leistungsfähigeren Personalcomputern unter den gängigen Betriebssystemen Windows sowie LINUX) 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.

2.6.4 Derive und CAS in der Schule

Einen anderen Weg der Kommerzialisierung gingen A.D. Rich und D.R. Stoutemyer mit der Gründung von Soft Warehouse, Inc. in Honolulu (Hawaii) ebenfalls im Jahre 1979. Neben Mainframes begannen zu dieser Zeit Arbeitsplatzcomputer mit geringen technischen Ressourcen eine zunehmend wichtige Rolle zu spielen, so dass die Frage entstand, ob man CAS mit ihren traditionell hohen Hardware-Anforderungen auch für solche Plattformen „zuschneiden“ kann. Mit dem System muMATH-79 und der (wiederum LISP-basierten) Sprache muLISP-79 wurde darauf eine überzeugende Antwort gefunden. Nach fast zehnjähriger „Ehe“ mit Microsoft nahm die Firma die weitere Entwicklung in die eigenen Hände und brachte 1988 ein Nachfolgeprodukt unter dem Namen DERIVE – *A Mathematical Assistant* auf den Markt, das mit minimalen Hardware-Anforderungen unter dem Betriebssystem DOS gute symbolische Fähigkeiten entwickelt. Nachdem in den folgenden Jahren durch die rasante Erweiterung der Hardware-Ressourcen von Arbeitsplatzrechnern dieser Anwendungsbereich auch von den anderen CAS zunehmend erobert wurde, konzentrierte sich die Firma auf das Taschenrechner-Geschäft und entwickelte in Zusammenar-

beit mit HP und TI zwei interessante Kleinstrechner mit symbolischen Fähigkeiten, den HP-486X (1991) und den TI-92 (1995).

Heute ist DERIVE ein Produkt, das mit großem Erfolg im schulischen Bereich eingesetzt wird. In Österreich existiert sogar eine landesweite Schullizenz, so dass Derive im Mathematikunterricht zum Einsatz gebracht werden kann. Weitergehende Informationen finden sich im „Austrian Center for Didactics of Computer Algebra“ (`http://www.acdca.ac.at`). In Deutschland sind Computeralgebrasysteme in Schulen in Modellprojekten vor allem in Form von Nachfolgeprodukten des TI-92 als Taschenrechner eingesetzt, siehe die Zusammenstellung der Computeralgebra-Fachgruppe unter `http://www.fachgruppe-computeralgebra.de/CLAW/bundeslaender.html`.

Um auf diesem Markt (dessen wirtschaftliche Potenzen die des gesamten wissenschaftlichen Bereichs um Größenordnungen übersteigen) erfolgreicher agieren zu können, wurde DERIVE im August 1999 von Texas Instruments aufgekauft. Wie weit solche Produkte mit Laptops, die die volle Leistungsfähigkeit „großer“ Systeme anbieten, konkurrieren können, wird die (nahe) Zukunft erweisen. Die (bereits seit mehreren Jahren) aktuelle Version DERIVE 5 ist nur für Windows-Plattformen erhältlich.

2.6.5 Entwicklungen der 90er Jahre – MUPAD

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. Das galt zunächst auch für das System MUPAD, dessen Entwicklung im Jahre 1989 von einer Arbeitsgruppe an der Uni-GH Paderborn unter der Leitung von Prof. B. Fuchssteiner begonnen und in den folgenden Jahren unter aktiver Beteiligung einer großen Zahl von Studenten, Diplomanden und Doktoranden intensiv vorangetrieben worden ist. MUPAD war von Anfang an als System allgemeiner Ausrichtung konzipiert. Sein grundlegendes Design orientierte sich an MAPLE, jedoch bereichert um einige moderne Software-Konzepte (Parallelverarbeitung, Objektorientierung), die bisher im Bereich des symbolischen Rechnens aus Gründen, die im nächsten Kapitel noch genauer dargelegt werden, kaum Verbreitung gefunden hatten. Mit den speziellen Designmöglichkeiten, die sich aus solchen Konzepten ergeben, gehört MUPAD bereits zu den CA-Systemen der dritten Generation.

Im Laufe der 90er Jahre entwickelte sich MUPAD, nicht zuletzt dank der freizügigen Zugangsmöglichkeiten, gerade für Studenten zu einer interessanten Alternative zu den stärker kommerziell aufgestellten „großen M“. Auch hier stellte sich heraus, dass ein solches System, wenn es eine gewisse Dimension erreicht hat, nicht allein aus dem akademischen Bereich heraus gewartet und gepflegt werden kann. Seit dem Frühjahr 1997 hat deshalb die eigens dafür gegründete Firma *SciFace* einen Teil dieser Aufgaben insbesondere aus dem software-technischen Bereich übernommen und begonnen, MUPAD nach ähnlichen Prinzipien wie MAPLE und MATHEMATICA aufzustellen.

Damit verbunden ist die kommerzielle Vermarktung von MUPAD, insbesondere der aufwändigen Windowsversion MUPAD Pro, die zu der Zeit eine sehr kontroverse Debatte in der deutschen Gemeinde der Computeralgebraiker auslöste. Schließlich waren die entsprechenden Entwicklungen zu einem großen Teil mit öffentlichen Geldern finanziert worden. Allerdings ließen die mit solcher Forschungsförderung einher gehenden Refinanzierungs-Zwänge der Paderborner Gruppe keine andere Wahl, wenn sie nicht entscheidendes (immer an konkrete Personen gebundenes) Know how verlieren wollte. Andererseits nimmt die MUPAD-Gruppe derartige Argumente ernst und stellt für Nutzer aus dem akademischen Bereich und insbesondere Studenten die Versionen MUPAD light und MUPAD für Linux zum freien download zur Verfügung, deren Rechenleistung mit der Profiversion identisch sind.

In den letzten Jahren ging die MUPAD-Gruppe insbesondere im Vertriebsbereich strategische Allianzen mit größeren Software-Häusern (McKichan) ein und versucht sich auch stärker im schulischen Bereich zu positionieren. MUPAD ist derzeit das einzige große CAS allgemeiner Ausrichtung

mit Bedeutung für einen akademisch-technischen Anwendermarkt, dessen „Headquarter“ sich in Europa befindet.

MAGMA ist ein zweites System, das seine Wurzeln außerhalb Amerikas hat und in den letzten Jahren an Bedeutung gewann. Es wird von einer Gruppe um John Cannon an der School of Mathematics and Statistics an der University of Sydney entwickelt und integriert ebenfalls moderne Aspekte der Objektorientierung wie parametrisierte Datentypen. Auch die MAGMA-Gruppe kann sich nicht vollständig aus öffentlichen Mitteln refinanzieren und hat ein Lizenzmodell entwickelt, mit dem die wissenschaftlichen Einrichtungen, die das System nutzen, an dessen Refinanzierung beteiligt werden. Die Rückläufe werden (nach Aussagen von J. Cannon) vollständig darauf verwendet, um – ähnlich Wolfram Research für MATHEMATICA – Entwickler mit vielversprechenden algorithmischen Ideen für eine begrenzte Zeit nach Australien einladen, damit sie Ihre Ideen in MAGMA implementieren. Entwickler von MAGMA und Mitarbeiter von Einrichtungen, die MAGMA lizensiert haben, können MAGMA unter besonderen Lizenz-Bedingungen freizügig nutzen. Die finanzielle Beteiligung wird also davon abhängig gemacht, in welchem Verhältnis jeweils auch das institutionelle Geben und Nehmen stehen.

System	neueste Version	Webseite	Preis Einzellizenz	Preis Stud.-version
Derive	5	`www.derive.com`	200 €	81 €
Maple	8	`www.maplesoft.com`	1200 $ (?)	199 €
Mathematica	4.2	`www.wri.com`	1415 €	150 €
Maxima	5.9.0	`maxima.sourceforge.net`	Open Source	
MuPAD	2.5 Pro	`www.mupad.de`	408 €	135 €
	Versionen Light und für Linux kostenfrei im akademischen Bereich			
Reduce	3.7.	`www.zib.de/Symbolik/reduce`	495 €	99 €

Die großen Computeralgebrasysteme allgemeiner Ausrichtung im Überblick
(Stand Mai 2003, Alle Preise ohne MWst.)

2.6.6 Computeralgebra – ein schwieriges Marktsegment für Software

Generell ist zu verzeichnen, dass im Laufe der 90er Jahre neben der algorithmischen Leistungsfähigkeit eine ausgewogene Lizenzpolitik, mit der die richtige Balance zwischen Refinanzierungserfordernissen einerseits und freizügigen Zugangsmöglichkeiten andererseits gefunden werden kann, für das weitere Schicksal der einzelnen Systeme zunehmend an Bedeutung gewonnen hat.

So gelang es weder MACSYMA noch REDUCE, mit den „großen M“ in Bezug auf Oberfläche sowie Vertriebs- und Vermarktungsaufwand Schritt zu halten. Trotz exzellenter algorithmischer Fähigkeiten und einer langjährig gewachsenen Nutzergemenide ging deshalb die Bedeutung beider Systeme im Laufe der 90er Jahre deutlich zurück.

Besonders interessant ist in diesem Zusammenhang das Schicksal von MACSYMA. Der Grundstein für das System wurde, wie bereits ausgeführt, in den 60er Jahren am MIT im Rahen eines Darpa-Projekts gelegt. Es entstand ein wirklich gutes System auf LISP-Basis, das in den 70er Jahren weite internationale Verbreitung in Wissenschaftlerkreisen fand und eng mit der Geschichte von Unix und dem ArpaNet verbunden war. Um 1980 herum war MACSYMA das weltweit beste symbolisch-numerisch-graphische Softwaresystem.

Im Jahre 1982 lizensierte das MIT MACSYMA an seine Spin-off-Company Symbolics Inc., womit die kommerzielle Seite des Lebens dieses Systems begann. Wegen der restriktiven amerikanischen Gesetzgebung konnten aber (glücklicherweise) nicht alle Rechte an die Firma übertragen werden, so dass neben der kommerziellen Variante auch nichtkommerzielle Versionen unter teilweise

leicht abgeänderten Namen (Vaxima, Maxima) kursierten und von interessierten Wissenschaftlern weiterentwickelt wurden.

Ende der 80er Jahre geriet Symbolics Inc. in zunehmende kommerzielle Schwierigkeiten und MACSYMA ging in den Jahren 1988 – 92 zunächst geneimsam mit Symbolics unter. Die Überreste wurden 1992 von Richard Petti aufgekauft und mit einer neu gegründeten Firma Macsyma Inc. wieder auf den Markt gebracht. In zwei größeren Benchmark-Tests für CAS schnitt MACSYMA sehr erfolgreich ab und im CA-Handbuch [7, S. 283 ff.] wird es noch gelobt. Allerdings war zum Erscheinungstermin des Buches (Anfang 2003) die neue Firma ebenfalls pleite und unter der dort noch zitierten Webadresse `http://www.macsyma.com` ist inzwischen ein Online-Shop zweifelhafter Qualität zu finden.

In all den Jahren war die Entwicklung einer freien Version von MACSYMA unter dem Titel MAXIMA von William Schelter vorangetrieben worden, so dass die noch verbliebenen MACSYMA-Anhänger nicht ganz mit leeren Händen dastanden. Schließlich bedeutet das Wegbrechen des Supports für ein solches CA-System, dass die darauf aufbauenden Forschungs- und Lehrmaterialien entwertet werden.

Mit dem Tod von Bill Schelter im Jahre 2001 stand die kleine MACSYMA-Nutzergemeinde erneut vor einer großen Herausforderung, die sie diesmal ganz im Geiste der GNU-Traditionen löste: Maxima ist das erste und bisher einzige große CAS, das heute unter der GPL als Open-Source-Projekt weiter vorangetrieben wird, siehe `http://maxima.sourceforge.net`.

Es existieren einige weitere Open-Source-Projekte zur Computeralgebra wie etwa YACAS (Yet Another Computaer Algebra System, siehe `http://yacas.sourceforge.net`), jedoch blieb deren Leistungsfähigkeit bisher ebenso beschränkt wie die der meisten firmenbasierten Versuche, mit Neuimplementierungen in das Marktsegment einzusteigen. Experten schätzen, dass sich die aufzubringende Entwicklungsleistung im algorithmischen Bereich für ein einigermaßen interessantes CAS allgemeiner Ausrichtung im Bereich von einigen hundert Mannjahren bewegt, so dass erfolgversprechende Ansätze nur denkbar sind

- auf der Basis eines der großen Systeme, dessen Quellen unter die GPL gestellt werden („Netscape-Modell“)
- auf der Basis eines dichten weltweiten Netzwerks von Computeralgebraikern, welche in der Lage sind, die bisher implementierten Bausteine zusammenzutragen und zu integrieren oder
- wenn eine große Software-Firma mit langem Atem entsprechende Vorlaufforschung in ihren eigenen Labs finanziert.

Für alle drei Ansätze gibt es Beispiele. Den ersten hatten wir mit MAXIMA bereits belegt. Der zweite wird derzeit im Rahmen des OSCAS-Projekts (OSCAS = Open Source Computer Algebra System) besonders in Frankreich vorangetrieben, siehe `http://www.univ-orleans.fr/EXT/ASTEX`.

Für den dritten Ansatz möge IBM als Beispiel dienen, die sich als eine der großen Softwarefirmen seit den Anfangstagen der Computeralgebra mit eigenen Aktivitäten an den wichtigsten Entwicklungen beteiligt hat. Das betrifft insbesondere nach einigen Sprachentwicklungen in den 60er Jahren das System SCRATCHPAD, mit dem im *IBM T.J. Watson Research Center at Yorktown, NY,* seit den 70er Jahren diese Untersuchungen fortgesetzt wurden. SCRATCHPAD verwendete als damals einziges System im Design ein strenges Typkonzept. Bis zum Beginn der 90er Jahre standen diese Entwicklungen ausschließlich auf IBM-Rechnerplattformen und nur in experimentellen Versionen zur Verfügung und fanden damit trotz ihrer Exzellenz keine weite Verbreitung. Auch mit der ersten offiziellen Version von AXIOM im Jahre 1991 änderte sich daran wenig. Das mag IBM bewogen haben, 1992 im Zuge einer generellen „Flurbereinigung“ des Firmenprofils die Rechte an AXIOM der Firma NAG Ltd. zu übertragen, einer Non-profit-Firma, die sich bereits auf dem Gebiet wissenschaftlicher Software ausgewiesen hatte. Um eine Schnittstelle zu ihrem Hauptprodukt, der NAG Numerik-Bibliothek, erweitert, wurde AXIOM in den folgenden Jahren auf eine Vielzahl von Plattformen portiert und auch der zugehörige Standalone-Compiler ALDOR weiterentwickelt.

Im Sommer 1998 jedoch straffte NAG seine Produktlinien und begann, sich im Computeralgebrabereich strategisch auf MAPLE zu orientieren. AXIOM und ALDOR werden heute von NAG nicht mehr unterstützt.

Damit endet derzeit die 30-jährige Geschichte eines weiteren wichtigen Computeralgebra-Projekts. Jedoch gibt es Bemühungen im Rahmen des OSCAS-Projekts, diese Leistungen der wissenschaftlichen Öffentlichkeit zu erhalten bzw. neu zugänglich zu machen, und erste Erfolge sind mit der Einrichtung der Website `http://www.aldor.org` zu verzeichnen.

2.6.7 Computeralgebra jenseits der „Main Road“

AXIOM ist mit seinem streng getypten Ansatz ein

Computeralgebrasystem der dritten Generation,

mit denen versucht wird, ins Zentrum des Designs moderne programmiertechnische Ansätze der Typisierung, Modularisierung und Datenkapselung zu setzen und diese konsequent in einem symbolisch orientierten Kontext zu realisieren. Computeralgebrasysteme bieten hierfür denkbar gute Voraussetzungen, denn ein solches Typsystem ist ja seinerseits symbolischer Natur mit einem stark algebraisierten Hintergrund, sollte also prinzipiell mit den vorhandenen sprachlichen Mitteln eines CAS modelliert werden können. Andere Versuche, Typ-Informationen mit den Mitteln eines CAS der zweiten Generation zu modellieren, wurden auch mit dem Maple-Paket Gauss ([8]) sowie dem Domain-Konzept in MUPAD vorgelegt. Die *Integration* eines solchen Typsystems in die Programmiersprache bedeutet jedoch, diese symbolischen Manipulationen unterhalb der Interpreterebene zu vollziehen, also gegenüber Systemen der zweiten Generation eine weitere Ebene zwischen Interpreter und Systemkern einzufügen, die diese Typinformationen in der Lage ist auszuwerten. Dieses Konzept wird von ALDOR, dem aus AXIOM abgespaltenen Sprachkonzept, mit beeindruckenden Ergebnissen verfolgt. Es zeigt sich, dass die algebraische Natur des Targets dieser Bemühungen mit der algebraischen Natur der theoretischen Fundierung von programmiertechnischen Entwicklungen gut harmoniert und etwa mit parametrisierten Datentypen und virtuellen Objekten (bzw. abstrakten Datentypen) bereits zu einer Zeit experimentiert wurde, in der an die bis heute rudimentären Versuche mit „Templates“ und Ähnlichem in den „großen Sprachfamilien“ C/C++ bzw. PASCAL/MODULA/OBERON noch nicht zu denken war. Besonders deutlich wird dies im Design von MAGMA, einem Nachfolger von CAYLEY, das für komplexe Fragestellungen aus den Bereichen Algebra, Zahlentheorie, Geometrie und Kombinatorik entworfen wurde und Konzepte der universellen Algebra und Kategorientheorie in einem objektorientierten Ansatz direkt umsetzt.

Im Zusammenhang mit der Entwicklung paralleler Plattformen und verteilter Systeme auf Client-Server-Basis wird auch zunehmend über die Bereitstellung von symbolischer Rechenleistung in einem solchen Kontext nachgedacht. Hierfür sind in die gängigen Computeralgebrasysteme neue Kommunikationskonzepte zu integrieren, die an die besonderen Anforderungen an Struktur und Umfang des Datenaustauschs bei symbolischen Rechnungen angepasst sind.

Daneben gibt es eine große Anzahl kleinerer experimenteller Systeme mit eingeschränkter Ausrichtung. Sie wurden für spezielle Kalküle auf begrenzten Gebieten der Mathematik oder Physik entwickelt und erreichen durch 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 MAGMA und GAP (Gruppentheorie), PARI, KANT, SIMATH (Zahlentheorie), Macaulay, Macaulay-2, CoCoA und Singular (Algebra und algebraische Geometrie) und LIE (Lietheorie).

2.6.8 Und wie wird es weitergehen ?

Im Zuge der Herausbildung einer „Computermathematik“, wie in [6] prognostiziert, entsteht zunehmend die Frage nach der Integration von Softwareentwicklungen, die bisher in anderen Kreisen vorangetrieben und gepflegt wurden. Dies betrifft insbesondere die Interaktion zwischen

- symbolischen Verfahren der Computeralgebra im engeren Sinne,
- in sehr umfangreichen Programmpaketen vorhandene ausgefeilte numerische Applikationen des „wissenschaftlichen Rechnens“ sowie
- Entwicklungen der Computergraphik.

Die heute existierenden Computeralgebrasysteme haben sowohl numerische als auch grafische Fähigkeiten, die durchaus beeindrucken, jedoch von der Leistungsfähigkeit der genannten „professionellen“ Entwicklungen weit entfernt sind. Aktuelle Bemühungen sind deshalb darauf gerichtet, in Zukunft die Vorteile arbeitsteiligen wissenschaftlichen Vorgehens stärker zu nutzen, statt das Fahrrad mehrfach zu erfinden. Auf der Softwareseite finden sich solche Bemühungen in Designkonzepten wieder, die sich stärker an der eigenen *Kernkompetenz* orientieren und den „Rest“ durch Anbindung an andere Kernkompetenzen abdecken. Dies erfolgte für die großen CAS zuerst im Grafikbereich, wo sich AXIOM (Open Inventor) und REDUCE (Gnuplot) an eigenständige Entwicklungen im Grafiksektor angekoppelt hatten. Weiterhin spielen Verbindungen zu Numerikpaketen wie die AXIOM- bzw. MAPLE-Anbindungen an die Fortranbibliothek f90 oder die Scilab-Aktivitäten der MUPAD-Gruppe eine zunehmend wichtige Rolle. Auch von der Numerikseite her gibt es derartige Aktivitäten, wie die Einbindung von symbolischen Fähigkeiten von MAPLE in das im ingenieurtechnischen Bereich stark verbreitete System MathCad belegt.

Inzwischen ist dies zu einem deutlich sichtbaren Trend geworden und eigene Protokolle wie etwa MathLink oder JLink für die Kommunikation zwischen C bzw. Java und MATHEMATICA entwickelt worden. Auch die MAPLE-Maplets (seit Version 8) sowie MUPADs dynamische Moduln weisen in diese Richtung. Darüber hinaus gibt es inzwischen mit MathML sowie OpenMath Bemühungen, ein eigenes XML-Protokoll für symbolische Rechnungen zu entwickeln, um den Datenaustausch zwischen verschiedenen Applikationen aus diesem Gebiet generell zu erleichtern.

Der dritte große Bereich des Zugriffs auf Fremdkompetenz liegt in der Gestaltung des Ein- und Ausgabedialogs der großen Systeme. Mathematische Formeln sind oft auch von drucktechnisch komplizierter Gestalt und verwenden eine Fülle extravaganter Symbole in den verschiedensten Größen. Hierfür hat sich im Bereich der mathematischen Fachliteratur das Satzsystem TEX von D.E.Knuth und dessen etwas komfortablere Umgebung LATEX als ein de facto Standard weitgehend durchgesetzt. Deshalb bieten die meisten großen CAS auch eine Ausgabe in dieser Form an, die es erlaubt, Formeln direkt in mathematische Texte zu übernehmen bzw. diese direkt mit entsprechenden Wiedergabewerkzeugen in optisch ansprechender Form auszugeben. Ein großer Schritt vorwärts in dieser Richtung wurde durch die Notebook-Oberflächen von MATHEMATICA (seit Version 3.0) und MAPLE (seit Version 5) erreicht, die als direktes Grafik-Frontend für eine solche Darstellungsart ausgelegt sind. Inzwischen sind andere Systeme (etwa MUPAD) nachgezogen und mit dem TeXmacs-Projekt (`http://www.texmacs.org`) gibt es auch im Open-Source-Bereich entsprechende Initiativen.

Computeralgebrasysteme werden damit zunehmend bequeme Werkzeuge für die eigene geistige Arbeit, die als *persönlicher digitaler Assistent (PDA)* lokal auf dem Schreibtisch des Wissenschaftlers oder Ingenieurs einen immer größeren Teil des globalen Know Hows verschiedener Fachrichtungen in einer auch algorithmisch leicht zugänglichen Form bereithalten.

In Verbindung mit Client-Server-Techniken eröffnen sich für diese Entwicklungen nochmals vollkommen neue Perspektiven, die derzeit verstärkt untersucht werden: Ein „Kontrollzentrum“

mit im wesentlichen nur Notebook- und Kommunikationsfähigkeiten als Mensch-Maschinensystem-Schnittstelle hat Zugang zu einer Vielzahl von Spezialprogrammen unterschiedlicher Kompetenz, die die gestellten Aufgaben arbeitsteilig lösen. So kann z.B. ein Programm den symbolischen Teil der Aufgabe bearbeiten, das Ergebnis (über das Kontrollzentrum) zur numerischen Auswertung einem zweiten Programm zur Verfügung stellen, dieses daraus Daten für eine grafische Auswertung erzeugen, die einem dritten Programm zur Grafikausgabe weitergereicht werden. Solche primär an funktionalen und nicht an ökonomischen Aspekten orienterte Kooperationsverbindungen würden auch im Bereich der „Web Services“ einiges vom Kopf auf die Füße stellen.

Derartige Systeme würden es dann auch gestatten, auf die global und dezentral verteile Kompetenz anderer *unmittelbar* zuzugreifen, indem (wenigstens für ausgewählte Fragen, denn das Ganze wäre auch teuer) jeweils die neuesten und fortgeschrittensten Algorithmen und die leistungsfähigsten Implementierungen auf besonders diffizile Fragen angewendet werden könnten.

Wagen wir einen **Blick in die Zukunft:** Die Möglichkeiten, die sich aus einer zunehmenden Vernetzung auf der einen Seite und Allgegenwart von Computern auf der anderen abzeichnen, haben wir bisher noch gar nicht erfasst. In naher Zukunft wird es mit diesen Instrumenten möglich sein, zunehmend *selbst* in einem heterogenen Informationsraum zu operieren, statt mit vorgefertigten Instrumenten vor Ort eine *vorab generierte Kompetenz* zu konsultieren. Man wird statt dessen

1. weltweit auf von Spezialisten an verschiedenen Orten gepflegte Kompetenz unmittelbar zugreifen können
2. dezentral Messdaten mit Monitoring-Verfahren erfassen können (Brückenprüfung, Patientenüberwachung,...)
3. die damit generierten Erfahrungen selbst unmittelbar in diesem Informationsraum einbringen und damit anderen zugänglich machen können (World Wide Web)

Die Entstehung derartiger „kollektiver Vernunftformen“ wird den Weg in die viel beschworene Wissensgesellschaft ganz entscheidend prägen.

Kapitel 3

Aufbau und Arbeitsweise eines Computeralgebrasystems (CAS) der zweiten Generation

In diesem Kapitel wollen wir uns näher mit dem Aufbau und der Arbeitsweise eines Computeralgebrasystems der zweiten Generation vertraut machen und dabei insbesondere Unterschiede und Gemeinsamkeiten mit ähnlichen Arbeitsmitteln aus dem Bereich der Programmiersysteme herausarbeiten.

3.1 CAS. Eine Anforderungsanalyse

Interpreter versus Compiler

Programmiersysteme werden zunächst grob in Interpreter und Compiler unterschieden. CAS haben wir bisher ausschließlich über eine interaktive Oberfläche, also als Interpreter, kennengelernt. Es erhebt sich damit die Frage, ob es gewichtige Gründe gibt, symbolische Systeme gerade so auszulegen.

Interpreter und Compiler sind Programmiersysteme, die dazu dienen, die in einer Hochsprache fixierte Implementierung eines Programms in ein Maschinenprogramm zu übertragen und auf dem Rechner auszuführen.

Beide Arten durchlaufen dabei die Phasen *lexikalische Analyse, syntaktische Analyse* und *semantische Analyse* des Programms. Ein **Interpreter** bringt den aktuellen Befehl nach dieser Prüfung *sofort* zur Ausführung.

Ein **Compiler** führt in einer ersten Phase, der **Übersetzungszeit**, die Analyse des vollständigen Programms aus, übersetzt dieses in eine maschinennahe Sprache und speichert es ab. Er durchläuft dabei die weiteren Phasen *Codegenerierung* und evtl. *Codeoptimierung.* In einer zweiten Phase, zur **Laufzeit**, wird das übersetzte Programm von einem *Lader* zur Abarbeitung in den Hauptspeicher geladen.

Ein Compiler betreibt also einen größeren Aufwand bei der Analyse des Programms, der sich in der Abarbeitungsphase rentieren soll. Das ist nur sinnvoll, wenn dasselbe Programm mit mehreren Datensätzen ausgeführt wird. Das zentrale Paradigma des Compilers ist also das des *Programmflusses*, längs dessen Daten geschleust werden.

Compiler werden deshalb vorwiegend dann eingesetzt, wenn ein und dasselbe Programm mit einer Vielzahl unterschiedlicher Datensätze abgearbeitet werden soll und die (einmaligen) Übersetzungszeitnachteile durch die (mehrmaligen) Laufzeitvorteile aufgewogen werden.
Dies erfolgt besonders bei einer Sicht auf den Computer als "virtuelle Maschine", d.h. als programmierbarer Rechenautomat.

Compiler sind typische Arbeitsmittel einer stapelorientierten Betriebsweise.

Ein **Interpreter** dagegen zeichnet sich durch eine höhere Flexibilität aus, da auch noch während des Ablaufs in das Geschehen eingegriffen werden kann. Werte von (globalen) Variablen sind während der Programmausführung abfrag- und änderbar und einzelne Anweisungen oder Deklarationen im Quellprogramm können geändert werden. Ein Interpreter ist also dort von Vorteil, wo man verschiedene Daten auf unterschiedliche Weise kombinieren und modifizieren möchte. Das zentrale Paradigma des Interpreters ist also das der *Datenlandschaft*, die durch Anwendung verschiedener Konstruktionselemente erstellt und modifiziert wird.

Als entscheidender Nachteil schlagen längere Rechenzeiten für einzelne Operationen zu Buche, da z.B. die Adressen der verwendeten Variablen mit Hilfe der Bezeichner ständig neu gesucht werden müssen. Ebenso ist Code-Optimierung nur beschränkt möglich.

Interpreter werden deshalb vor allem in Anwendungen eingesetzt, wo diese Nachteile zugunsten einer größeren Flexibilität des Programms, der Möglichkeit einer interaktiven Ablaufsteuerung und der Inspektion von Zwischenergebnissen in Kauf genommen werden.

Interpreter sind typische Arbeitsmittel einer dialogorientierten Betriebsweise.

Eine solche Flexibilität ist eine wichtige Anforderung an ein CAS, wenn es als *Hilfsmittel für geistige Arbeit* eingesetzt werden soll. Dies ist folglich auch der Grund, weshalb alle großen CAS wenigstens in ihrer obersten Ebene Interpreter sind.

Von dieser Dialogfläche aus werden einzelne Datenkonstruktoren (Funktionen, Unterprogramme) aufgerufen, für die jedoch eine andere Spezifik gilt: Bei diesen handelt es sich um standardisierte, auf einer höheren Abstraktionsebene optimierte algorithmische Lösungen, die während des Dialogbetriebs mit einer Vielzahl unterschiedlicher Datensätze aufgerufen werden (können). Sie gehören also in das klassische Einsatzfeld von Compilern.

Es ist deshalb nur folgerichtig, diese Teile eines CAS in vorübersetzter (und optimierter) Form bereitzustellen. Allerdings trifft dies nicht nur auf Systemteile selbst zu. Auch der Nutzer sollte die Möglichkeit haben, selbstentwickelte Programmteile, die mit mehreren Datensätzen abgearbeitet werden sollen, zu übersetzen, um in den Genuss der genannten Vorteile zu kommen.

Entsprechend dieser Anforderungsspezifikation sind große CAS allgemeiner Ausrichtung, wie übrigens heute alle größeren Interpretersysteme,

top-level interpretiert, low-level compiliert.

Bibliotheken, Pakete, Module

Bei der Übersetzung von Programmteilen gibt es drei Ebenen, die unterschiedliche Anforderungen an die Qualität der Übersetzung stellen:

- Während der Systementwicklung erstellte Übersetzungen, die in Form von Bibliotheken grundlegender Algorithmen mit dem System mitgeliefert (oder nachgeliefert) werden.
- Eigenentwicklungen, die mit geeigneten Instrumenten übersetzt und archiviert und damit in nachfolgenden Sitzungen in bereits compilierter Form verwendet werden können.

- Im laufenden Betrieb erzeugte Übersetzungen, die für nachfolgende Sitzungen nicht mehr zur Verfügung stehen (müssen).

Schalenbilder Programmsphären, Nutzersphären

Beispiele für die **erste Ebene** sind die Implementierungen der wichtigsten mathematischen Algorithmen, die die Leistungskraft eines CAS bestimmen. Hier wird man besonderen Wert auf den Entwurf geeigneter Datenstrukturen sowie die Effizienz der verwendeten Algorithmen legen, wofür neben entsprechenden programmiertechnischen Kenntnissen auch profunde Kenntnisse aus dem jeweiligen mathematischen Teilgebiet erforderlich sind.

Beispiele für die **zweite Ebene** sind Anwenderentwicklungen für spezielle Zwecke, die auf den vom System bereitgestellten Algorithmen aufsetzen und so das CAS zum Kern einer (mehr oder weniger umfangreichen) speziellen Applikation machen. Solche Applikationen reichen von kleinen Programmen, mit denen man verschiedene Vermutungen an entsprechendem Datenmaterial testen kann, bis hin zu umfangreichen Sammlungen von Funktionen, die Algorithmen aus einem bestimmten Gebiet von Mathematik, Naturwissenschaft oder Technik zusammenstellen.

Beispiele für die **dritte Ebene** sind schließlich zur Laufzeit entworfene Funktionen, die mit umfangreichem Datenmaterial ausgewertet werden müssen, wie es etwa beim Erzeugen einer Grafikausgabe anfällt.

Natürlich sind die Grenzen zwischen den verschiedenen Ebenen fließend. So muss im Systemdesign der ersten Ebene beachtet werden, dass nicht nur eine Flexibilität hin zu komplexeren Algorithmen zu gewährleisten ist, sondern auch eine Flexibilität nach unten, damit das jeweilige CAS möglichst einfach an unterschiedliche Hardware und Betriebssysteme angepasst werden kann. Hierfür ist es sinnvoll, das Prinzip der *virtuellen Maschine* anzuwenden, d.h. Implementierungen komplexerer Algorithmen in einer maschinenunabhängigen Zwischensprache zu erstellen, die dann von leistungsfähigen Werkzeugen plattformabhängig übersetzt und optimiert werden.

Die einzelnen CAS sind deshalb so aufgebaut, dass in einem Systemkern (unterschiedlichen Umfangs) Sprachelemente und elementare Datenstrukturen eines leistungsfähigen Laufzeitsystems implementiert werden, in dem dann seinerseits komplexere Algorithmen codiert sind.

Auch zwischen der ersten und zweiten Ebene ist eine strenge Abgrenzung nicht möglich, da die Implementierung guter konstruktiver mathematischer Verfahren eine gute Kenntnis des zugehörigen theoretischen Hintergrunds und damit oft eine akademische Einbindung dieser Entwicklungen wünschenswert macht oder gar erfordert. Außerdem werden in der Regel von Nutzern entwickelte besonders leistungsfähige spezielle Applikationen neuen Versionen des jeweiligen CAS hinzugefügt, um sie so einem weiteren Nutzerkreis zugänglich zu machen. In beiden Fällen ist es denkbar und auch schon eingetreten, dass auf diese Weise entstandene Programmstücke aus Performance-Gründen Auswirkungen auf das Design tieferliegender Systemteile haben.

CAS als komplexe Softwareprojekte

Auf diese Weise entstanden aus den Versionen der verschiedenen CAS, die bis Mitte der 80er Jahre von überschaubaren Personengruppen meist an einzelnen wissenschaftlichen Einrichtungen entwickelt wurden, Softwareprodukte, die vielfältigen "in Bytes gegossenen" Sachverstand bereithalten und deren weitere Entwicklung von einem schwer zu überschauenden Kreis von Nutzern und Entwicklern vorangetrieben wird.

Das ist aber nur möglich, wenn große Teile des Systemdesigns selbst offen liegen und auch die technischen Mittel, die Nutzern und Systementwicklern zur Übersetzung und Optimierung von Quellcode zur Verfügung stehen, in ihrer Leistungsfähigkeit nicht zu stark voneinander differieren. Aus einer solchen Interaktion ergibt sich als weitere Forderung an Design *und* Produktphilosophie, dass Computeralgebrasysteme **offene Systeme** sein müssen.

Diese inhaltliche und oft auch personelle Interaktion zwischen Systementwicklern und Nutzern ist die wohl entscheidende Quelle der enormen Leistungskraft heutiger CAS. Dabei sind die Rollen zwischen Entwicklern und Anwendern von CAS nicht so klar verteilt wie bei vielen anderen Programmpaketen, sind doch gerade die *mathematischen* Fähigkeiten der großen CAS aus einer Vielzahl von Aktivitäten zur programmtechnischen Fixierung mathematischen Know-Hows entstanden, an denen sich Arbeitsgruppen beteiligen, die rund um die Welt verteilt sind und nur sehr lose Kontakte zueinander und zum engeren Kreis der Systementwickler halten. Letztere tragen hauptsächlich die Verantwortung für die Weiterentwicklung der entsprechenden programmiertechnischen Instrumentarien. Die Kommunikation zwischen diesen Gruppen erfolgt über Mailing-Listen, News-Gruppen, Anwender- und Entwicklerkonferenzen, Artikel in Zeitschriften, Bücher etc., insgesamt also mit den im üblichen Wissenschaftsbetrieb anzutreffenden Mitteln. Natürlich ergeben sich für ein konsistentes Management der Anstrengungen eines solch heterogenen Personenkreises im Umfeld des jeweiligen CAS

hohe Anforderungen auch im organisatorischen Bereich, die weit über Fragen des reinen Software-Managements hinausgehen

und mit den Anforderungen, die ein klassisches Produkt stellt, bei dem Nutzer und Entwicklerkreis streng getrennt sind, kaum zu vergleichen sind.

Die großen CAS haben eher den Charakter von Entwicklungsumgebungen bzw. -werkzeugen, die zu verschiedenen, von den Entwicklern und Softwarefirmen nicht vorhersehbaren Zwecken vor allem im wissenschaftlichen Bereich eingesetzt werden. Dabei entstand und entsteht eine Vielzahl von Materialien unterschiedlicher Qualität und Ausrichtung, welche von den meisten Autoren – wie in Wissenschaftskreisen üblich – frei zitier- und nachnutzbar zur Verfügung gestellt werden.

Diese zu sammeln und zu systematisieren war schon immer ein Anliegen der weltweiten Gemeinde der Anhänger der verschiedenen CAS. In den letzten Jahren wurden diese frei verfügbaren Sammlungen, die von MAPLE als *Maple Shared Library* oder von MATHEMATICA als *MathSource* mit den Distributionen mitgeliefert wurden, unter den Namen

Maple Application Center	(`http://www.mapleapps.com`)
Maple Student Center	(`http://www.maple4students.com/`) bzw.
Mathematica Information Center	(`http://library.wolfram.com/infocenter`)

zu Internet-Portalen reorganisiert, über welche Nutzer relevante Materialien bereitstellen oder suchen und sich herunterladen können.

CAS als moderne Softwaresysteme

Der Umfang des bereits implementierten mathematischen Wissens macht es aus Effizienzgründen weiterhin erforderlich, über eine **flexible Konfigurierbarkeit** der Systeme entsprechend der jeweiligen Aufgabenstellung nachzudenken. In diesem Zusammenhang spielen insbesondere das Nachladen von Paketen und andere Konzepte des *dynamischen Ladens* eine wichtige Rolle. Dabei wird eine Startkonfiguration des Systems in den Hauptspeicher geladen, zu der je nach den Erfordernissen der auszuführenden Rechnungen zur Laufzeit weitere Codestücke hinzugeladen bzw. entfernt werden, womit der Hauptspeicherbedarf begrenzt werden kann.

Die (konsistente) Entwicklung und Betreuung solcher Systeme stellt sehr hohe Anforderungen an die verwendeten **Konzepte des Software-Designs.** Alle modernen Entwicklungen, begon-

nen von Modularisierungs- und Datenkapselungskonzepten, über dynamisches Laden und Client-Server-Modelle bis hin zu Modellen verteilten Rechnens finden deshalb in diesem Bereich sowohl ein sehr komplexes Target, in dem die Leistungsfähigkeit neuer Konzepte geprüft werden kann, als auch eine Fülle von Anregungen und Fragestellungen, wie sie in dieser Komplexität von kaum einem zweiten Gebiet in der Informatik aufgeworfen werden.

3.2 Der prinzipielle Aufbau eines Computeralgebrasystems

Nach dem Start des entsprechenden Systems meldet sich die **Interpreterschleife** und erwartet eine Eingabe, typischerweise einen in *linearisierter Form* einzugebenden symbolischen Ausdruck, der vom System ausgewertet wird. Das Ergebnis wird in einer stärker an mathematischer Notation orientierten *zweidimensionalen Ausgabe* angezeigt.

Neben derartigen Eingaben sowie einem `quit`-Befehl zum Verlassen der Interpreterschleife gibt es noch eine Reihe von Eingaben, die offensichtlich andere Reaktionen hervorrufen. Dies sind in MuPAD zum Beispiel

- Eingaben der Form `?topic` zur Aktivierung des Hilfesystems und
- Eingaben der Form `plot(...)`, worauf im Rahmen der X-Windows-Umgebung ein Grafik-Ausgabefenster geöffnet wird.

Offensichtlich werden hierbei andere als die unmittelbaren symbolischen Fähigkeiten des CAS herangezogen.

Die Interpreterschleife des CAS bringt also verschiedene Teile des Systems zusammen, wovon nur eines der unmittelbar symbolische Rechnungen ausführende Kern ist, den wir als den *Systemkern* bezeichnen wollen. Neben den beiden Komponenten *Hilfesystem* und *Grafik-Interface* sind dies noch die der Ein- und Ausgabe dienende *Nutzerschnittstelle* (front end) sowie ein *numerisches Interface*, das die numerische Auswertung symbolischer Ausdrücke übernimmt. Letzteres ist oftmals enger in den Systemkern intergriert. Wir wollen es trotzdem an dieser Stelle einordnen, da die entsprechende Funktionalität nicht direkt mit symbolischen Rechnungen zu tun hat.

Ehe wir uns Aufbau und Arbeitsweise des Systemkerns zuwenden, in dem die symbolischen Fähigkeiten des jeweiligen Systems konzentriert sind, wollen wir einen kurzen Blick auf die anderen Komponenten werfen.

Das äußere Erscheinungsbild des Systems wird in erster Linie durch die **Nutzerschnittstelle** bestimmt. Genereller Bestandteil derselben ist ein *Formatierungssystem* zur Herstellung zweidimensionaler Ausgaben, das sich stärker an der mathematischen Notation orientiert als die Systemeingaben. Man unterscheidet *Terminalversionen*, in denen die Ausgabe im Textmodus erfolgt und die neben Ein- und Ausgabe meist einen rudimentären Zeileneditor besitzen, und *fensterbasierte Versionen*, in denen die Ausgabe im Grafikmodus erfolgt.

Grafikbasierte Ausgabesysteme sind heute meist in der Lage, *integrierte Dokumente* zu produzieren, die Texte, Ergebnisse von Rechnungen und Grafiken verbinden und in gängigen Formaten (HTML, LaTeX) exportieren können. Eine besondere Vorreiterrolle spielen auf diesem Gebiet die Systeme MATHEMATICA und MAPLE mit dem Konzept des *Arbeitsblatts*. Hier treffen sich Entwicklungen aus dem Bereich des wissenschaftlichen Publizierens (mit Produkten wie *Scientific Word* von MacKichan), der Gestaltung interaktiver Oberflächen und Methoden des symbolischen Rechnens, womit sich zugleich vollkommen neue Horizonte in Richtung der (elektronischen) Publikation interaktiver mathematischer Texte eröffnen.

Hilfesysteme sind bei den einzelnen CAS sehr unterschiedlich entwickelt. Generell ist jedoch auch hier ein Trend hin zur Verwendung gängiger Techniken zum Entwurf von Hilfesystemen in Form hypertextbasierter Dokumente und entsprechender Analysewerkzeuge zu verspüren. Dabei

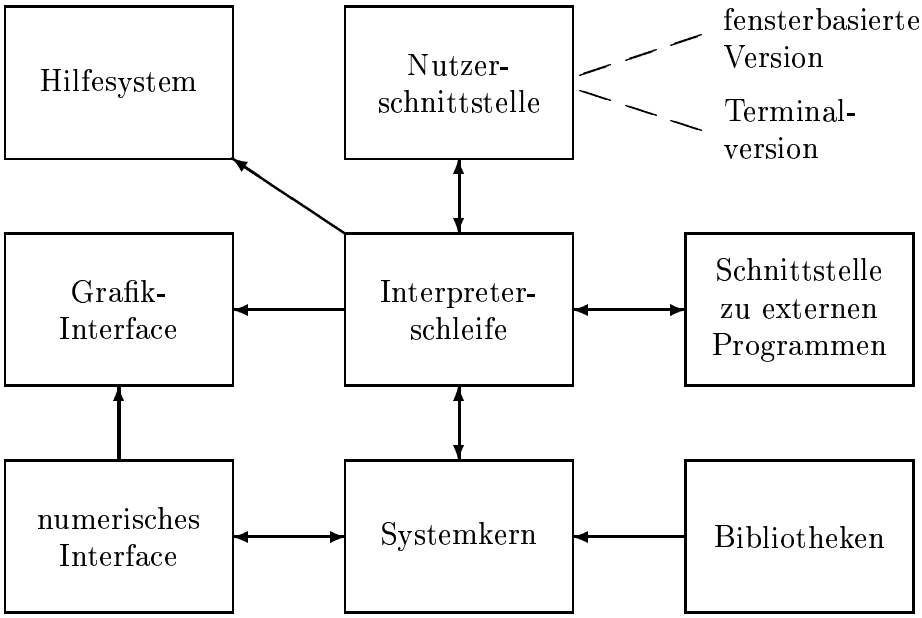


Bild 1: Prinzipieller Aufbau eines Computeralgebra-Systems

wird zunehmend, wie im letzten Kapitel bereits für das Grafik-Interface beschrieben, nicht nur auf entsprechende Ansätze, sondern auch auf bereits fertige Software-Komponenten zurückgegriffen. Hilfesysteme sind meist hierarchisch oder/und nach Schlagworten sortiert, so dass man relativ genaue Kenntnisse benötigt, wie konkrete Kommandos oder Funktionen heißen bzw. an welcher Stelle in der Hierarchie man relevante Informationen findet.

Obwohl es auch große und leistungsfähige Programmpakete zur Numerik gibt, treiben die CAS die Entwicklung ihres **numerischen Interface** in zwei Richtungen voran. Zum einen ist zu bedenken, dass ein CAS nur dann zu einem nützlichen Werkzeug, einem persönlichen digitalen Mathematik-Assistenten, in der Hand eines Wissenschaftlers oder Ingenieurs wird, wenn es die volle "compute power" bereitstellt, die von dieser Klientel im Alltag benötigt wird. Dazu gehören neben symbolischen Rechenfertigkeiten und Visualisierungstools auch die Möglichkeit, ohne weitergehenden Aufwand symbolische Ergebnisse numerisch auszuwerten. Deshalb hat jedes der großen CAS eigene Routinen für numerische Berechnungen etwa von Nullstellen oder bestimmten Integralen "für den Hausgebrauch". Dabei wird stark von den speziellen Möglichkeiten adaptiver Präzision einer bigfloat-Arithmetik Gebrauch gemacht, die auf der Basis der vorhandenen Langzahlarithmetik leicht implementiert werden kann. Diese Fähigkeiten, die etwa beim Bestimmen nahe beieinander liegender Nullstellen von Polynomen oder beim Berechnen numerischer Näherungswerte hoher Präzision eine Rolle spielen, sind stärker in den Systemkern integriert.

Andererseits ist eine wichtige, wenn nicht gar die wichtigste[1] Anwendung von Computeralgebra die Aufbereitung symbolischer Daten zu deren nachfolgender numerischer Weiterverarbeitung. Deshalb werden neben Codegeneratoren auch zunehmend **explizite Schnittstellen und Protokolle** entworfen, über welche die Programme mit externem Sachverstand kommunizieren können. Über solche Schnittstellen kann insbesondere mit vorhandenen Numerikbibliotheken kommuniziert werden. Allerdings kann eine solche Schnittstelle umfangreichere Funktionen erfüllen, etwa webba-

[1] Nach [13] werden moderne CAS zu 90 % zur Generierung effizienten Codes eingesetzt.

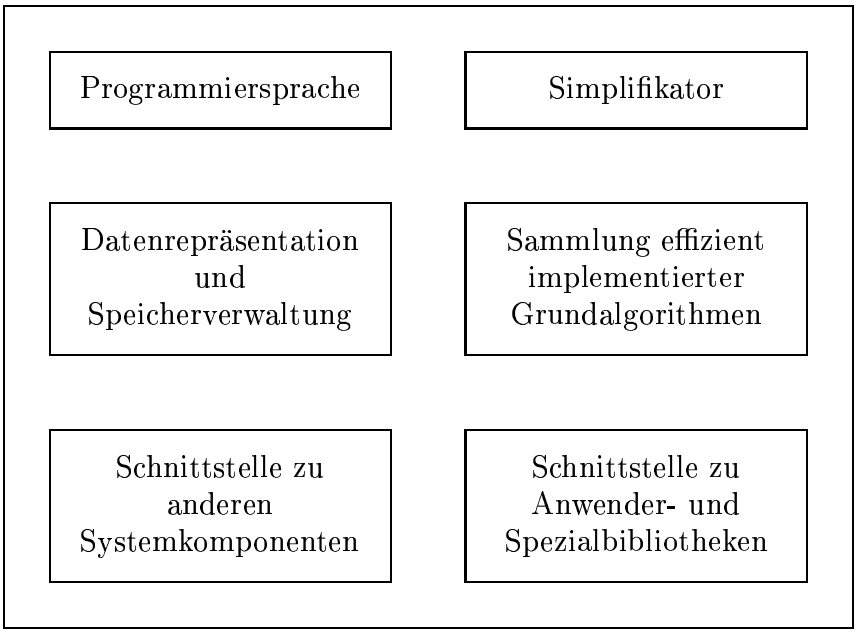


Bild 2: Komponenten des Systemkern-Designs

sierte Client-Kommunikation organisieren oder kooperative Prozesse mehrerer CAS oder mehrerer Prozesse eines CAS koordinieren.

Anforderungen an das Systemkerndesign

Betrachten wir nun die Anforderungen näher, die beim Design des Systemkerns zu berücksichtigen sind, in dem die uns in diesem Kurs interessierenden symbolischen Rechnungen letztendlich ausgeführt werden. Diese Anforderungen kann man grob folgenden sechs Komplexen zuordnen:

- Es ist ein Konzept für eine *Datenrepräsentation* zu entwickeln, das es erlaubt, heterogen strukturierte Daten, wie sie typischerweise im symbolischen Rechnen auftreten, mit der notwendigen Flexibilität, aber doch nach einheitlichen Gesichtspunkten zu verwalten und zu verarbeiten. Damit verbunden ist die Frage nach einer entsprechend leistungsfähigen *Speicherverwaltung*.

  Dabei ist zu berücksichtigen, dass symbolische Ausdrücke sehr unterschiedlicher und im Voraus nicht bekannter Größe auftreten, also als *dynamische Datentypen* mit einer ebensolchen *dynamischen* Speicherverwaltung anzulegen sind.

- Es wird eine *Programmiersprache* benötigt, mit der man den Ablauf der Rechnungen steuern kann. Bekanntlich reicht ein kleines Instrumentarium an Befehlskonstrukten aus, um die gängigen Programmablaufkonstrukte (Schleifen, Verzweigungen, Anweisungsverbünde) zu formulieren. Weiterhin sollte die Sprache Methoden des strukturierten Programmierens (Prozeduren und Funktionen, Modularisierung) unterstützen, vermehrt um Instrumente, die sich aus der Spezifik des symbolischen Rechnens ergeben.

- Es ist ein Konzept für den *Simplifikator* zu entwickeln, mit dessen Hilfe Ausdrücke gezielt in zueinander äquivalente Formen nach unterschiedlichen Gesichtspunkten umgeformt werden können. Als Minimalanforderung muss dieser Simplifikator wenigstens in der Lage sein, in gewissem Umfang die semantische Gleichwertigkeit syntaktisch unterschiedlicher Ausdrücke festzustellen.

Dabei handelt es sich meist um ein zweistufiges System, das aus einer effizient im Kern implementierten *Polynomarithmetik* besteht, die in der Lage ist, rationale Ausdrücke umzuformen, und einem (vom Nutzer erweiterbaren) *Simplifikationssystem*, das die Navigation in der transitiven Hülle der dem System bekannten elementaren Umformungsregeln gestattet.

- Es werden *effiziente Implementierungen grundlegender Algorithmen* (Langzahl- und Polynomarithmetik, Rechnen in modularen Bereichen, zahlentheoretische Algorithmen, Faktorisierung von Polynomen, Bigfloat-Arithmetik, Numerikroutinen, Differenzieren, Integrieren, Rechnen mit Reihen und Summen, Grenzwerte, Lösen von Gleichungssystemen, spezielle Funktionen ...) benötigt, die in verschiedenen Kontexten des symbolischen Rechnens immer wieder auftreten.

  Diese in Form von black-box-Wissen vorhandene Kompetenz ist die Kernkompetenz des Systems. Die Implementierung dieser Algorithmen baut wesentlich auf anderen Teilen des Designkonzepts auf, die damit darüber entscheiden, wie effektiv Implementierungen überhaupt sein können. Gewöhnlich sind Teile dieser Sammlung von Funktionen aus Gründen der Performance nicht in der Programmiersprache des jeweiligen Systems, sondern maschinennäher ausgeführt.

  Die Anzahl und die Komplexität der eingebauten Funktionen ist mit klassischen Programmiersprachen nicht vergleichbar.

- Es ist ein Konzept für das Zusammenwirken der verschiedenen *Spezial- und Anwenderbibliotheken* mit dem Systemkern zu entwickeln, um das in ihnen gespeicherte mathematisch-algorithmische Wissen zu aktivieren.

  Spezialbibliotheken sind Sammlungen von Implementierungen algorithmischer Verfahren aus mathematischen Teildisziplinen, die in der Sprache des jeweiligen Systems geschrieben sind und speziellere Kalküle (Tensorrechnung, Gruppentheorie) zur Verfügung stellen. Derartige Spezialbibliotheken werden oftmals von der jeweiligen mathematischen Community als Gemeineigentum entwickelt und gepflegt und von den CAS nur gesammelt und weitergegeben.

  Anwenderbibliotheken sind Sammlungen von Anwendungen mathematischer Methoden in anderen Wissenschaften, die auf den symbolischen Möglichkeiten des jeweiligen CAS aufsetzen. Solche Anwendungsbibliotheken, insbesondere im ingenieur-technischen und business-ökonomischen Bereich, werden oft kommerziell vertrieben.

- Schließlich ist ein Konzept für das *Zusammenwirken des Systemkern mit den anderen Systemkomponenten* zu entwickeln.

3.3 Klassische und symbolische Programmiersysteme

Das klassische Konzept einer imperativen Programmiersprache geht von der Vorstellung eines Programms als

> *schrittweise Transformation einer Menge von* **Eingabedaten** *in eine Menge von* **Ausgabedaten** *nach einem vorgegebenen* **Algorithmus**

aus ([4]).

Diese Transformation erfolgt durch *Abarbeiten einzelner Programmschritte,* in denen die Daten entsprechend den angegebenen Instruktionen verändert werden. Den Zustand der Gesamtheit der durch das Programm manipulierten Daten bezeichnet man auch als den *Programmstatus.*

Die Programmschritte werden gewöhnlich in *Anweisungen und Deklarationen* unterteilt, wobei Anweisungen den Programmstatus ändern, Deklarationen dagegen nicht, sondern lediglich Bedeutungen von Bezeichnern festlegen.

Anweisungen setzen sich aus elementaren Anweisungen zusammen, die durch entsprechende *Steuerstrukturen* miteinander verbunden sind. Elementare Anweisungen können *Zuweisungen* oder *Prozeduraufrufe* sein. Durch erstere wird direkt der Wert einer Variablen geändert, durch zweitere erfolgt die Änderung des Programmstatus als Seiteneffekt.

Bei **Deklarationen** unterscheidet man gewöhnlich zwischen Deklarationen von *Datentypen*, *Variablen*, *Funktionen* und *Prozeduren*. Jede solche Deklaration verbindet mit einem *Bezeichner* eine gewisse Bedeutung. Neben Variablenbezeichnern gibt es (z.B. in Pascal oder C) noch Bezeichner für Funktionen, Typen, Marken und Konstanten, die jedoch nur zur Compilezeit eine Bedeutung haben, zur Laufzeit dagegen maschinenabhängig durch konkrete Speicheradressen ersetzt sind.

Hinter demselben Bezeichner können sich in einem Programm allerdings unterschiedliche Objekte verbergen, was durch die Begriffe *Sichtbarkeit* und *Lebensdauer* beschrieben wird. Zur Übersetzung wird deshalb eine Tabelle, die **Symboltabelle** angelegt, in der die jeweils gültigen Kombinationen von Bezeichner und Bedeutung gegenübergestellt sind.

Obwohl diese Referenzen zur Laufzeit aufgelöst sind, verbleibt im Zusammenhang mit Variablen eine andere Referenz, die zwischen ihrer *Adresse* und dem dort hinterlegten *Wert*. Die Adresse spielt dabei die Rolle des Bezeichners, über den man das entsprechende Datum eindeutig identifizieren kann.

In einem Interpreter, der ja Syntaxanalyse und Abarbeitung kombiniert, existieren beide Referenzmechanismen nebeneinander, d.h. es gibt eine Symboltabelle, die z.B. einem Variablenbezeichner Adresse (Speicherplatz) und Typ (Speichergröße) zuordnet, und die Speicheradressen selbst, unter der man den jeweiligen Wert findet.

Dabei werden in der Phase der lexikalischen Analyse Ausdrücke geparst, in einem Parsebaum aufgespannt und dann mit Hilfe der in der Symboltabelle vorhandenen Referenzen vollständig ausgewertet. Der Parsebaum existiert damit auch in (klassischen) Interpretern nur in der Analysephase.

Das ist bei symbolischen Ausdrücken anders: Dort kann nur teilweise ausgewertet werden und der resultierende Ausdruck, sofern er noch symbolische Komponenten enthält, kann als Parsebaum nicht vollständig abgebaut werden.

klassisch: In der Phase der Syntaxanalyse wird ein Parsebaum des zu analysierenden Ausdrucks auf- und vollständig wieder abgebaut.
symbolisch: Als Form der Darstellung symbolischer Inhalte bleibt ein Parsebaum auch nach der Syntaxanalyse bestehen.
In der **Symboltabelle** werden zu den verschiedenen Bezeichnern nicht nur *programmrelevante Eigenschaften* gespeichert, sondern auch darüber hinaus gehende semantische Informationen über die mathematischen Eigenschaften des jeweiligen Bezeichners.

Ausdrücke

In klassischen Programmiersprachen ergibt sich der Wert, der einem Bezeichner zugeordnet wird, als Ergebnis eines *Ausdrucks* oder *Funktionsaufrufs*. Auch in Funktionsaufrufen selbst spielen (zulässige) Ausdrücke als Aufrufparameter eine wichtige Rolle, denn man kann sie statt Variablen an all den Stellen verwenden, an denen ein *call by value* erfolgt.

Der Begriff des Ausdrucks spielt in klassischen Programmiersprachen eine zentrale Rolle, wobei Funktionsaufrufe als spezielle Ausdrücke (*postfix-expression* in [18, R.5.2]) aufgefasst werden. Zulässige Ausdrücke werden rekursiv als Zeichenketten definiert, die nach bestimmten Regeln aus Konstanten, Variablenbezeichnern und Funktionsaufrufen sowie verschiedenen *Operationszeichen* zusammengesetzt sind. So sind etwa $a + b$ oder $b - c$ ebenso zulässige Ausdrücke wie $a + gcd(b, c)$, wenn a, b, c als `integer`-Variablen und `gcd` als zweistellige Funktion `gcd:(int,int)` $\mapsto$ `int` vereinbart wurden.

Solche zweistelligen Operatoren unterscheiden sich allerdings nur durch ihre spezielle Notation von zweistelligen Funktionen. Man bezeichnet sie als *Infix-Operatoren* im Gegensatz zu der gewöhnlichen Funktionsnotation als *Präfix-Operator*. Neben Infix-Operatoren spielen auch Postfix- (etwa $x!$), Roundfix- (etwa $|x|$) oder Mixfix-Notationen (etwa $f[2]$) eine Rolle.

Um den Wert von Ausdrücke mit Operatorsymbolen korrekt zu berechnen, müssen gewisse Vorrangregeln eingehalten werden, nach denen zunächst Teilausdrücke (etwa Produkte) zusammengefasst werden. Diese Hierarchie von Teilausdrücken spiegelt sich in einer analogen Hierarchie von Nichtterminalsymbolen der entsprechenden Grammatik wider ([18, R.5]: postfix-expression, unary-expression, cast-expression, pm-expression, multiplicative-expression, additive-expression, ...).

Zur konkreten Analyse solcher Ausdrücke wird vom Compiler ein Baum aufgebaut, dessen Ebenen der grammatischen Hierarchie entsprechen. So besteht ein *additive-expression* aus einer Summe von *multiplicative-expression*s, jeder *multiplicative-expression* aus einem Produkt von *pm-expression*s usw. Der Wert des Gesamtausdrucks ergibt sich durch rekursive Auswertung der Teilbäume und Ausführung von entsprechend von innen nach außen geschachtelten Funktionsaufrufen. So berechnet sich etwa der Ausdruck $a+b*c$ über einen Baum der Tiefe 2 als `+(a,*(b,c))`.

> Ausdrücke bestehen damit für einen klassischen Compiler (und Interpreter) nach der Auflösung einiger Diversitäten ausschließlich aus einer rekursiven Schachtelung von Funktionsaufrufen,

in die auf der untersten Ebene **Konstanten und Variablen** eingehen. Von Seiteneffekten einmal abgesehen (reference by name), ist der Unterschied zwischen Konstanten und Variablen unerheblich, da jeweils ausschließlich der *Wert* in die Berechnung des entsprechenden Funktionswerts eingeht.

Zur Analyse geschachtelter Funktionsaufrufe wird vom Compiler ebenfalls ein Wurzelbaum aufgebaut, dessen Wurzel dem Funktionssymbol der obersten Ebene entspricht und dessen Blätter selbst wieder Wurzelbäume sind, die den einzelnen Ausdrücken entsprechen, die als Argumente auftreten. Die Blätter dieses Baums sind einzelne Konstanten und Variablen. Aus diesem Baum wird durch den Compiler ein Programm generiert, das den Wert des entsprechenden Funktionsausdrucks bestimmt. Auch ein Interpreter einer klassischen Sprache generiert ein solches Programm, so dass sich hier der entsprechende Wurzelbaum ebenfalls nur als Zwischenprodukt in der Auswertungsphase eines Ausdrucks ergibt.

In symbolischen Rechnungen bleiben allerdings "Formeln" als Ergebnisse stehen, für die eine Darstellung durch ebensolche oder ähnliche Wurzelbäume sinnvoll ist. Solche Konzepte spielen eine zentrale Rolle bei der Darstellung symbolischer Ausdrücke.

> Die Prozesse, die in einem CAS zur *Laufzeit* ablaufen, haben damit viele Gemeinsamkeiten mit den Analyseprozessen, die in einem Compiler zur *Übersetzungszeit* stattfinden.

Datentypen und Polymorphie

Beim Auswerten von Ausdrücken spielt in klassischen Programmiersprachen *Polymorphie* eine wichtige Rolle, d.h. die Möglichkeit, Funktions- und insbesondere Operatorsymbolen unterschiedliche Bedeutung in Abhängigkeit von der Art der Argumente zu geben. Funktionen auf verschiedenen Strukturen mit gemeinsamen Eigenschaften durch dasselbe Symbol zu bezeichnen ist in der mathematischen Notation weit verbreitet und auch notwendig, um geeignete Abstraktionen überhaupt formulieren zu können.

Gleichwohl muss man diese Ambiguitäten in konkreten Implementierungen wieder auflösen, da natürlich z.B. zur Berechnung von $a + b$ für ganze und reelle Zahlen unterschiedliche Verfahren aufzurufen sind. Diese Unterscheidung vermögen Compiler klassischer Sprachen selbstständig zu

treffen. Sie sind darüberhinaus auch in der Lage, etwa bei der Addition einer ganzen und einer reellen Zahl selbständig eine passende Typkonversion auszuführen.

Beschränkt sich Polymorphie in klassischen Ansätzen noch auf von den Systemdesignern vorgesehene Typambiguitäten, so hat sie inzwischen mit dem Paradigma der Objektorientierung auch in allgemeinerer Form in die Informatik Einzug gehalten. Grundlage der Polymorphie ist dabei die Möglichkeit, Referenzen auf denselben Funktionsnamen an Hand der *Typinformationen*, die die Parameter des jeweiligen Funktionsaufrufes tragen, aufzulösen.

Typsysteme sind damit ein wichtiges Instrument in modernen Programmiersprachen und wären sicher auch für das symbolische Rechnen von Bedeutung. Warum fehlt den großen CAS, die wir (und andere Autoren) unter dem Begriff "2. Generation" zusammenfassen, trotzdem ein strenges Typkonzept bzw. ist ein solches nur in rudimentären Ansätzen vorhanden?

Untersuchen wir dazu am Beispiel des einfachen Ausdrucks $f := 2 * x + 3$, welche Ansprüche an ein Typsystem im symbolischen Rechnen zu stellen sind. f müsste ein passender Datentyp `Polynomial` zugeordnet werden, der (im Sinne eines strengen Typkonzepts) nicht nur ein Bezeichner ist, sondern für eine Signatur steht. Da in die Definition der Signaturen der Polynomoperationen die Signaturen der Operationen auf Koeffizienten und auf Termen eingehen, müsste ein qualifiziertes Typsystem wenigstens die unterschiedlichen Koeffizientenbereiche berücksichtigen. In einer Sprache wie Pascal oder bzw. C++ müssten wir also unterscheiden zwischen

```
class IntegerPolynomial {
    int coeff;
    Term exp; ...
}
```

und

```
class FloatPolynomial {
    float coeff;
    Term exp; ...
}
```

da diese unterschiedliche Additionen und Multiplikationen der Koeffizienten verwenden. Da jedoch viele andere Koeffizientenbereiche wie etwa rationale Zahlen, komplexe Zahlen und selbst Matrizen gelegentlich benötigt werden, ist es sinnvoller, Polynome als einen *parametrisierten Datentyp* anzulegen. `Polynomial(R)` konstruiert dann als Datentyp Polynome mit Koeffizienten aus dem Bereich R, für den selbst die entsprechenden Ringoperationen definiert sein müssen. Im Gegensatz zu Templates in C++,

```
<template R>
class Polynomial {
    R coeff;
    Term exp; ...
}
```

die für verschiedene Parameterwerte R vollständige Kopien der Implementierung des Datentyps anlegen, ist es allerdings sinnvoller, die Auflösung der Referenzen auf die Koeffizientenoperationen über den Aufrufparameter R vorzunehmen, der also ein *abstrakter Datentyp* sein müsste. Neben der Reduktion des entsprechenden Codes erlaubt ein solches Vorgehen auch, Polynomringe *dynamisch* zu erzeugen, d.h. als Koeffizienten Bereiche zu verwenden, an die zur Compilezeit des Polynomcodes nicht gedacht worden ist bzw. die vielleicht noch nicht einmal zur Verfügung standen. Solche Konzepte von *Typsprachen* sind aus dem funktionalen Programmieren gut bekannt, ebenso die Tatsache, dass das Wortproblem für einigermaßen aussagekräftige Typsprachen (insbesondere solche mit Selbstreferenzen) algorithmisch nicht entscheidbar ist.

Bereits diese einfache Überlegung zu Datentypkonzepten im symbolischen Rechnen führt uns also unvermittelt an die vorderste Front der Entwicklung programmiersprachlicher Instrumente.

Funktionale Programmiersprachen vermeiden (aus gutem Grund) die Einführung von Variablen als „Wertdepots“, denn insbesondere das *Problem der Inferenz von Datentypen* bringt erhebliche Schwierigkeiten mit sich. Um etwa den Ausdruck $f := 2 * x + 3$ als `Polynomial(Integer)` zu interpretieren, ist in einem ersten Schritt $2 * x$ zu berechnen, d.h. das Produkt aus `2:Integer` und `x:String` zu bilden. Dazu muss der gemeinsame Oberbereich erst gefunden werden, in den beide Datentypen transformiert werden können, damit die Multiplikation ausgeführt werden kann. Noch komplizierter wird es bei der Berechnung von $f + g$ mit $g = 2/3$. Für g ergibt sich bei entsprechender Analyse der Datentyp `Fraction(Integer)`, für f dagegen `Polynomial(Integer)`. Das Ergebnis könnte entweder die Einbettung `Integer` $\subset$ `Polynomial(Integer)` verwenden und damit den Typ `Fraction(Polynomial(Integer))` einer rationalen Funktion oder aber die Einbettung `Integer` $\subset$ `Fraction(Integer)` verwenden und damit den Typ `Polynomial(Fraction(Integer))` eines Polynoms mit rationalen Koeffizienten haben. Beiden Fällen ist gemein, dass der entsprechende Obertyp, in den eine Typkonversion erfolgen muss, aus einer Obertyprelation des *Parameters* zu inferieren ist. Programmiertechnisch sind damit aufwändige Restrukturierungen der inneren Darstellung von Objekten des jeweiligen Typs verbunden.

Diese kleine Liste von Fragestellungen mögen als Begründung dienen, weshalb CAS der zweiten Generation auf ein konsistentes Typsystem im Sinne der Theorie der Programmiersprachen weitestgehend verzichten und Typinformationen nur insoweit verwenden, wie sie sich aus der *syntaktischen Struktur* der entsprechenden Ausdrücke extrahieren lassen.

Allerdings gibt es langjährige Untersuchungen zu den genannten Fragen, die mit dem (heute Open-Source-) Projekt ALDOR (`http://www.aldor.org`) bis zu einer praktisch einsetzbaren Programmiersprache mit strengem Typkonzept für symbolische Rechnungen geführt worden sind (und weiter geführt werden).

Zur internen Darstellung von Ausdrücken in CAS

In einem (weitgehend) typlosen System ist es sinnvoll, eine geeignete Datenstruktur zu finden, mit der man Ausdrücke (also in unserem Verständnis geschachtelte Funktionsaufrufe) uniform darstellen kann. Anderenfalls hätte man für Polynome, Matrizen, Operatoren, Integrale, Reihen usw. jeweils eigene Datenstrukturen zu erfinden, was das Speichermanagement wesentlich erschweren würde. Die klassische Lösung des Ableitungsbaums mit dem Funktionsnamen in der Wurzel und den Argumenten als Söhne hat noch immer den Nachteil geringer Homogenität, da Knoten mit unterschiedlicher Anzahl von Söhnen unterschiedliche Speicherplatzanforderungen stellen.

Die Argumente von Funktionen mit unterschiedlicher Arität lassen sich allerdings in Form von Argumentlisten darstellen, wobei der typische Knoten einer solchen Liste aus zwei Referenzen besteht – dem Verweis auf das jeweilige Argument (car) und dem Verweis auf den Rest der Liste (cdr).

Sehen wir uns an, wie Ausdrücke in MAPLE, MATHEMATICA und REDUCE intern dargestellt werden. Die folgenden Prozeduren gestatten es jeweils, diese innere Struktur sichtbar zu machen.

MATHEMATICA:

Die Funktion `FullForm` gibt die interne Darstellung eines Ausdruck preis.

MAPLE:

```
level1:=u-> [op(0..nops(u),u)];
```

```
structure := proc(u)
  if type(u,atomic) then u else map(structure,level1(u)) fi
end;
```

`level1` extrahiert die oberste Ebene, `structure` stellt die gesamte rekursive Struktur der Ausdrücke dar.

MACSYMA:

```
level1(u):=map(lambda([v],part(u,v)),makelist(i,i,0,length(u)));

structure(u):= if atom(u) then u else map(structure,level1(u));
```

REDUCE:

```
procedure structure(u); lisp prettyprint u;
```

Wir betrachten folgende Beispiele:

$$(x+y)^5$$

MATHEMATICA:	`Power[Plus[x, y], 5]`
MAPLE und MACSYMA:	`[^,[+,x,y],5]`
REDUCE:	`(plus (expt x 5) (times 5 (expt x 4) y) (times 10 (expt x 3) (expt y 2)) (times 10 (expt x 2) (expt y 3)) (times 5 x (expt y 4)) (expt y 5))`

REDUCE überführt den Ausdruck sofort in eine expandierte Form. Dies können wir mit `expand` auch bei den anderen beiden Systemen erreichen. Die innere Struktur der expandierten Ausdrücke hat jeweils die folgende Gestalt:

MATHEMATICA:	`Plus[Power[x, 5], Times[5, Power[x, 4], y], Times[10, Power[x, 3], Power[y, 2]], Times[10, Power[x, 2], Power[y, 3]], Times[5, x, Power[y, 4]], Power[y, 5]]`
MAPLE und MACSYMA:	`[+, [^, x, 5], [*, 5, [^, x, 4], y], [*, 10, [^, x, 3], [^, y, 2]], [*, 10, [^, x, 2], [^, y, 3]], [*, 5, x, [^, y, 4]], [^, y, 5]]`

Ähnliche Gemeinsamkeiten findet man auch bei der Struktur anderer Ausdrücke.

Matrix in den verschiedenen Systemen definieren:

```
Mathematica
M={{1,2},{3,4}};

Maple
M:=matrix(2,2,[[1,2],[3,4]]);

Macsyma
M:matrix([1,2],[3,4]);

Reduce
M:= mat((1,2),(3,4));
```

$1/2$	MATHEMATICA:	`Rational[1, 2]`
	MAPLE:	`[fraction, 1, 2]`
	MACSYMA:	`[//, 1, 2]`
	REDUCE:	`(quotient 1 2)`
$1/x$	MATHEMATICA:	`Power[x, -1]`
	MAPLE:	`[^, x, -1]`
	MACSYMA:	`[//, 1, x]`
	REDUCE:	`(quotient 1 x)`
$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$	MATHEMATICA:	`List[List[1, 2], List[3, 4]]`
	MAPLE:	`[array, 1 .. 2, 1 .. 2, [(1, 1) = 1, (2, 1) = 3, (2, 2) = 4, (1, 2) = 2]]`
	MACSYMA:	`[MATRIX, ['[', 1, 2], ['[', 3, 4]]`
	REDUCE:	`(mat (1 2) (3 4))`
$\sin(x)^3 + \cos(x)^3$	MATHEMATICA:	`Plus[Power[Cos[x], 3], Power[Sin[x], 3]]`
	MAPLE:	`[+, [^, [sin, x], 3], [^, [cos, x], 3]]`
	MACSYMA:	`[+, [^, [SIN, x], 3], [^, [COS, x], 3]]`
	REDUCE:	`(plus (expt (cos x) 3) (expt (sin x) 3))`
Liste [a,b,c]	MATHEMATICA:	`List[a, b, c]`
	MAPLE:	`[list, a, b, c]`
	MACSYMA:	`['[', a, b, C]`
	REDUCE:	`(list a b c)`

Wir sehen, dass in allen betrachteten CAS die interne Darstellung der Argumente symbolischer Funktionsausdrücke in Listenform erfolgt, wobei die Argumente selbst wieder Funktionsausdrücke sein können, also der ganze Parsebaum in geschachtelter Listenstruktur abgespeichert wird.
Der zentrale Datentyp für ide interne Darstellung von Ausdrücken in CAS der 2. Generation ist also die geschachtelte Liste.

Bemerkenswert ist, dass sowohl in MAPLE als auch in REDUCE der Funktionsname keine Sonderrolle spielt, sondern als „nulltes“ Listenelement, als *Kopf*, gleichrangig mit den Argumenten in der Liste steht. Das gilt intern auch für MATHEMATICA, wo man auf die einzelnen Argumente eines Ausdruchs s mit `Part[s,i]` und auf das Kopfsymbol mit `Part[s,0]` zugreifen kann:

```
s=Sin[x+y]
s//FullForm

     Sin[Plus[x, y]]

s[[0]]

     Sin

s[[1]]

     x+y
```

Eine solche Darstellung erlaubt es, als Funktionsnamen nicht nur Bezeichner, sondern auch symbolische Ausdrücke zu verwenden. Ausdücke als Funktionsnamen entstehen im symbolischen Rechnen auf natürliche Weise: Betrachten wir etwa $f'(x)$ als Wert der Funktion f' an der Stelle x. Dabei ist f' ein symbolischer Ausdruck, der aus dem Funktionssymbol f durch Anwenden des Postfixoperators $'$ entsteht. Der entsprechende MAPLE-Ausdruck lautet `D(f)(x)` und ist semantisch gleichwertig zu `diff(f(x),x)`, der Ableitung des *Ausdrucks* `f(x)` nach der Variablen x. Der

erste Ausdruck wird intern als `[[D, f], x]` dargestellt (mit zusammengesetztem Kopfsymbol), der zweite als `[diff, [f, x], x]`.

Eine Darstellung von Funktionsaufrufen durch (geschachtelte) Listen, in denen das erste Listenelement den Funktionsnamen und die restlichen Elemente die Parameterliste darstellen, ist typisch für die Sprache LISP, die Urmutter aller modernen CAS. CAS verwenden das erste Listenelement darüber hinaus auch zur Kennzeichnung einfacher Datenstrukturen (Listen, Mengen, Matrizen) sowie zur Speicherung von Typangaben atomarer Daten, also für ein **syntaktisches Typsystem.**

Ein solches Datendesign erlaubt eine hochgradig homogene Datenrepräsentation, denn jedes Listenelement besteht aus einem Pointerpaar ("dotted pair"), wobei der erste Pointer auf das entsprechende Listenelement, der zweite auf die Restliste zeigt. Nur auf der Ebene der Blätter tritt eine überschaubare Zahl anderer (atomarer) Datenstrukturen wie ganze Zahlen, Floatzahlen oder Strings auf. Eine solche homogene Datenstruktur erlaubt trotz der sehr unterschiedlichen Größe der verschiedenen symbolischen Daten die effektive dynamische Allokation und Reallokation von Speicher, da einzelne Zellen jeweils dieselbe Größe haben.

Listen als abstrakte Datentypen werden dabei allerdings anders verstanden als in den meisten Grundkursen "Algorithmen und Datenstrukturen" (etwa [11] oder [23]). Die dort eingeführte *destruktiv veränderbare Listenstruktur* mit den zentralen Operationen `Einfügen` und `Entfernen` ist für unsere Zwecke ungeeignet, da Ausdrücke gemeinsame Teilstrukturen besitzen können und deshalb ein destruktives Listenmanagement zu unüberschaubaren Seiteneffekten führen würde. Stattdessen werden Listen als *rekursiv konstruierbare Objekte* betrachtet mit Zugriffsoperatoren `first` (erstes Listenelement) und `rest` (Restliste) sowie dem Konstruktor `cons`, der ein neues Pointerpaar erstellt, mit dem (ein Pointer auf) ein Listenelement und (ein Pointer auf) eine Restliste zu einer neuen Liste zusammengefügt werden. Dieses für die Sprache LISP typische Vorgehen erlaubt es, auf aufwändiges Duplizieren von Teilausdrücken zugunsten des Duplizierens von Pointern zu verzichten, indem beim Kopieren einer Liste die Referenzen übernommen, die Links aber kopiert werden. Diese Sprachelemente prädestinieren einen funktionalen und rekursiven Umgang mit Listen, der deshalb in LISP und damit in CAS weit verbreitet ist. Für einen aus einer imperativen Welt kommenden Nutzer ist das gewöhnungsbedürftig.

Datendarstellung in MuPAD

In CAS der dritten Generation liegt zwischen der internen Datendarstellung und dem Nutzerinterface noch eine weitere Schicht der Datenorganisation. Im Fall von MuPAD wird ein objektorientierter Ansatz verfolgt, der Ausdrücke verschiedenen Grundbereichen (Domains) zuordnet, deren interner Aufbau jeweils spezifisch organisiert ist. Die Funktion `domtype` ermittelt den Typ des jeweiligen Ausdrucks, wobei klassische Ausdrücke meist vom Typ `DOM_EXPR` sind, die intern ebenso dargestellt werden wie Ausdrücke in anderen CAS (siehe oben). Ausdrücke anderer Typen haben nicht unbedingt ein Feld `op(u,0)`. Mit gewissen Einschränkungen können Sie die folgenden Funktionen zur Strukturbestimmung verwenden:

```
level1:=proc(u) begin
  if domtype(u)=DOM_EXPR then [op(u,0..nops(u))] else u end_if;
end_proc;

structure:=proc(u) local i;
  begin
  if args(0)>1 then structure(args(i))$i=1..args(0)
  elif type(u) in {DOM_LIST,DOM_SET} then
       [structure(extop(u,i))$ i = 1 .. extnops(u)]
  elif extop(u,0)=FAIL then u
```

```
      else [structure(extop(u,i))$ i = 0 .. extnops(u)]
      end_if
    end_proc;
```

Beispiele:

```
level1(a+b*c);
                              [_plus, a, b*c]
structure(a+b*c);
                           [_plus, a, [_mult, b, c]]
structure(sin(x));
                                   [sin, x]
structure(sin(2));
                                   [sin, 2]
M:=Dom::Matrix();
A:=M([[1,2],[3,4]]);
structure(A);

  [Dom::Matrix(), 2, 2, [2, [_range, 1, 2], [_range, 1, 2], 1, 2, 3, 4]]

print(NoNL,A);

  array(1..2, 1..2, (1,1) = 1, (1,2) = 2, (2,1) = 3, (2,2) = 4)
```

`level1` bestimmt die Struktur der obersten Ebene, wenn es sich um einen Ausdruck vom Typ `DOM_EXPR` handelt. Dies entspricht in etwa der Wirkung der Funktion `prog::exprtree`, die ebenfalls nur die Struktur von Ausdrücken des Typs `DOM_EXPR` expandiert.

`structure` versucht, die interne Struktur von Ausdrücken auch anderer Typen genauer zu bestimmen, wobei berücksichtigt wird, dass in einem Slot auch ganze Ausdruckssequenzen stehen können. `print(NoNL,A)` gibt den Ausdruck A mit `PRETTYPRINT:=FALSE` aus, was ebenfalls einen gewissen Einblick in die innere Struktur geben kann. Für Funktionen steht auch die Ausgabe mit `expose` zur Verfügung.

3.4 Das Variablenkonzept des symbolischen Rechnens

Wir hatten bereits gesehen, dass die Analyse symbolischer Ausdrücke *zur Laufzeit*, die ja im Mittelpunkt des Interpreters eines CAS steht, weniger Ähnlichkeiten zum Laufzeitverhalten einer klassischen Programmiersprache hat als vielmehr zu den Techniken, die Compiler oder Interpreter derselben *zur Übersetzungszeit* verwenden.

Das trifft insbesondere auf das Variablenkonzept zu, in dem statt der klassischen Bestandteile *Bezeichner, Wert, Speicherbereich* und *Datentyp*, von denen in symbolischen Umgebungen sowieso nur die ersten beiden eine Bedeutung besitzen, neben dem Bezeichner *verschiedenartige* symbolische Informationen eine Rolle spielen, die wir als **Eigenschaften** dieses Bezeichners zusammenfassen wollen.

Die entscheidende Besonderheit in der Verwendung von Variablen in einem symbolischen Kontext besteht aber darin, dass sich **Namensraum und Wertebereich überlappen.** In der Tat ist in einem Ausdruck wie $x^2 + y$ nicht klar, ob der Bezeichner x hier als *Symbolvariable* für sich selbst steht oder als *Wertvariable* Container eines anderen Werts ist. Mehr noch kann sich die Bedeutung je nach Kontext unterscheiden. So wird etwa in einem Aufruf

```
u:=int(1/(sin(x)*cos(x)-1),x);
```

x symbolisch gemeint sein, selbst wenn der Bezeichner bereits einen Wert besitzt.

Sehen wir uns diese Besonderheit des Variablenkonzepts zunächst in einigen MAPLE-Beispielen an.

```
p:=9*x^3-37*x^2+47*x-19;
```

$$9\,x^3 - 37\,x^2 + 47\,x - 19$$

```
x:=2;
p;
```

$$-1$$

Das ist das erwartete Verhalten. Aus der *Symbolvariablen* x wurde durch die Wertzuweisung eine *Wertvariable*, deren *Wert* in die nachfolgende Auswertung von p eingeht.

Versuchen wir nun, diese Umwandlung rückgängig zu machen, etwa mit dem Versuch

```
x:=unknown;
p;
```

$$9\,unknown^3 - 37\,unknown^2 + 47\,unknown - 19$$

x ist noch immer eine Wertvariable, also Container eines nunmehr allerdings symbolischen Wertes, und kein Symbol.

```
x:=x;
```

$$x := unknown$$

Auch diese "Verzweiflungstat" half nicht weiter, denn bei einer Zuweisung wird die rechte Seite ausgewertet und deren *Wert* der linken Seite zugewiesen.

Um x als Symbolvariable zu verwenden, müssen wir diese Auswertung verhindern, was in MAPLE wie folgt erreicht wird:

```
x:='x'; p;
```

$$x := x$$

$$9\,x^3 - 37\,x^2 + 47\,x - 19$$

Allerdings wurde hierbei nicht dem Bezeichner x das Symbol x als Wert zugewiesen, wie die Syntax des Befehls suggeriert, sondern der Wert des Bezeichners x gelöscht und dieser damit in seinen (ursprünglichen) Symbolvariablenzustand zurückversetzt. In den anderen CAS wird dieser Sachverhalt durch die Syntax des entsprechenden Befehls deutlicher ausgedrückt.

MACSYMA	kill(x)
MAPLE	x:='x'
MATHEMATICA	Clear[x]
MUPAD	delete x
REDUCE	clear x

Tabelle 3: Bezeichner in Symbolvariable zurückverwandeln

Wir sehen, dass ein Bezeichner in einem Ausdruck bei seiner ersten Verwendung automatisch als Symbolvariable interpretiert wird. Durch eine Wertzuweisung verwandelt er sich in eine (globale) Wertvariable. Die *Deklaration* einer solchen globalen Variablen ist nicht notwendig, da ja kein Typ vereinbart werden muss.

Bezeichner werden in klassischen Compilern und Interpretern in einer *Symboltabelle* erfasst, um sie an allen Stellen, an denen sie im Quellcode auftreten, in einheitlicher Weise zu verarbeiten. Für jeden Bezeichner existiert **genau** ein Eintrag in der Tabelle, unter dem auch weitere Informationen über *Eigenschaften*, wie Datentyp, Speicheradresse, Auftreten in lokalen Kontexten usw. vermerkt sind.

Da in einem CAS jede in irgendeinem Ausdruck auftretende Symbolvariable später als Wertvariable verwendet werden kann, muss dieser *Mechanismus der eindeutigen Referenz* in einem solchen Kontext auf **alle** auftretenden Symbole ausgedehnt werden. Dementsprechend wird bei der Analyse der einzelnen Eingaben jeder String, der einen Bezeichner darstellen könnte, daraufhin geprüft, ob er bereits an anderer Stelle aufgetreten ist und in diesem Fall eine Referenz an die entsprechende Stelle der Symboltabelle eingetragen. Andernfalls ist die Symboltabelle um einen neuen Eintrag zu erweitern.

In einigen CAS kann man sich diese Symboltabelle ganz oder teilweise anzeigen lassen. MAPLE verfügt sogar über zwei solche Funktionen. `unames()` listet alle Symbolvariablen (unassigned names) auf, `anames()` alle Wertvariablen[2].

Betrachten wir, wie sich diese beiden Listen im Zuge von Wertzuweisungen ändern. Mit den folgenden Anweisungen kann man MAPLE zurücksetzen und sich damit den ursprünglichen Zustand beider Sequenzen ansehen:

```
restart: [unames()]; [anames()];
```

Eine ganze Reihe von Informationen, die beim Start von MAPLE geladen werden, liegen als Strings vor und sind deshalb nach dem beschriebenen Mechanismus in diese Tabelle aufgenommen worden. Die zweite Menge ist dagegen noch leer.

Betrachten wir einen Teil aus diesem Raum, die Namen der Länge 1, und beobachten, wie sich dieser Namensraum durch die Verwendung von Variablen verändert:

```
[anames()]; select(s->length(s)=1,[unames()]);
```

```
                                    []

                          [!, ., <, =, I, O, ^, s, x, y]
```

```
u; [anames()]; select(s->length(s)=1,[unames()]);
```

```
                                    u

                                    []

                         [!, ., <, =, I, O, ^, s, u, x, y]
```

```
u:=1; [anames()]; select(s->length(s)=1,[unames()]);
```

```
                                  u := 1

                                  [u]

                          [!, ., <, =, I, O, ^, s, x, y]
```

[2]Die in der Dokumentation gegebene Spezifikation ist allerdings nicht ganz korrekt, da nicht alle Symbole, die einen Wert haben, angezeigt werden, wie man sich leicht überzeugt, wenn man sich etwa mit `anames(integer)` alle Symbole mit Integer-Werten anzeigen lässt. Außerdem ist das Ergebnis des Aufrufs in Maple 8 unter Linux und Windows unterschiedlich.

```
u:='u';[anames()]; select(s->length(s)=1,[unames()]);
```

```
                                u := u

                                  []

                          [!, ., <, =, I, O, ^, s, u, x, y]
```

In der letzten Zeile ist u aus der Liste der Wertvariablen wieder verschwunden, d.h. die angegebene Zuweisung "löscht" tatsächlich den Wert von u.

Zusammenfassung

In CAS treten Bezeichner als Symbolvariablen und als Wertvariablen auf. Ein Bezeichner wird so lange als Symbolvariable behandelt, bis ihm ein Wert zugewiesen wird. Hat ein Bezeichner einen Wert, so ist zwischen dem Bezeichner als Wertvariable und dem Bezeichner als Symbol zu unterscheiden. Bezeichner können durch entsprechende Befehle in ihren ursprünglichen Symbolzustand zurück versetzt werden. Bezeichner werden in einer Symboltabelle zusammengefasst, um ihre eindeutige Referenzierbarkeit zu sichern.

Mit dem MATHEMATICA-Befehl `Remove` können auch Einträge aus dieser Symboltabelle entfernt werden. Eine solche Möglichkeit ist aber mit Vorsicht zu verwenden, da es sein kann, dass später auf das bereits gelöschte Symbol weitere Referenzen gesetzt werden. Stattdessen sollten Sie nur mit `Clear` bzw. `ClearAll` arbeiten.

Dieses einheitliche Management der Bezeichner führt dazu, dass auch zwischen Variablenbezeichnern und Funktionsbezeichnern nicht unterschieden wird. Wir hatten beim Auflisten der dem System bekannten Symbole bereits an verschiedenen Stellen Funktionsbezeichner in trauter Eintracht neben Variablenbezeichnern stehen sehen. Dies ist auch deshalb erforderlich, weil in einen Ausdruck wie `D(f)`, die Ableitung der einstelligen Funktion f, Funktionssymbole auf dieselbe Weise eingehen wie Variablensymbole in den Ausdruck `sin(x)`.

CAS unterscheiden nicht zwischen Variablen- und Funktionsbezeichnern.

Da andererseits Funktionsdefinitionen ebenfalls symbolischer Natur sind und "nur" einer entsprechenden Interpretation bedürfen, verwenden einige Systeme wie etwa MAPLE sogar das Zuweisungssymbol, um Funktionsdefinitionen mit einem Bezeichner zu verbinden.

Eine solche einheitliche Behandlung interner und externer Bezeichner erlaubt es auch, Systemvariablen und selbst -funktionen zur Laufzeit zu überschreiben oder neue Funktionen zu erzeugen und (selbst in den compilierten Teil) einzubinden. Da dies zu unvorhersagbarem Systemverhalten führen kann, sind die meisten internen Funktionen allerdings vor Überschreiben geschützt. Hier einige Beispiele in MAPLE:

```
> Pi:=3.14;

Error, attempting to assign to `Pi` which is protected

> unprotect(Pi);
> Pi:=1;
                                  Pi := 1

> eval(arctan(1));
```

```
                                  1/4

> gcd:=2;

Error, attempting to assign to `gcd` which is protected

>  unprotect(gcd);
> gcd:=5;
                                 gcd := 5

> gcd(2,3);
                                    5

> gcd(12,13);
                                    5
```

Wir sehen, dass es MAPLE nach Entfernen des Überschreibschutzes selbst erlaubt, Funktionsnamen als Variablennamen zu verwenden. Die meisten CAS unterscheiden jedoch zwischen Wert- und Funktionsdefinitionen und ordnen sie als *unterschiedliche* Eigenschaften dem jeweiligen Bezeichner zu.

Auswerten von Ausdrücken

Die Tatsache, dass der Wert von Bezeichnern symbolischer Natur sein kann, in den Bezeichner eingehen, die ihrerseits einen Wert besitzen können, führt zu einer weiteren Besonderheit von CAS. Betrachten wir die folgenden Zuweisungen in MAPLE

```
a:=b+1; b:=c+3; c:=3;
```

und sehen uns an, was als Wert des Symbols a berechnet wird:

```
a;
  7
```

Es ist also die gesamte Evaluationskette durchlaufen worden: In den Wert $b+1$ von a geht das Symbol b ein, das seinerseits als Wert den Ausdruck $c+3$ hat, in den das Symbol c eingeht, welches den Wert 3 besitzt. Denkbar wäre auch eine eingeschränkte Evaluationstiefe, etwa nur Tiefe 1. In MAPLE kann man diese Tiefe mit speziellen Kommandos variieren:

```
seq(eval(a,i),i=1..6);
       b + 1, c + 4, 7, 7, 7, 7
```

Ändern wir den Wert eines Symbols in dieser Evaluationskette, so kann sich auch der Wert von a bei erneuter Evaluation ändern.

```
b:=7*d+3;
                                 b := 7 d + 3
a;
                                   7 d + 4
seq(eval(a,i),i=1..3);
       b + 1, 7 d + 4, 7 d + 4
```

Die bisher verwendeten Bezeichner auf den rechten Seiten waren Symbolvariable. Werden Wertvariablen verwendet, so müssen wir unterscheiden, ob wir das Symbol oder dessen Wert meinen.

```
u:=3;a:=u;b:='u';
                                        u := 3

                                        a := 3

                                        b := u

a;b;
                                           3

                                           3
```

An dieser Stelle ist der Unterschied noch nicht zu erkennen. Betrachten wir jedoch

```
u:=5;a;b;
                                        u := 5

                                           3

                                           5
```

so erkennen wir, dass sich Änderungen des Werts von u auf b auswirken, auf a dagegen nicht.

```
seq(eval(a,i), i=1..5);
                                    3, 3, 3, 3, 3

seq(eval(b,i), i=1..5);
                                    u, 3, 3, 3, 3
```

Geht in einen Ausdruck ein Bezeichner x als *Wertvariable* ein, so wird der Bezeichner *ausgewertet*, wird er dagegen nur als *Symbolvariable* verwendet, so wird der Bezeichner *nicht ausgewertet*. Oft spricht man in diesem Zusammenhang von *früher Auswertung* und *später Auswertung*. Diese Terminologie ist allerdings etwas irreführend, denn beide Ergebnisse werden natürlich bei einem späteren Aufruf, wenn sie als Teil in einen auszuwertenden Ausdruck eingehen, auch selbst ausgewertet. Korrekt müsste man also von *früher Auswertung* und *früher Nichtauswertung* sprechen.

CAS benötigen Mechanismen, um Bezeichner in beiden Bedeutungen, Wertvariable und Symbolvariable, einzusetzen. Dafür gibt es zwei gängige Herangehensweisen:

- Auf Bezeichner, die häufiger in ihrer symbolischen Gestalt benötigt werden, werden keine globalen Wertzuweisungen angewendet. Muss einem solchen Bezeichner in einem lokalen Kontext ein Wert zugewiesen werden, so geschieht dies unter Verwendung des **Substitutionsoperators**. Diese Wertzuweisung ist nur in dem entsprechenden Ausdruck wirksam. In einigen CAS wird dabei keine vollständige Evaluation oder gar Simplifikation des Ergebnisses ausgeführt.
- Für Wertvariablen gibt es einen Mechanismus, diesen Bezeichner vor der Auswertung zu bewahren und so als Symbolvariable zu betrachten. Dies geschieht entweder wie in MAPLE, MACSYMA oder MUPAD (und LISP) durch eine spezielle *Hold*-Funktion oder wie in AXIOM oder REDUCE durch zwei Zuweisungsoperatoren, wovon einer die in die rechte Seite eingehenden Bezeichner als Wertvariable betrachtet und diese *vor der Zuweisung auswertet*, der andere dagegen diese Bezeichner als Symbolvariable betrachtet und *vor der Zuweisung nicht auswertet*. MATHEMATICA verfügt sogar über beide Mechanismen.

Diese Besonderheiten sind in einer klassischen Programmiersprache, in der Namensraum und Wertebereich getrennt sind, unbekannt.

System	Substitutions-operator	Zuweisung mit Auswertung	Zuweisung ohne Auswertung	Hold-Operator
AXIOM	subst(f(x),x=a)	x:=a	x==a	–
MACSYMA	subst(x=a,f(x))	x:a	–	'x
MAPLE	subs(x=a,f(x))	x:=a	–	'x'
MATHEMATICA	f(x) /.x↦a	x=a	x:=a	Hold[x]
MUPAD	subs(f(x),x=a)	x:=a	–	hold(x)
REDUCE	subst(x=a,f(x))	x:=a	let x=a	–

Tabelle 4: Substitutions- und Zuweisungsoperatoren der verschiedenen CAS

Bei dem bisherigen Evaluierungsverfahren maximaler Tiefe kann es eintreten, dass ein zu evaluierendes Symbol nach endlich vielen Evaluationsschritten selbst wieder in dem entstehenden Ausdruck auftritt und damit der Evaluationsprozess stets von Neuem durchlaufen wird wie in folgendem Beispiel (MATHEMATICA):

```
x:=y+1;
y:=x+1;
x

$RecursionLimit::reclim: Recursion depth of 256 exceeded.

255 + Hold[x + 1]
```

Solche rekursiven Verkettungen können leicht eintreten, wenn man die Konsequenzen der getroffenen Wertzuweisungen nicht genau überblickt. Auch aus diesem Grund spielen lokale Wertzuweisungen, die nur innerhalb eines einzigen Aufrufs Gültigkeit haben, eine wichtige Rolle. MATHEMATICAverwendet die Systemvariablen `$RecursionLimit` und `$IterationLimit` zur Steuerung des Verhaltens in diesem Fall. Die rekursive Auswertung von Symbolen wird nach Erreichen der vorgegebenen Tiefe abgebrochen und der nicht weiter ausgewertete symbolische Ausdruck in eine `Hold`-Anweisung eingeschlossen, um ihn auch zukünftig vor (automatischer) Auswertung zu schützen.

MUPAD erlaubt rekursive Zuweisungen, bricht aber nach Erreichen einer durch die Systemvariable `MAXLEVEL` vorgegebenen Tiefe mit einer Fehlermeldung ab.

MAPLE (seit Version 8) sowie REDUCE lassen eine solche rekursive Zuweisung erst gar nicht zu, wenn die zu belegende Variable im zuzuweisenden Wert vorkommt.

```
> x:=x+1;
Error, recursive assignment
```

Allerdings kann das leicht „versteckt“ und durch eine scheinbar harmlose Zuweisung *ohne* Auswertung eine solche unendliche Evaluationskette geschlossen werden:

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

MAPLE 8 hängt sich an dieser Stelle auf, während REDUCE mit einer „harten“ Fehlermeldung abbricht:

```
x := y + 1
let y=x;

x;
***** Binding stack overflow, restarting...
```

MACSYMA verwendet standardmäßig nur Auswertungen der Tiefe 1, d.h. betrachtet nach dem ersten Evaluationsschritt auftretende Bezeichner als Symbole, wodurch dieses Problem der rekursiven Wertzuweisung ebenfalls vermieden wird[3]. Allerdings unterscheidet sich damit das Auswertungsverhalten von dem der anderen CAS. Das Auswertungsverhalten der anderen CAS kann (in erster Näherung) mit der Funktion `ev(..,infeval)` erreicht werden. Allerdings können mit der Evaluierungsfunktion `ev` in MACSYMA komplexere Effekte erzielt werden.

Zuweisung	MACSYMA	(z.B.) MAPLE
a:=b+1	b+1	b+1
b:=c+1	c+1	c+1
c:=d+1	d+1	d+1
a	b+1	d+3
a:=b+1	c+2	d+3

Tabelle 5: Unterschiedliches Auswertungsverhalten von MACSYMA und MAPLE

3.5 Funktionen

Funktionen bezeichnen im mathematischen Sprachgebrauch *Abbildungen* $f : X \to Y$ von einem Definitionsbereich X in einen Wertevorrat Y. In diesem Verständnis steht der ganzheitliche Aspekt stärker im Mittelpunkt, der es erlaubt, über Klassen von Funktionen und deren Eigenschaften zu sprechen sowie abgeleitete Objekte wie Stammfunktionen und Ableitungen zu betrachten.

In klassischen Programmiersprachen steht dagegen der konstruktive Aspekt der Funktionsbegriffs im Vordergrund, d.h. der *Algorithmus*, nach welchem die Abbildungsvorschrift f jeweils realisiert werden kann. Eine Funktion ist in diesem Sinne eine (beschreibungs-)endliche Berechnungsvorschrift, die man auf Elemente $x \in X$ anwenden kann, um entsprechende Elemente $f(x) \in Y$ zu produzieren. Wir unterscheiden dabei Funktionsdefinitionen und Funktionsaufrufe.

Betrachten wir den Unterschied am Beispiel der Wurzelfunktion `sqrt`. Die Mathematik gibt sich durchaus mit dem Ausdruck `sqrt(2)` zufrieden und sieht darin sogar eine exaktere Antwort als in einem dezimalen Näherungswert. Sie hat dafür extra die symbolische Notation $\sqrt{2}$ erfunden. In einer klassischen Programmiersprache wird dagegen beim Aufruf `sqrt(2)` ein Näherungswert für $\sqrt{2}$ berechnet, etwa mit dem Newtonverfahren. Beim Aufruf `sqrt(x)` würde sogar mit einer Fehlermeldung abgebrochen, da dieses Verfahren für symbolische Eingaben nicht funktioniert. Mathematiker würden dagegen, wenigstens für $x \geq 0$, als Antwort `sqrt(x)` $= \sqrt{x}$ gelten lassen als "diejenige positive reelle Zahl, deren Quadrat gleich x ist".

Symbolische Systeme kennen deshalb auch Funktionssymbole, d.h. "Funktionen ohne Funktionsdefinition". Dies ist vollkommen analog zum Wechselverhältnis zwischen Wertvariablen und Symbolvariablen, da wir letztere als "Variablen ohne Wert" kennengelernt haben.

Die Auswertung eines Funktionsaufrufs folgt dem klassischen Schema, wobei als Wert eines Aufrufparameters ein symbolischer Ausdruck im weiter oben definierten Sinne auftritt. Existiert keine Funktionsdefinition, so wird allerdings nicht mit einer Fehlermeldung abgebrochen, sondern aus den Werten der formalen Parameter und dem Funktionssymbol als "Kopf" ein neuer Ausdruck gebildet. Einen solchen symbolischen Ausdruck, dessen Kopf ein Funktionssymbol ist, hatten wir bereits weiter oben als Funktionsausdruck bezeichnet. In unserem Sprachgebrauch ist (fast) jeder symbolische Ausdruck ein solcher Funktionsausdruck.

Es kann sogar der Fall eintreten, dass eine Funktion nur partiell (im informatischen Sinne) definiert ist, d.h. für spezielle Argumente ein Funktionsaufruf stattfindet, für andere dagegen ein

[3]Dasselbe gilt etwa in MAPLE oder MUPAD auch für lokale Bezeichner innerhalb von Prozedurrümpfen.

Funktionsausdruck gebildet wird. So wird etwa in den folgenden MuPAD-Konstrukten das Funktionssymbol `sin` zur Konstruktion symbolischer Ausdrücke verwendet, die für Sinus-Funktionswerte stehen, die nicht weiter ausgewertet werden können.

```
sin(x); sin(2);
```

$$\sin(x)$$

$$\sin(2)$$

Im Gegensatz dazu sind

```
sin(PI/4);
```

$$1/2\,\sqrt{2}$$

```
sin(2.55);
```

$$0.5576837174$$

klassischen Funktionsaufrufen ähnlich, die eine Funktionsdefinition von *sin* verwenden.

Ähnlich den Bezeichnern für Variablen können auch neue Funktionsbezeichner eingeführt werden, von denen zunächst nichts bekannt ist und die damit als Funktionssymbole behandelt werden.

```
u:=f(x)+2*x+1;
```

$$f(x) + 2\,x + 1$$

```
subs(u,x=2);
```

$$f(2) + 5$$

Mit dem neuen Funktionssymbol f wurden zwei Ausdrücke `f(x)` und `f(2)` konstruiert.

Transformationen

Eine besondere Art von Funktionen sind die MuPAD-Funktionen `expand, collect, rewrite, normal`, die symbolische Ausdrücke *transformieren*, d.h. Funktionsaufrufe darstellen, deren Wirkung auf einen Umbau der als Parameter übergebenen Ausdrücke ausgerichtet ist.

> Solche *Transformationen* ersetzen gewisse Kombinationen von Funktionssymbolen und evtl. speziellen Funktionsargumenten durch andere, semantisch gleichwertige Kombinationen.

Transformationen (auch Simplifikationen genannt) sind eines der zentralen Designelemente von CAS, da auf diesem Wege syntaktisch verschiedene, aber semantische gleichwertige Ausdrücke produziert werden können. Sie werden bis zu einem gewissen Grad automatisch ausgeführt. So wurde etwa bei der Berechnung von `sin(PI/4)` das Zusammentreffen der Symbole bzw. symbolischen Ausdrücke `sin` und `PI/4` „von selbst“ erkannt. Auch Vereinfachungen wie etwa `sqrt(2)^2` zu 2 werden automatisch vorgenommen. Da Transformationen in einem breiten und in seiner Gesamtheit widersprüchlichen Spektrum möglich sind, gibt es für einzelne Transformationsaufgaben spezielle *Transformationsfunktionen*, die aus dem Gesamtspektrum eine (konsistente) Teilmenge von Transformationen auf einen als Parameter übergebenen Ausdruck *lokal* anwenden. Transformationsfunktionen spielen damit für Transformationen eine ähnliche Rolle wie der Substitutionsoperator für lokale Wertzuweisungen.

Betrachten wir die Wirkung eine solchen Transformationsfunktion, hier der MuPAD-Funktion `expand`, näher.

```
expand((x+1)*(x+2));
```

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

Dieser Aufruf von `expand` verwandelt ein Produkt von zwei Summen in eine Summe nach dem Distributivgesetz.

```
expand(sin(x+y));
```

$$\sin(x)\cos(y) + \cos(x)\sin(y)$$

Dieser Aufruf von `expand` verwandelt eine Winkelfunktion mit zusammengesetztem Argument in einen zusammengesetzten Ausdruck mit einfachen Winkelfunktionen. Es wurde eines der Additionstheoreme für Winkelfunktionen angewendet.

```
expand(exp(a+sin(b)));
```

$$e^a e^{\sin(b)}$$

Dieser Aufruf von `expand` ersetzt eine Summe im Exponenten durch ein Produkt entsprechend den Potenzgesetzen.

Charakteristisch für solche Transformationen ist die Möglichkeit, das Aufeinandertreffen von vorgegebenen Funktionssymbolen in einem Ausdruck festzustellen. Dabei wird ein Grundsatz der klassischen Funktionsauswertung verletzt: Es wird nicht nur der *Wert* der Aufrufargumente benötigt, sondern auch Information über deren *Struktur*.

So muss etwa beim Aufruf `expand(`sum_1`*`sum_2`)` die Funktion erkennen, dass ihr Argument ein Produkt ist, dessen Faktoren extrahieren und eine Funktion `expandproduct(`sum_1`,`sum_2`)` aufrufen, die aus sum_1 und sum_2 die Summanden extrahiert, alle paarweisen Produkte zwischen den einzelnen Summanden der beiden Summen bildet und diese danach aufsummiert. Details sind hier im Systemkern verborgen:

```
expose(expand);
                      builtin(2060, NIL, "expand", NIL)
expose(_plus);
                       builtin(817, NIL, "_plus", NIL)
```

Ähnlich müsste `expand(sin(`sum`))` das Funktionssymbol `sin` des Aufrufarguments erkennen und danach eine Funktion `expandsin` (in MuPAD heisst sie `sin::expand`) aufrufen.

> Charakteristisch für Transformationsfunktionen ist also der Umstand, dass nicht nur der (semantische) Wert, sondern auch die (syntaktische) Struktur der Aufrufparameter an der Bestimmung des Rückgabewerts beteiligt ist, womit komplexere Teilstrukturen des Ausdrucks zu analysieren sind.

Transformationen sind ihrer Natur nach *rekursiv,* da nach einer Transformation ein Ausdruck entstehen kann, auf den weitere Transformationen angewendet werden können. So führt MuPAD bei der Berechnung von

```
cos(exp(ln(arcsin(x))));
```

$$\sqrt{1 - x^2}$$

erst die Transformation $\exp(\ln(u)) = u$ und dann $\cos(\arcsin(x)) = \sqrt{1 - x^2}$ aus.

Derselbe Funktionsbezeichner kann in unterschiedlichem Kontext in verschiedenen Rollen auftreten, wie etwa das folgende Beispiel für die `exp`-Funktion in MuPAD zeigt:

```
exp(x);
```

$$\exp(x)$$

```
exp(2.0);
```

$$7.389056099$$

```
exp(ln(x+y));
```

$$x + y$$

Im ersten Fall erhalten wir einen Funktionsausdruck mit dem Symbol `exp` als Kopf, im zweiten Fall eine Float-Zahl, die mit einem entsprechenden Näherungsverfahren in einem Funktionsaufruf berechnet wurde, im letzten Fall dagegen wurde eine Transformation angewendet, die das Zusammentreffen von *exp* und *ln* erkannt hat.

Auch stärker „algorithmische“ Funktionen wie etwa `diff`, mit der man in MuPAD Ableitungen berechnen kann, sind ihrer Natur nach Transformationsfunktionen:

```
diff(sin(x),x);
```

$$\cos(x)$$

Beim Zusammentreffen von `diff` und `sin` wurde diese Kombination durch `cos` (multipliziert mit der hier trivialen inneren Ableitung) ersetzt.

```
diff(f(x),x);
```

$$\frac{d}{dx} f(x)$$

Dies ist nur die zweidimensionale Ausgabeform des Funktionsausdrucks `diff(f(x),x)`, der nicht vereinfacht werden konnte, weil über f hier nichts weiter bekannt ist. Einzig die Kettenregel gilt für beliebige Funktionssymbole, so dass folgende Transformation automatisch ausgeführt wird:

```
h:=diff(f(g(x)),x);
```

$$\mathrm{D}(f)(g(x)) \frac{d}{dx} g(x)$$

Hinter der zweidimensionalen Ausgabe verbirgt sich der Ausdruck `D(f)(g(x))*diff(g(x),x)`.

Im letzten Ergebnis tritt mit dem Symbol D sogar eine Funktion auf, die ein reines Funktions*symbol* als Argument hat, weil die Ableitung der Funktion f an der Stelle $y = g(x)$ einzusetzen ist. Ersetzen wir g durch ein "bekanntes" Funktionssymbol, so erhalten wir die aus der Kettenregel gewohnte Formeln.

```
eval(subs(h,f=sin));
```

$$\cos(g(x)) \frac{d}{dx} g(x)$$

Solche "Funktionen von Funktionen" entstehen in verschiedenen mathematischen Zusammenhängen auf natürliche Weise, da auch Funktionen Gegenstand mathematischer Kalküle (etwa der Analysis) sind. Die Ableitung von $f(x)$, die in Maple nicht nur als `diff(f(x),x)`, sondern auch als `D(f)(x)` dargestellt wird, kann in Mathematica und MuPAD näher an der üblichen mathematischen Notation als `f'[x]` bzw. `f'(x)` eingegeben werden. Es ist in diesem Zusammenhang wichtig und mathematisch korrekt, zwischen der Ableitung der Funktion f und des Ausdrucks $f(x)$ zu unterscheiden; gerade dieser Umstand wird durch die beschriebene Notation berücksichtigt. Schließlich ist `f'` keine neue syntaktische Einheit, sondern das Ergebnis der Anwendung des Postfix-Operators `'` auf f, wie eine Strukturanalyse zeigt.

```
f'(x);
```

$$\texttt{D(f)(x)}$$

"Funktionen von Funktionen" können auch ein Eigenleben führen:

```
D(sin);
```

$$\cos$$

```
D(exp+ln);
```

$$\exp\ +\ \left(a \mapsto a^{-1}\right) \quad \text{(Maple)}$$

$$\frac{1}{id} + \exp \quad \text{(MuPAD)}$$

Um die Antwort im letzten Beispiel zu formulieren, wurde eine weitere Funktion benötigt, von der nur die Zuordnungsvorschrift bekannt ist, die aber keinen Namen besitzt. Eine solche Funktion wird auch als *reine Funktion* (pure function) bezeichnet. Sie ist das Gegenteil eines Funktionssymbols, von dem umgekehrt der Name, aber keine Anwendungsvorschrift bekannt war.

3.6 Steuerstrukturen im symbolischen Rechnen

Die größte Ähnlichkeit mit klassischen Programmiersprachen weist ein CAS im Bereich der Steuerstrukturen auf. Es werden in der einen oder anderen Form alle für eine imperative Programmiersprache üblichen Steuerstrukturen (Anweisungsfolgen, Verzweigungen, Schleifen) zur Verfügung gestellt, die sich selbst in der Syntax an gängigen Vorbildern wie PASCAL oder C orientieren. Auch Unterprogrammtechniken stehen zur Verfügung, wie wir im Abschnitt "Funktionen" bereits gesehen hatten.

Da (geschachtelte) Listen eine zentrale Stellung im Datentypdesign von CAS einnehmen, wird für diese auch ein ausreichendes Repertoire an Funktionalität zur Verfügung gestellt. Dabei ist zu beachten, dass MAPLE und MUPAD die etwa in einem Funktionsaufruf auftretende, kommaseparierte *Argumentliste* (expression sequence) als grundlegenden Datentyp verwenden, aus dem durch entsprechende Funktionen Listen, Mengen und andere Datenstrukturen erstellt werden können, die anderen CAS dagegen direkt mit auch dem Nutzer zugänglichen Listen als grundlegendem Datentyp operieren.

Um mit solchen Listen umzugehen wird erst einmal ein Zugriffsoperator auf einzelne Listenelemente, evtl. auch über mehrere Ebenen hinweg, benötigt, der gewöhnlich `part` (als Teil einer Liste) oder `op` (für Operand in einer Argumentliste) heißt. Die meisten Systeme stellen auch eine iterierte Version zur Verfügung, mit der man tiefergelegene Teilausdrücke in einer geschachtelten Liste selektieren kann.

Mit diesen Operatoren wäre eine Listentraversion nun mit der klassischen `for`-Anweisungen möglich, etwa als

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

wobei `nops(l)` die Länge der Liste l zurückgibt und mit `op(l,i)` auf die einzelnen Listenelemente zugegriffen wird. Aus naheliegenden Effizienzgründen stellen die meisten CAS jedoch eine spezielle Listentraversion der Form

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

zur Verfügung.

Daneben spielt die Listenmanipulation eine wichtige Rolle, insbesondere deren Generierung, selektive Generierung und uniforme Manipulation. Zur Erzeugung von Listen gibt es in allen CAS einen *Sequenzierungsoperator*, der eine Liste aus einzelnen Elementen nach einer Bildungsvorschrift generiert. In MAPLE und MUPAD wird dafür ein eigener Operator verwendet. REDUCE hat die Syntax von `for` so erweitert, dass ein Wert zurückgegeben wird.

MAPLE:

```
u:=[seq(i^2,i=1..5)];
```

$$u := [1, 4, 9, 16, 25]$$

MUPAD:

```
u:=[i^2 $ i=1..5];
```

$$u := [1, 4, 9, 16, 25]$$

REDUCE:

```
u:=for i:=1:5 collect i^2;
```

$$u := \{1, 4, 9, 16, 25\}$$

Bei der selektiven Listengenerierung möchte man aus einer bereits vorhandenen Liste alle Elemente mit einer gewissen Eigenschaft auswählen. MAPLE, MUPAD und MATHEMATICA haben dafür einen eigenen Select-Operator, der als Argumente eine boolesche Funktion und eine Liste nimmt und die gewünschte Teilliste zurückgibt. REDUCE nutzt für diesen Zweck die erweiterte `for`-Syntax, mit der man auch iteriert erzeugte Listen zusammenhängen kann, sowie `if`-Anweisungen, die einen Wert zurückgeben.

Betrachten wir etwa, wie man die Primzahlen bis 50 in einer Liste aufsammeln kann:

MAPLE (aehnlich MUPAD):

```
select(isprime,[$1..50]);
```

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

MATHEMATICA:

```
Select[Range[1,50],PrimeQ]
```

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

MACSYMA:

```
sublist(makelist(i,i,1,50),primep);
```

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

REDUCE:

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

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

Uniforme Listenmanipulationen treten immer dann auf, wenn man auf alle Elemente einer Liste ein und dieselbe Funktion anwenden möchte. Viele der Systeme gehen bei Funktionen, die auf Listen angewendet werden, jedoch nur für deren einzelne Elemente einen Sinn ergeben, davon aus, dass die Funktion auf die einzelnen Elemente anzuwenden ist. So liefert etwa MuPAD

```
u:=[sin(i) $ i=1..3];
```

$$[\sin(1), \sin(2), \sin(3)]$$

```
float(u);
```

$$[0.8414709848, 0.9092974268, 0.141120008]$$

wobei offensichtlich die Transformationsfunktion `float` $\circ$ `list` $\longrightarrow$ `list` $\circ$ `float` vorgenommen worden ist. Dort, wo dies nicht automatisch geschieht, kann die Funktion `map` eingesetzt werden, die als Argumente eine Funktion f und eine Liste l nimmt und die Funktion auf alle Listenelemente anwendet.

```
u:=[1,2,3];
```

$$[1, 2, 3]$$

```
sin(u);
```

$$\sin([1, 2, 3])$$

aber

```
map(u,sin);
```

$$[\sin(1), \sin(2), \sin(3)]$$

In einem Anhang zu diesem Kapitel sind die wichtigsten Operationsmöglichkeiten auf Listen in den einzelnen CAS zusammengestellt.

Wir hatten in den Beispielen bereits gesehen, dass in REDUCE

Steuerstrukturen einen Wert zurückgeben.

Dieses Verhalten zeigen die meisten der Systeme (ausgenommen MAPLE, MuPAD und teilweise MATHEMATICA). Es zeigt sich, dass die spezielle Art einer Steuerstruktur es sogar erlaubt, diese selbst als Funktionsaufruf zu implementieren, etwa wie in MATHEMATICA:

```
> While[test,body]
```

führt die Anweisungsfolge `body` so lange aus, bis `test` den Wert *true* zurückgibt. Dies eröffnet weitere Möglichkeiten der Unifizierung von Funktions- und Steuerstrukturen.

Ein Beispiel (MuPAD)

Gleichungen über Restklassenringen lassen sich lösen, indem alle möglichen (endlich vielen) Reste nacheinander in die Gleichung eingesetzt werden. Bestimmen wir als Beispiel alle Lösungen der Kongruenz $x^3 + x + 1 \equiv 0 \ (mod\ 31)$.

Zunächst erstellen wir eine Wertetafel der Funktion.

```
Z:=Dom::IntegerMod(31);
```

```
                              Dom::IntegerMod(31)
```

```
wertetafel:=[[x,Z(x^3+x+1)]$x=0..30];
```

```
[[0, 1 mod 31], [1, 3 mod 31], [2, 11 mod 31], [3, 0 mod 31],
   [4, 7 mod 31], [5, 7 mod 31], [6, 6 mod 31], [7, 10 mod 31],
   [8, 25 mod 31], [9, 26 mod 31], [10, 19 mod 31], [11, 10 mod 31],
   [12, 5 mod 31], [13, 10 mod 31], [14, 0 mod 31], [15, 12 mod 31],
   [16, 21 mod 31], [17, 2 mod 31], [18, 23 mod 31], [19, 28 mod 31],
   [20, 23 mod 31], [21, 14 mod 31], [22, 7 mod 31], [23, 8 mod 31],
   [24, 23 mod 31], [25, 27 mod 31], [26, 26 mod 31], [27, 26 mod 31],
   [28, 2 mod 31], [29, 22 mod 31], [30, 30 mod 31]]
```

Sie sehen, dass genau für die Reste $x = 3$ und $x = 14$ der Funktionswert gleich 0 ist. Diese beiden Elemente können mit einem `select`-Kommando ausgewählt werden.

```
select(wertetafel,x->iszero(op(x,2)));
```

```
                        [[3, 0 mod 31], [14, 0 mod 31]]
```

Schließlich extrahieren wir mit `map` die Liste der zugehörigen x-Werte aus der Liste der Paare.

```
map(%,op,1);
```

```
                                    [3, 14]
```

`map(l,op,1)` ist dabei eine Kurzform für `map(l,x->op(x,1))`.

Eine kompakte Lösung der Aufgabe lautet also:

```
select([$0..30],x->iszero(Z(x^3+x+1)));
```

Mit dem folgenden Kommando können Sie sich einen Überblick über die Nullstellen von $x^3 + x + 1$ $(mod\ p)$ für verschiedene Primzahlen p verschaffen:

```
sol:=proc(p) begin
  if isprime(p)<>TRUE then hold(sol)(p)
  else select([$0..(p-1)],x->iszero(Dom::IntegerMod(p)(x^3+x+1)))
  end_if
end_proc;
```

Das `if`-Kommando beschränkt den Definitionsbereichs von `sol` auf Primzahlen. Für alle anderen Argumente x bleibt `sol(p)` in seiner symbolischen Form stehen.

Nun wird diese Funktion auf die weiter oben generierte Liste von Primzahlen angewendet.

```
primelist:=select([$1..50],isprime);
map(primelist,p->[p,sol(p)]);
```

```
[[2, []], [3, [1]], [5, []], [7, []], [11, [2]], [13, [7]], [17, [11]],
   [19, []], [23, [4]], [29, [26]], [31, [3, 14]], [37, [25]], [41, []],
   [43, [38]], [47, [25, 34, 35]]]
```

Sie erkennen, dass die Gleichung für verschiedene p keine (etwa für $p = 2$), eine (etwa für $p = 3$), zwei (für $p = 31$) oder drei (für $p = 47$) verschiedene Lösungen haben kann. Dem entspricht eine Zerlegung des Polynoms $P(x) = x^3 + x + 1$ über dem Restklassenkörper $\mathbf{Z}_p$ in Primpolynome: Im

ersten Fall ist $P(x)$ irreduzibel, im zweiten zerfällt es in einen linearen und einen quadratischen Faktor und in den letzten beiden Fällen in drei Linearfaktoren, wobei im vorletzten Fall eine der Nullstellen eine doppelte Nullstelle ist. Mit `map` und `factor` kann das nachgeprüft werden.

```
map(primelist,p->[p,map(factor(poly(x^3+x+1,[x],IntMod(p))),expr)]);
```

$$\Big[\left[2, x+x^3+1\right],\ \left[3, (x-1)\left(x+x^2-1\right)\right],\ \left[5, x+x^3+1\right],$$
$$\left[7, x+x^3+1\right],\ \left[11, (x-2)\left(2x+x^2+5\right)\right],\ \left[13, (x+6)\left(x^2-6x+-2\right)\right],$$
$$\left[17, (x+6)\left(x^2-6x+3\right)\right],\ \left[19, x+x^3+1\right],\ \left[23, (x+-4)\left(4x+x^2-6\right)\right],$$
$$\left[29, (x+3)\left(x^2-3x+10\right)\right],\ \left[31, (x+-3)\left(x-14\right)^2\right],\ \left[37, (x+12)\left(x^2-12x-3\right)\right],$$
$$\left[41, x+x^3+1\right],\ \left[43, (x+5)\left(x^2-5x+-17\right)\right],\ \left[47, (x+12)\,(x+13)\,(x+22)\right]\Big]$$

Einige Erläuterungen: `poly(x^3+x+1,[x],IntMod(p))` konstruiert das Polynom $f = x^3 + x + 1 \in \mathbf{Z}_p[x]$, `factor(f)` zerlegt dieses Polynom in Faktoren, allerdings als Objekt vom Typ `Factored`, der für die Ausgabe nicht so optimal ist. Deshalb werden vom äußeren `map` die einzelnen Faktoren in Ausdrücke vom Typ `DOM_EXPR` zurückverwandelt.

Funktionsausdrücke mit booleschem Wert

Boolesche Funktionen spielen in gewissen Steuerstrukturen (`while`, `if`) eine wichtige Rolle, wobei zur korrekten Ablaufsteuerung an dieser Stelle Funktions*aufrufe* abgesetzt werden müssen, die keinen Funktions*ausdruck*, sondern (garantiert) einen der beiden Werte `true` oder `false` zurückliefern. Dies gilt vor allem für relationale Operatoren wie = (equal), die in vielen anderen Kontexten auch als Operator*symbole* verwendet werden.

So wird Gleichheit in den verschiedenen CAS sowohl bei der Formulierung eines Gleichungssystems als auch dessen Lösung verwendet (exemplarisch in MuPAD)

```
gls:={2*x+3*y=2, 3*x+2*y=1};
```

```
                    {3 x + 2 y = 1, 2 x + 3 y = 2}
```

```
solve(gls,{x,y});
```

```
                          {{x = -1/5, y = 4/5}}
```

In beiden Kontexten wurde = als Operatorsymbol verwendet, aus dem ein syntaktisches Objekt "Gleichung" als Funktionsausdruck konstruiert worden ist.

Die meisten CAS verfahren mit symbolischen Ausdrücken der Form $a = b$ auf ähnliche Weise. Eine boolesche Auswertung wird in einem booleschen Kontext automatisch vorgenommen oder kann in einigen CAS durch spezielle Funktionen (Maple: `evalb`, MuPAD: `bool`) erzwungen werden. Allerdings entsprechen die Ergebnisse nicht immer den Erwartungen (MuPAD, ähnlich auch die anderen CAS):

```
1=2;
```

```
                                  1 = 2
```

aber

```
if 1=2 then yes else no end_if;
```

```
                                   no
```

Sehen wir uns die boolesche Auswertung einzelner Ausdrücke näher an:

```
bool(x+y=3);
```

```
                                    FALSE
```

```
bool(x+y=x+y);
```

```
                                    TRUE
```

Im ersten Fall ist die Antwort für alle Variablenbelegungen $(x, y) = (t, 3 - t)$ falsch, aber danach war hier nicht gefragt. Im zweiten Fall sind linke und rechte Seite syntaktisch gleich.

```
bool(x+x=2*x);
```

```
                                    TRUE
```

```
bool(x*(x+1)=x*x+x);
```

```
                                    FALSE
```

```
bool(1<sqrt(5));
```

```
Error: Can't evaluate to boolean [_less]
```

```
bool(1<float(sqrt(5)));
```

```
                                    TRUE
```

Im ersten Beispiel sind die beiden Seiten der Gleichung *nach Auswertung* syntaktisch gleich, im zweiten Fall besteht semantische, nicht aber syntaktische Gleichheit. Im dritten Beispiel schließlich konnte gar keine boolesche Auswertung vorgenommen werden, da `sqrt(5)` einen Funktionsausdruck konstruiert, von dem MuPAD nicht weiss, wie er mit 1 zu vergleichen ist. Erst der Hinweis darauf, es doch einmal mit Näherungswerten zu versuchen, führt hier zum gewünschten Ergebnis.

Substitutionslisten

Mehrere der Konstrukte, die wir in diesem Kapitel kennengelernt haben, spielen in Substitutionslisten zusammen, die die Mehrzahl der CAS als Ausgabeform des `solve`-Operators verwendet.

Betrachten wir etwa die Ausgabe, die MAPLE beim Lösen des Gleichungssystems

$$\{x^2 + y = 2,\ y^2 + x = 2\}$$

produziert:

```
sys:={x^2+y=2, y^2+x=2};
s:=[solve(sys,{x,y})];
```

$$[\{y = -2, x = -2\},\ \{y = 1, x = 1\},$$
$$\{y = RootOf(\_Z^2 - \_Z - 1), x = 1 - RootOf(\_Z^2 - \_Z - 1)\}]$$

MAPLE verwendet aus Gründen, die wir später noch kennenlernen werden, hier selbst Quadratwurzeln nicht von sich aus. Wir wenden deshalb noch auf jeden einzelnen Eintrag der Liste `s` die Funktion `allvalues` an, die einen ROOTOF-Ausdruck, hinter dem sich mehrere Nullstellen verbergen, in die entsprechenden Wurzelausdrücke oder, wenn dies nicht möglich ist, in numerische Näherungslösungen aufspaltet.

```
s:=map(allvalues,s);
```

$$[\{y=-2, x=-2\},\ \{y=1, x=1\},$$
$$\{y=1/2\sqrt{5}+1/2, x=1/2-1/2\sqrt{5}\},$$
$$\{y=1/2-1/2\sqrt{5}, x=1/2\sqrt{5}+1/2\}]$$

Beachten Sie, dass `s` eine Liste aus 3 Elemente war, jetzt aber in eine Liste von 4 Lösungen expandiert. Deren mathematisch ansprechende Form (Verwendung des Gleichheitszeichens) ist auch aus programmiertechnischer Sicht günstig. Wir können jeden einzelnen Eintrag der Liste als lokale Variablensubstitution in komplexeren Ausdrücken verwenden. Eine solche Art von Liste wird als *Substitutionsliste* bezeichnet. Wollen wir etwa durch die Probe die Richtigkeit der Rechnungen prüfen, so können wir nacheinander jeden Listeneintrag aus `s` in `sys` substituieren und nachfolgend vereinfachen:

```
for v in s do print(expand(subs(v,sys))) od;
```

oder noch einfacher

```
probe:=map(v->expand(subs(v,sys)),s);
```

$$[\{2=2\}, \{2=2\}, \{2=2\}, \{2=2\}]$$

Die booleschen Ausdrücke $2=2$ werden aus bereits erläuterten Gründen nur sehr zögerlich ausgewertet. Dies können wir hier wie folgt erzwingen:

```
map(x->evalb(x[1]),probe);
```

$$[true, true, true, true]$$

Ähnlich kann man auch in MuPAD vorgehen, wobei die Gleichungen auch als Liste angegeben werden können.

```
polys:=[x^2+y=2, y^2+x=2];
sol:=solve(polys,{x,y});
```

$$\Big\{[x=1, y=1], [x=-2, y=-2],$$
$$\Big[x=\tfrac{\sqrt{5}}{2}+1/2, y=1/2-\tfrac{\sqrt{5}}{2}\Big], \Big[x=1/2-\tfrac{\sqrt{5}}{2}, y=\tfrac{\sqrt{5}}{2}+1/2\Big]\Big\}$$

MuPAD produziert die Lösungen bereits in expandierter Form. Der Rückgabetyp ist außerdem bereits eine Menge (von Lösungspaaren) und nicht nur eine Sequenz wie in Maple. Für die Probe verwandeln wir die Lösungs*menge* in eine Liste, um Reihenfolge und Anzahl der Ergebnisse zu erhalten.

```
probe:=map([op(sol)],v->map(subs(polys,v),expand));
```

$$[[2=2, 2=2], [2=2, 2=2], [2=2, 2=2], [2=2, 2=2]]$$

Die etwas kompliziertere Syntax ist erforderlich, um `expand` auf die Potenzausdrücke in der zweiten Ebene des Ergebnisses anzuwenden, da imGegensatz zu Maple (siehe oben) `expand` in MuPAD nicht mit Listenbildung vertauscht. Mit entsprechenden Listenoperationen können wir die Probe wieder bis zu einer Liste von `TRUE`-Werten umformen.

```
map(probe,x->bool(_and(op(x))));
```

[TRUE, TRUE, TRUE, TRUE]

Aus solchen Substitutionslisten lassen sich auf einfache Weise aus den Lösungen abgeleitete Ausdrücke zusammenstellen. So liefert das folgende Kommando die Menge aller Lösungspaare in der üblichen Notation:

```
map(sol,u->subs([x,y],u));
```

$$\left\{\left[\frac{1-\sqrt{5}}{2},\frac{1+\sqrt{5}}{2}\right],\left[\frac{1+\sqrt{5}}{2},\frac{1-\sqrt{5}}{2}\right],[-2,-2],[1,1]\right\}$$

Zu jeder der Lösungen kann auch die Summe der Quadrate und die Summe der dritten Potenzen berechnet werden. Hier ist gleich die Berechnung der Potenzen bis zum Exponenten 5 zusammengefasst. Alle diese Rechnungen ergeben ganzzahlige Werte.

```
[map([op(sol)],u->expand(subs(x^i+y^i,u)))$i=1..5];
```

$$[[1,1,-4,2],[3,3,8,2],[4,4,-16,2],[7,7,32,2],[11,11,-64,2]]$$

Wir haben hierbei wesentlich davon Gebrauch gemacht, dass bis auf MATHEMATICA die einzelnen CAS nicht zwischen der mathematischen Relation $A = B$ (`A equal B`) und dem Substitutionsoperator $x = A$ (`x replaceby A`) unterscheiden. MACSYMA, MAPLE, MATHEMATICA, MUPAD und REDUCE geben ihre Lösungen für Gleichungssysteme in mehreren Variablen als solche Substitutionslisten zurück. MACSYMA, MATHEMATICA und REDUCE verwenden diese Darstellung auch für Gleichungen in einer Variablen, MAPLE und MUPAD dagegen in diesem Fall nur, wenn Gleichungen und Variablen als einelementige *Mengen* angegeben werden.

Obige Rechnungen können wie folgt auch mit MACSYMA ausgeführt werden:

```
sys:[x^2+y=2,y^2+x=2];
/* Lösung bestimmen */
sol:solve(sys,[x,y]);
```

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

```
/* Probe ausführen */
map(lambda([u],expand(subst(u,sys))),sol);
```

$$[[2=2,2=2],[2=2,2=2],[2=2,2=2],[2=2,2=2]]$$

```
/* Boolesche Konjunktion aller Ausdrücke dieser Liste */
every(%);
```

TRUE

```
/* Berechnung von x^i+y^i für die 4 Lösungen und i=1..20 */
makelist(map(lambda([u],expand(subst(u,x^i+y^i))),sol),i,1,20);
```

$$[[1,1,-4,2],[3,3,8,2],[4,4,-16,2],\ldots,[9349,9349,-1048576,2],[15127,15127,2097152,2]]$$

Anhang: Operationen auf Listen in den einzelnen CAS

Im Folgenden werden die wichtigsten Listenoperationen für die einzelnen CAS, die wir betrachten, an exemplarischen Beispielen demonstriert. Die Beispiele sind im Einzelnen

Zugriff:	Zugriff auf das Element i der Liste l
Erzeugung:	Erzeugung einer Liste $\{f(i),\ i = a, \ldots, b\}$
Selektion:	Liste der Primzahlen bis 50
Mapping:	Wende die Funktion f auf alle Elemente der Liste l an.
Flatten:	Liste l von Listen aneinander hängen.

Macsyma

Zugriff	`part(l,i)`
Erzeugung	`makelist(f(i),i,a,b)`
Selektion	`sublist(makelist(i,i,1,50),primep)`
Mapping	`map(f,l)`
Flatten	`flatten(l)`

Maple

Zugriff	`op(i,l)`
Erzeugung	`[seq(f(i),i=a..b)]` oder `map(f,[$a..b])`
Selektion	`select(isprime,[seq(i, i=1..50)])`
Mapping	`map(f,l)`
Flatten	`map(op,l)`

Mathematica

Zugriff	`Part[l,i]`
Erzeugung	`Table[f(i),i,{a,b}]` oder `Map[f,Range[a,b]]`
Selektion	`Select[Range[1,50],PrimeQ]`
Mapping	`Map[f,l]`
Flatten	`Flatten[l,1]`

MuPAD

Zugriff	`op(l,i)`
Erzeugung	`[f(i)$ i=a..b]` oder `map([$a..b],f)`
Selektion	`select([$1..50],isprime)`
Mapping	`map(l,f)`
Flatten	`map(l,op)`

Reduce

Zugriff	`part(l,i)`
Erzeugung	`for i:=a:b collect f(i)`
Selektion	`for i:=1:50 join if primep(i) then {i} else {}`
Mapping	`map(f,l)`
Flatten	`for each x in l join x`

Kapitel 4

Das Simplifizieren von Ausdrücken

Eine wichtige Eigenschaft von CAS ist die Möglichkeit, *zielgerichtet* Ausdrücke in eine semantisch gleichwertige, aber syntaktisch verschiedene Form zu transformieren. Wir hatten im letzten Kapitel gesehen, dass solche *Transformationen* einen zentralen Stellenwert im symbolischen Rechnen einnehmen und dass dazu – wieder einmal ähnlich einem Compiler zur Compilezeit – die syntaktische Struktur von Ausdrücken zu analysieren ist.

Zum besseren verständnis der dabei ablaufenden Prozesse ist zunächst zu berücksichtigen, dass einige zentrale Funktionen wie etwa die Polynomaddition aus Effizienzgründen als Funktionsaufrufe[1] implementiert sind und deshalb Vereinfachungen wie $(x+2)+(2x+3) \to 3x+5$ unabhängig von jeglichen Transformationsmechanismen ausgeführt werden.

Weiterhin gibt es eine Reihe von Vereinfachungen wie

$$\begin{array}{rcl} \sin(\arcsin(x)) & \to & x \\ \sin(\arctan(x)) & \to & \frac{x}{\sqrt{x^2+1}} \\ \mathrm{abs}(\mathrm{abs}(x)) & \to & \mathrm{abs}(x), \end{array}$$

die von den meisten CAS automatisch ausgeführt werden.

Jedoch ist nicht immer klar, in welcher Richtung eine mögliche Umformung auszuführen ist. An verschiedenen Stellen einer Rechnung können Transformationen mit unterschiedlichen Intentionen und sogar einander widersprechenden Zielvorgaben erforderlich sein. Zur Berechnung des Integrals

$$\int \log\left(\frac{x+1}{x-1}\right)\, dx = (x+1)\log(x+1) - (x-1)\log(x-1)$$

und zur Lösung der Gleichung

$$\log(x+1) - \log(x-1) = 1 \quad \Leftrightarrow \quad x = \frac{e+1}{e-1}$$

sind etwa die Logarithmengesetze in jeweils unterschiedlicher Richtung anzuwenden. Ähnlich kann man polynomiale Ausdrücke expandieren oder aber in faktorisierter Form darstellen, Basen in Potenzfunktionen zusammenfassen oder aber trennen, Additionstheoreme anwenden, um trigonometrische Ausdrücke eher als Summen oder eher als Produkte darzustellen, die Gleichung $\sin(x)^2 + \cos(x)^2 = 1$ verwenden, um eher `sin` durch `cos` oder eher `cos` durch `sin` zu ersetzen usw.

Eine solche *zielgerichtete Transformation* von Ausdrücken in semantisch gleichwertige *mit gewissen vorgegebenen Eigenschaften* wollen wir als **Simplifikation** bezeichnen.

[1] Zudem auf teilweise speziellen Datenstrukturen.

In den meisten CAS sind für solche Simplifikationen eine Reihe von *Transformnationsfunktionen* wie `expand`, `collect`, `factor` oder `normal` implementiert, welche verschiedene, häufig erforderliche, aber fest vorgegebene Simplifikationsstrategien (Ausmultiplizieren, Zusammenfassen von Termen nach gewissen Prinzipien, Anwendung von Additionstheoremen für Winkelfunktionen, Anwendung von Potenz- und Logarithmengesetzen usw.) *lokal* auf einen Ausdruck anwenden.

Daneben existiert meist eine (oder mehrere) komplexere Funktion `simplify`, die das Ergebnis verschiedener Transformationsstrategien miteinander vergleicht und an Hand des Ergebnisses entscheidet, welches denn nun das "einfachste" ist.

Überdies kann es einzelne Funktionen geben, das gilt besonders für den Substitutionsoperator, wo ein noch eingeschränkteres Transformationsrepertoire ausgeführt wird (d.h. diese Funktionen führen nich einmal zu einer "vollständigen Evaluierung").

Diese Strategie wird von den Systemen DERIVE, MAPLE, MATHEMATICA und MUPAD verwendet. Ihr Vorteil ist die leichte Änderbarkeit der Simplifikationsrichtung im Laufe des interaktiven Dialogs. Der Nachteil dieses Herangehens besteht in der relativen Starrheit des Simplifikationssystems, womit ein Abweichen von den fest vorgegebenen Simplifikationsstrategien nur unter erheblichem Aufwand möglich ist.

Betrachten wir jedoch zunächst die Art, wie solche Transformationen in den einzelnen CAS ausgeführt werden. Es sind zwei grundlegend verschiedene Herangehensweisen bekannt, das *funktionale Transformationskonzept* (MAPLE, MUPAD) und das *regelbasierte Transformationskonzept* (REDUCE). MACSYMA und MATHEMATICA stellen beide Mechanismen zur Verfügung.

4.1 Das funktionale Transformationskonzept

Beim funktionalen Konzept werden Transformationen als Funktionsaufrufe realisiert, die die beschriebenen Besonderheiten durch eine genaue syntaktische Analyse der (ausgewerteten) Aufrufparameter berücksichtigen. Da in die Abarbeitung der entsprechenden Funktion die *Struktur* der Argumente mit eingeht, werden dazu Funktionen benötigt, die diese Struktur wenigstens teilweise analysieren. MAPLE verfügt für diesen Zweck über die Funktion `type(A,T)`, die prüft, ob ein Ausdruck A den "Typ" T hat. Eine ähnliche Rolle spielt die MUPAD-Funktion `type(x)`. Wie bereits an anderer Stelle erwähnt handelt es sich dabei allerdings **nicht um ein strenges Typkonzept für Variablen**, sondern um eine **syntaktische Typanalyse für Ausdrücke**, die in der Regel nur die Syntax der obersten Ebene des Ausdrucks analysiert (z.B. entsprechende Schlüsselworte an der Stelle 0 der zugehörigen Liste auswertet) oder ein mit dem Bezeichner verbundenes Typschlüsselwort abgreift. Dies erkennen wir etwa an folgendem Beispiel:

```
exp(ln(x));
```

$$x$$

```
h:=exp(ln(x)+ln(y));
```

$$e^{\ln(x)+\ln(y)}$$

```
simplify(h);
```

$$x\,y$$

Die Struktur des Arguments verbirgt im zweiten Beispiel die Anwendbarkeit der Transformationsregel. Erst nach eingehender Analyse, die mit `simplify` angestoßen wird, gelingt die Umformung.

Betrachten wir als Beispiel, wie MAPLE die Exponentialfunktion definiert:

```
interface(verboseproc=2):print(`exp`);
```

```
 proc(x)
local i,t,q;
options `Copyright 1993 by Waterloo Maple Software`;
    if nargs <> 1 then ERROR(`expecting 1 argument, got `.nargs)
    elif type(x, 'complex(float)') then evalf('exp'(x))
    elif type(x, 'rational') then exp(x) := 'exp'(x)
    elif type(x, 'function') and op(0, x) = ln then exp(x) := op(1, x)
    elif type(x, 'function') and type(op(0, x), 'function') and
    op([0, 0], x) = `@` and op([0, 1], x) = ln then
        exp(x) := op([0, 2], x)(op(x))
    elif type(x, 'function') and type(op(0, x), 'function') and
    op([0, 0], x) = `@@` and op([0, 1], x) = ln then
        exp(x) := (ln@@(op([0, 2], x) - 1))(op(x))
    elif type(x, `*`) and type(- I*x/Pi, 'rational') then
        i := - I*x/Pi;
        if 1 < i or i <= -1 then
            t := trunc(i);
            t := t + irem(t, 2);
            exp(x) := exp(I*(i - t)*Pi)
        elif type(6*i, 'integer') or type(4*i, 'integer') then
            exp(x) := cos(- I*x) + I*sin(- I*x)
        else exp(x) := 'exp'(x)
        fi
    elif type(x, `*`) and nops(x) = 2 and type(op(1, x), 'rational') and
    type(op(2, x), 'function') and op([2, 0], x) = ln then
        if type(op(1, x), fraction) and irem(op([1, 2], x), 2, 'q') = 0
        then exp(x) := sqrt(op([2, 1], x))^(op([1, 1], x)/q)
        else exp(x) := op([2, 1], x)^op(1, x)
        fi
    else exp(x) := 'exp'(x)
    fi
end
```

Die meisten Zeilen beginnen mit `type(x, ...)`, analysieren also zuerst die Struktur des aufrufenden Arguments. Sehen wir uns die einzelnen Zeilen nacheinander an.

```
    elif type(x, 'complex(float)') then evalf('exp'(x))
```

Handelt es sich um eine Float-Zahl, wird `evalf`, das numerische Interface, mit dem Argument `'exp'(x)` aufgerufen. Die Quotes stehen für die fehlende Auswertung, d.h. an dieser Stelle wird das Funktionssymbol samt Argument, also der *Funktionsausdruck* $\exp(x)$ an `evalf` übergeben. `evalf` seinerseits ist eine Transformationsfunktion, die am Kopf `exp` des Aufrufparameters erkennt, dass sie die Arbeit an eine numerische Auswertungsfunktion `'evalf/exp'` (durch die Quotes entsteht ein Funktionsbezeichner!) für die Exponentialfunktion mit komplexwertigem Argument weiterzureichen hat.

Dasselbe gilt in der nächsten Zeile

```
    elif type(x, 'rational') then exp(x) := 'exp'(x)
```

wenn das Argument x nämlich zu einer (ganzen oder) rationalen *Zahl* auswertet. In diesem Fall kann man den Ausdruck nicht weiter vereinfachen und es wird der entsprechende *Funktionsausdruck* zurückgeben:

```
exp(1/2);
```

```
                                  exp(1/2)
```

```
exp(27/3);
```

```
                                   exp(9)
```

Am letzten Beispiel sehen wir noch einmal, dass das Argument vor dem Aufruf von `exp` wirklich ausgewertet wurde. Im dritten else-Zweig

```
elif type(x, 'function') and op(0, x) = ln then exp(x) := op(1, x)
```

wird schließlich die von uns betrachtete Transformation vorgenommen: Ist x ein Funktionsausdruck, der mit `ln` beginnt, so wird das (erste und einzige) Argument dieses Funktionsausdrucks zurückgegeben. Die folgenden Zeilen dienen der Analyse von Ausdrücken, die den Funktionsiterationsoperator verwenden. Wir wollen sie übergehen. Interessant ist noch die folgenden Zeilen:

```
elif type(x, `*`) and type(- I*x/Pi, 'rational') then
    i := - I*x/Pi;
    if 1 < i or i <= -1 then
        t := trunc(i);
        t := t + irem(t, 2);
        exp(x) := exp(I*(i - t)*Pi)
    elif type(6*i, 'integer') or type(4*i, 'integer') then
        exp(x) := cos(- I*x) + I*sin(- I*x)
    else exp(x) := 'exp'(x)
    fi
```

Hier wird geprüft, ob es sich vielleicht um einen Ausdruck $e^{I\pi i}$ handelt, den man bekanntlich in die Form $cos(\pi i) + I\,sin(\pi i)$ umformen kann. Wenn nicht $-1 < i \leq 1$ gilt, wird `exp` mit einem entsprechend adjustierten Argument rekursiv aufgerufen, ansonsten wird geprüft, ob i ein Vielfaches von $1/6$ oder $1/4$ ist. Für die entsprechenden Winkelgrößen sind die Funktionswerte $\sin(x)$ und $\cos(x)$ bekannt und die entsprechende Transformation wird durchgeführt, ansonsten wird der Funktionsausdruck in unveränderter Form zurückgegeben. Ähnlich ist die `exp`-Funktion in MuPAD definiert, was Sie sich mit `expose(exp)` anschauen können.

Transformationen sind in einem solchen Konzept manchmal nur über Umwege zu realisieren, da Funktions*aufrufe* ihre Argumente auswerten, was deren ursprüngliche Struktur verändern kann. Möchte man etwa das Polynom $f = x^3 + x^2 - x + 1 \in \mathbf{Z}[x]$ nicht über den ganzen Zahlen, sondern modulo 2, also im Ring $\mathbf{Z}/2\mathbf{Z}[x]$, faktorisieren, so führen in Maple weder

```
factor(f) mod 2;
```

$$x^3 + x^2 + x + 1$$

noch

```
factor(f mod 2);
```

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

zum Ziel. Das zweite Ergebnis kommt der Wahrheit zwar schon näher (Ausmultiplizieren ergibt $x^3 + x^2 + x + 1 \equiv f \ (mod\ 2)$), aber ist wegen $(x^2 + 1) \equiv (x + 1)^2 \ (mod\ 2)$ noch immer falsch.

In beiden Fällen wird bei der Auswertung der Funktionsargumente die für eine Transformation notwendige Kombination der Symbole `factor` und `mod` zerstört, denn `factor` ist ein Funktions*aufruf*, der als Ergebnis ein Produkt, die Faktorzerlegung des Arguments über den ganzen Zahlen, zurückliefert. Im ersten Fall (der intern als `mod(factor(f),2)` umgesetzt wird) wird vor

dem Aufruf von `mod` das Polynom (in $\mathbf{Z}[x]$) faktorisiert. Das Ergebnis enthält die Information `factor` nicht mehr, so dass `mod` als Reduktionsfunktion $\mathbf{Z}[x] \to \mathbf{Z}_2[x]$ operiert und die Koeffizienten dieser ganzzahligen Faktoren reduziert. Im zweiten Fall dagegen wird das Polynom f erst modulo 2 reduziert. Das Ergebnis enthält die Information `mod` nicht mehr und wird als Polynom aus $\mathbf{Z}[x]$ betrachtet und faktorisiert.

In klassischen Programmiersprachen werden derartige Ambiguitäten heutzutage über Polymorphie aufgelöst, was ein detailliertes Typkonzept voraussetzt. Da ein solches in CAS der zweiten Generation nicht zur Verfügung steht, müsste auf verschiedene Funktionsnamen, etwa `factor` und `factormod`, zurückgegriffen werden. Um dies wenigstens in dem Teil, der dem Nutzer sichtbar ist, zu vermeiden, führt MAPLE ein Funktionssymbol `Factor` ein, das sein Argument unverändert zurückgibt, so dass `mod` im Aufruf

```
Factor(f) mod 2;
```

$$(x+1)^3$$

der intern sofort als `mod(Factor(f),p)` umgesetzt wird, an der speziellen Form seines ersten Arguments erkennen kann, dass modular zu faktorisieren ist und die spezielle modulare Faktorisierungsroutine `'mod/factor'(f,p)` aufrufen kann[2]. Solche Funktionssymbole werden in der MAPLE-Dokumentation als *inerte Funktionen* bezeichnet. In der hier verwendeten Terminologie handelt es sich um Funktionssymbole ohne Funktionsdefinition, so dass alle Funktionsaufrufe zu Funktionsausdrücken mit 'Factor' als Kopf auswerten.

In MATHEMATICA und MUPAD sind solche Hilfskonstruktionen nicht erforderlich. So lässt sich etwa in MATHEMATICA der modulare Faktorisierungsalgorithmus als

```
Factor[f,Modulus -> 2]
```

$$(1+x)^3$$

aufrufen, wobei an Vorhandensein und Struktur des zweiten Arguments erkannt wird, welche Faktorisierungsroutine genau zu verwenden ist und diese nach einer entsprechenden Transformation als Funktionsaufruf aktiviert wird. Dieser Zugang ließe sich auch in MAPLE realisieren.

In MUPAD wird diese Unterscheidung nach dem objektorientierten Ansatz aus dem (syntaktischen) Typ der Aufrufargumente deduziert:

```
f:=poly(x^3+x^2-x+1,[x],Dom::IntegerMod(2));
```

$$\mathrm{poly}(x^3+x^2+x+1, [x], \mathrm{Dom} :: \mathrm{IntegerMod}(2))$$

```
factor(f);
```

$$\mathrm{poly}(x+1, [x], \mathrm{Dom} :: \mathrm{IntegerMod}(2))^3$$

```
expr(%);
```

$$(x+1)^3$$

Das funktionale Transformationskonzept kann an manchen Stellen (die allerdings von den Systementwicklern explizit vorgesehen sein müssen) relativ einfach erweitert werden. Betrachten wir dazu die MAPLE-Funktion `expand`. `expand(A)` analysiert die Gestalt des Ausdrucks $A = f(x_1, \ldots, x_n)$ und ruft eine entsprechende Funktion

$$\texttt{'expand/f'}(x_1, \ldots, x_n)$$

[2] Die Interna sind im Systemkern verborgen.

auf, die die eigentliche Transformation vornimmt. Hier wird die Fähigkeit eines CAS zur Stringmanipulation verwendet, indem zur Laufzeit (!), während des Aufrufs von `expand`, aus einem in den Argumenten vorkommenden Funktionssymbol `f` ein neuer Funktionsname `` `expand/f` `` generiert und, sofern eine entsprechende Funktion definiert ist, diese aufgerufen wird. Ansonsten bleibt $f(x_1, \ldots, x_n)$ unverändert. Dies ist zugleich der Mechanismus, der es einem Nutzer erlaubt, die Fähigkeiten des Systems zum "Expandieren" für selbstdefinierte Funktionen zu erweitern.

Soll etwa l eine additive Funktion darstellen, für die `expand` die Transformation $l(x+y) = l(x) + l(y)$ ausführt, so muss man dazu eine Funktion

```
`expand/l`:= proc(y) local x;
   x:=expand(y);
   if type(x,`+`) then convert(map(l,[op(x)]),`+`)
   else l(x) fi
end;
```

definieren.

Einen ähnlichen Zugang verfolgt MuPAD: Hier kann einem (speziell als Funktionsenvironment zu vereinbarenden) Funktionssymbol l, auf dem `expand` nichttrivial agieren soll, ein Attribut "expand" im objektorientierten Sinn (Slot) mit einer entsprechenden **Funktion** zugeordnet werden. Obigen Effekt für l kann man folgendermaßen erreichen:

```
l:= funcenv(l);
l::expand:=
  proc(x) begin x:= expand(x);
       if type(x) = "_plus" then _plus(map(op(x),l))
       else l(x)
       end_if
  end_proc;
```

Die erste Zeile verwandelt das Symbol l in ein Funktionsenvironment (vom Typ `DOM_FUNC_ENV`), das einen Eintrag `l::expand` erlaubt, die zweite Zuweisung fügt diesen Slot zur Funktionsdefinition hinzu[3].

Die Nachteile dieses Konzepts fallen sofort ins Auge:

1. Man hat mit jedem Funktionsaufruf eine Menge verschiedener Typinformationen zu ermitteln, womit jeder Funktionsaufruf relativ aufwändige Operationen anstößt. Deshalb ist es oft sinnvoll, bereits berechnete Funktionswerte zu speichern, wenn man weiß, dass sie sich nicht verändern.

2. Die Typanalyse ist bei vielen Funktionen von ähnlicher Bauart, so dass unnötig Code dupliziert wird.

3. Die so definierten Transformationsregeln sind relativ "starr". Möchte man z.B. das Verhalten der `exp`-Funktion dahingehend ändern, dass sie bei rein imaginärem Argument *immer* die trigonometrische Darstellung verwendet, so müsste man den gesamten oben gegebenen Code kopieren, an den entsprechenden Stellen ändern und dann die (natürlich außerdem vor Überschreiben geschützte) `exp`-Funktion entsprechend neu definieren.

Deshalb hat eine Funktionsdefinition eine komplexere Struktur als in einer klassischen Programmiersprache. In Maple etwa kann eine Funktion eine *Funktionswert-Tabelle* anlegen, in die alle bereits berechneten Funktionswerte eingetragen werden. Diese wird inspiziert, bevor die Auswertung entsprechend der Funktionsdefinition initiiert wird. In MuPAD kann eine Funktionsdefinition außerdem noch über eine Tabelle von Funktionsattributen verfügen.

[3] In Versionen MuPAD 1.* wurde `funcattr` als (fest eingebrannter) Funktionsattribut-Konstruktor verwendet.

4.2 Das regelbasierte Transformationskonzept

Beim regelbasierten Zugang wird die im funktionalen Zugang notwendige Code-Redundanz vermieden, indem der Transformationsprozess in einen Programm- und einen Datenteil aufgespalten wird. Der Datenteil enthält die jeweils konkret anzuwendenden Ersetzungsregeln, also Informationen darüber, welche Kombinationen von Funktionssymbolen wie zu ersetzen sind. Der Programmteil, der *Simplifikator*, stellt die erforderlichen Routinen zur Mustererkennung und Unifikation bereit. Auf diese Weise werden die Transformationen über einen allgemeinen Simplifikationsmechanismus als direkte Transformationen des Ableitungsbaums ausgeführt. Dazu wird ein Satz von *Transformationsregeln* der Gestalt `lhs => rhs` auf den aktuellen Ausdruck A angewendet, indem grob gesprochen in A Teilausdrücke der Gestalt *lhs* gesucht und durch Ausdrücke der Gestalt *rhs* ersetzt werden. Das wird so lange wiederholt, bis keine Ersetzungen mehr möglich sind. Im Gegensatz zum funktionalen Zugang sind hier also der Simplifikator und die jeweils anzuwendenden Regelsätze voneinander getrennt, was es auf einfache Weise ermöglicht, Regelsätze zu ergänzen und für spezielle Zwecke zu modifizieren und anzupassen.

Damit enthält die Programmiersprache eines CAS neben funktionalen und imperativen auch Elemente einer logischen Programmiersprache. Wir werden uns in einem späteren Abschnitt genauer mit der Funktionsweise eines solchen Regelsystems vertraut machen. An dieser Stelle wollen wir uns nur einmal anschauen, welche Regeln REDUCE zum Simplifizieren verschiedener Funktionen kennt. Die jeweiligen Regeln sind unter dem Funktionssymbol als Liste gespeichert und können mit der Funktion `showrules` ausgegeben werden:

```
showrules exp;

{exp(~x) => e^x}

showrules log;

{log(1) => 0,
 log(e) => 1,
 log(e^~x) => x,
 df(log(~x),~x) => 1/x,
 df(log(~x/~y),~z) => df(log(x),z) - df(log(y),z)}
```

Zum Verständnis der ersten Regel muss hinzugefügt werden, dass REDUCE die Exponentialfunktion damit als spezielle Potenzfunktion auffasst und Potenzfunktionen wie die anderen Arithmetikoperatoren aus Effizienzgründen außerhalb des Regelsystems "fest verdrahtet" sind. Wie bei Funktionen ist auch bei Regeln zwischen Deklaration und Anwendung zu unterscheiden. Dabei ist insbesondere auch die Verwendung von formalen Parametern denkbar, die als Platzhalter für beliebige Teilausdrücke, d.h. für entsprechende Teilbäume in den Regeln auftreten. So vereinfacht REDUCE etwa $log(exp(A))$ zu A, egal wie der Teilausdruck A beschaffen ist. Solche formalen Parameter werden dabei durch eine vorangestellte Tilde gekennzeichnet und sind bei einer Anwendung an *allen* Stellen durch denselben Ausdruck zu ersetzen. Regeln dieser Art nennt man allgemeine Regeln im Gegensatz zu Regeln, wo der zu ersetzende Teilausdruck *genau* mit *lhs* übereinzustimmen hat, welche man spezielle Regeln nennt. So sind die beiden ersten Regeln für `log` spezielle, die beiden anderen allgemeine. Letztere beiden Regeln geben die Transformationsvorschriften an, wenn die Funktionssymbole `log` und `exp` bzw. `df` (für die Ableitung) und `log` irgendwo unmittelbar aufeinandertreffen.

Ein solches Regelsystem kann durchaus einen größeren Umfang erreichen:

```
showrules sin;

{sin(pi) => 0,
```

```
sin(pi/2) => 1,
sin(pi/3) => sqrt(3)/2,
sin(pi/4) => sqrt(2)/2,
sin(pi/6) => 1/2,
sin((5*pi)/12) => sqrt(2)/4*(sqrt(3) + 1),
sin(pi/12) => sqrt(2)/4*(sqrt(3) - 1),
sin((~(~ x)*i)/~(~ y)) => i*sinh(x/y) when impart(y)=0,
sin(atan(~u)) => u/sqrt(1 + u**2),
sin(2*atan(~u)) => 2*u/(1 + u**2),
sin(~n*atan(~u)) => sin((n - 2)*atan(u))*(1 - u**2)/(1 + u**2)
       + cos((n - 2)* atan(u))*2*u/(1 + u**2)  when fixp(n) and n>2,
sin(acos(~u)) => sqrt(1 - u**2),
sin(2*acos(~u)) => 2*u*sqrt(1 - u**2),
sin(2*asin(~u)) => 2*u*sqrt(1 - u**2),
sin(~n*acos(~u)) => sin((n - 2)*acos(u))*(2*u**2 - 1)
       + cos((n - 2)*acos(u))*2* u*sqrt(1 - u**2) when fixp(n) and n>2,
sin(~n*asin(~u)) => sin((n - 2)*asin(u))*(1 - 2*u**2)
       + cos((n - 2)*asin(u))*2* u*sqrt(1 - u**2) when fixp(n) and n>2,
sin((~x + ~(~ k)*pi)/~d) => sign(k/d)*cos(x/d)
               when x freeof pi and abs(k/d)=1/2,
sin((~(~ w) + ~(~ k)*pi)/~(~ d)) =>
       (if evenp(fix(k/d)) then 1 else  - 1)*sin(( w + remainder(k,d)*pi)/d)
               when w freeof pi and ratnump(k/d) and abs(k/d)>=1,
sin((~(~ k)*pi)/~(~ d)) => sin((1 - k/d)*pi) when ratnump(k/d) and k/d>1/2,
sin(asin(~x)) => x,
df(sin(~x),~x) => cos(x)}
```

Manche der angegebenen Regeln sind noch konditional untersetzt, d.h. nur dann anwendbar, wenn die in der Definition verwendeten formalen Parameter mit aktuellen Parametern belegt sind, die noch Zusatzvoraussetzungen erfüllen. Diese Effekte sind von regelorientierten Programmiersprachen wie etwa Prolog aber gut bekannt.

Diese Regeln werden automatisch angewendet, wenn REDUCE das Funktionssymbol `sin` in einem Ausdruck antrifft. So werden etwa (Regel 9 – 11) Ausdrücke der Form $\sin(n \cdot \arctan(x))$ aufgelöst:

```
sin(5*atan(x));
```

$$\frac{x^5 - 10\,x^3 + 5\,x}{\sqrt{x^2+1}\,(x^4 + 2\,x^2 + 1)}$$

Dasselbe Ergebnis kann man mit MAPLE erreichen, da die Definition von `sin` die Information enthält, dass

$$\sin(\arctan(x)) = \frac{x}{\sqrt{1+x^2}}$$

gilt, wie wir an folgendem Quelltextfragment erkennen:

```
print(sin);
proc(x)
    local n, t;
    ...
        elif type(x, 'function') and nops(x) = 1 then
            n := op(0, x);
            t := op(1, x);
    ...
            elif n = 'arctan' then t/sqrt(1 + t^2)
```

```
...
end
```

Allerdings muss dazu über `expand` erst eine Transformation angestoßen werden, die die Funktionssymbole `sin` und `arctan` unmittelbar zusammenbringt, indem die Mehrfachwinkelausdrücke aufgelöst werden.

```
sin(5*arctan(x));
```

$$\sin(5\ \arctan(x))$$

```
expand(%);
```

$$16\,\frac{x}{\left(1+x^2\right)^{5/2}}-12\,\frac{x}{\left(1+x^2\right)^{3/2}}+\frac{x}{\sqrt{1+x^2}}$$

```
normal(%);
```

$$\frac{x\left(5-10\,x^2+x^4\right)}{\left(1+x^2\right)^{5/2}}$$

Solche Regelsysteme gestatten es, Transformationssysteme relativ flexibel zu kombinieren und neue Transformationsfunktionen zu schreiben, wenn es im entsprechenden CAS einen Mechanismus zur lokalen Anwendung von Regelsystemen gibt.

Ein anderer Zugang wird von REDUCE und in gewissem Sinn auch von MACSYMA verfolgt. Dort sind alle Regeln global definiert, aber nicht immer aktiv, sondern vom gerade eingestellten *Kontext* abhängig. Ein eingegebener Ausdruck wird automatisch nach allen im entsprechenden Kontext gerade aktiven Simplifikationsregeln umgeformt. Die Änderung der Regeln erfolgt durch Veränderung des gültigen Kontexts, der durch den Wert von globalen Variablen, sogenannter *Schalter*, bestimmt ist, die man frei untereinander kombinieren kann und die das Zu- bzw. Abschalten bestimmter Simplifikationseffekte bewirken. Auf diese Weise sind Simplifikationsstrategien wesentlich flexibler formulierbar als durch Transformationsfunktionen. Allerdings besteht der Nachteil dieses Zugangs darin, dass man in der Vielzahl der verschiedenen Schalter schnell den Überblick über den jeweils aktuell gültigen Kontext verlieren kann.

Ein größeres Problem bei diesem Zugang ist der Umgang mit einander widersprechenden Simplifikationsstrategien, da man diese nicht gleichzeitig wirken lassen darf, wenn man Endlosschleifen vermeiden möchte. Was im ersten Zugang durch das Design der verschiedenen systemspezifischen Simplifikationsfunktionen gewährleistet wurde, kann hier bei einer *beliebigen* Kombinierbarkeit der verschiedenen Schalter *prinzipiell* nicht gewährleistet werden. So kann man die Beziehung $log(x * y) = log(x) + log(y)$ als Transformationsregel

$$log(x * y) \mapsto log(x) + log(y)$$

aber auch als

$$log(x) + log(y) \mapsto log(x * y)$$

anwenden. Verwendet man einen Schalter, um zwischen beiden Strategien *umzuschalten*, so wird das System einen Ausdruck der Form $\log(u \cdot v) + \log(x) + \log(y)$ immer umformen. Da auch solche Ausdrücke manchmal eine Daseinsberechtigung haben, stellt REDUCE zwei Schalter `combinelogs` und `expandlogs` zur Verfügung, mit denen man das entsprechende Simplifikationsverhalten ein- oder ausschalten kann. Würde man beide gleichzeitig aktivieren, so würde eine unendliche Simplifikationsschleife eintreten. Dies kann man vermeiden, indem das Einschalten des einen Schalters den anderen automatisch ausschaltet[4].

[4] In der Version 3.6 können allerdings beide Schalter nebeneinander existieren. Sind beide aktiviert, so wird zwischen beiden Darstellungen bei jedem Simplifikationszyklus gewechselt. In der Version 3.7 sind die Schalter ohne Wirkung, weil diese Umformungen im Bereich **C** nicht mathematisch exakt sind.

Wir sehen an diesem Beispiel zugleich, dass es in einem solchen Ansatz schwierig ist, gezielte *teilweise* Simplifikationen vorzunehmen.

MACSYMA verfolgt eine Mischung aus beiden Strategien, indem sowohl Schalter zur Beeinflussung des Simplifikationsverhaltens als auch verschiedene Transformationsfunktionen zur Verfügung stehen.

Die Vorteile konsistenter Simplifikationssysteme und deren flexible Einsetzbarkeit werden, ähnlich den lokalen Wertzuweisungen an Variablen, durch *lokale Simplifikationssysteme* miteinander kombiniert. Über einfache Mechanismen zur Erstellung und Anwendung lokaler Simplifikationssysteme verfügen derzeit nur die Systeme MACSYMA, MATHEMATICA und REDUCE. Sie nutzen dabei denselben zentralen Simplifikationsmechanismus wie auch für die interne Transformationen, indem dieser mit *lokalen Regelsystemen* gefüttert wird, welche dieselbe Struktur haben wie die dem System bekannten Regeln für Funktionssymbole, wie wir sie für REDUCE mit `showrules` extrahiert hatten.

4.3 Simplifikation und mathematische Exaktheit

Wir hatten bereits gesehen, dass es in beiden Zugängen zur Simplifikationsproblematik einen

Kern von allgemeingültigen Simplifikationen

gibt, die allen Simplifikationsstrategien gemeinsam sind und deshalb stets automatisch ausgeführt werden.

Dazu gehört zunächst einmal die Strategie, spezielle Werte von Funktionsausdrücken, sofern diese durch "einfachere" Symbole exakt ausgedrückt werden können, durch diese zu ersetzen.

Beispiele (MUPAD):

`sqrt(36)`	$\Rightarrow$	6
`sin(PI/4)`	$\Rightarrow$	$\frac{\sqrt{2}}{2}$
`tan(PI/6)`	$\Rightarrow$	$\frac{\sqrt{3}}{3}$
`arcsin(1)`	$\Rightarrow$	$\frac{\pi}{2}$

Dies trifft auch für kompliziertere Funktionsausdrücke zu, die auf "elementarere" Funktionen zurückgeführt werden, in denen mehr oder weniger gut studierte spezielle mathematische Funktionen auftreten.

`gamma(1/2)`	$\Rightarrow$	$\sqrt{\pi}$
`int(exp(-x^2),x=0..infinity)`	$\Rightarrow$	$\frac{\sqrt{\pi}}{2}$
`int(exp(-x^2),x=0..y)`	$\Rightarrow$	$\frac{\sqrt{\pi}\,\mathrm{erf}(y)}{2}$
`sum(1/i^2,i=1..infinity)`	$\Rightarrow$	$\frac{\pi^2}{6}$
`sum(1/i^7,i=1..infinity)`	$\Rightarrow$	$\zeta(7)$
`sum(1/i^n,i=1..infinity)`	$\Rightarrow$	$\sum_{i=1}^{\infty} i^{-n}$

In den Beispielen wurden die Gamma-Funktion $\Gamma(x)$, die Gaußsche Fehlerfunktion $erf(x)$ sowie die Riemannsche Zeta-Funktion $\zeta(n)$ verwendet. Die entsprechende Summentransformation wird aber nur für ganzzahlige n vorgenommen.

Weiterhin wird auch eine Reihe komplizierterer Umformungen von einigen der Systeme automatisch ausgeführt wie z.B. (MAPLE automatisch, MUPAD erst nach Aufruf von `radsimp`):

`sqrt(24)`	$\Rightarrow$	$2\sqrt{6}$
`sqrt(2*sqrt(3)+4)`	$\Rightarrow$	$\sqrt{3}+1$
`sqrt(11+6*sqrt(2))+sqrt(11-6*sqrt(2))`	$\Rightarrow$	6

Auch werden eindeutige Simplifikationen von Funktionsausdrücken ausgeführt wie etwa

`abs(abs(x))`	$\Rightarrow$	$\lvert x\rvert$
`tan(arctan(x))`	$\Rightarrow$	x
`tan(arcsin(x))`	$\Rightarrow$	$\frac{x}{\sqrt{1-x^2}}$
`abs(-PI*x)`	$\Rightarrow$	$\pi\,\lvert x\rvert$
`cos(-x)`	$\Rightarrow$	$\cos(x)$
`exp(3*ln(x))`	$\Rightarrow$	x^3

Auf den ersten Blick mag es deshalb verwundern, dass folgende Ausdrücke nicht vereinfacht werden:

`sqrt(x^2)`	$\Rightarrow$	$\sqrt{x^2}$
`ln(exp(x))`	$\Rightarrow$	$\ln(\exp(x))$
`arctan(tan(x))`	$\Rightarrow$	$\arctan(\tan(x))$

In jedem der drei Fälle würde der durchschnittliche Nutzer als Ergebnis wohl x erwarten. Für die letzte Beziehung ist das allerdings vollkommen falsch, wie ein Plot der Funktion unmittelbar zeigt:

```
plot(plot::Function2d(arctan(tan(x)),x=-10..10));
```

Wir sehen, dass arctan nur im Intervall $[-\pi/2, \pi/2]$ die Umkehrfunktion von tan ist. Die korrekte Antwort lautet für $x \in \mathbf{R}$ also

$$\arctan(\tan(x)) = x - \left[\frac{x}{\pi} + \frac{1}{2}\right] \cdot \pi,$$

wobei $[a]$ für den ganzen Teil der Zahl $a \in \mathbf{R}$ steht.

Dass auch $\sqrt{x^2} = x$ mathematisch nicht exakt ist, dürfte bei einigem Nachdenken ebenfalls einsichtig sein und als Ergebnis der Simplifikation $|x|$ erwartet werden. Diese Antwort wird auch von REDUCE und MACSYMA gegeben. MAPLE allerdings gibt nach expliziter Aufforderung

`simplify(sqrt(x^2))` $\Rightarrow$ $\operatorname{csgn}(x)\,x$

zurück, obwohl wir in den Beispielen gesehen hatten, dass es auch die Betragsfunktion kennt. Der Grund liegt darin, dass das nahe liegende Ergebnis $|x|$ nur für *reelle* Argumente korrekt ist, nicht dagegen für komplexe. Für komplexe Argumente ist die Wurzelfunktion mehrwertig, so dass $\sqrt{x^2} = \pm x$ eine korrekte Antwort wäre. Da man in diesem Fall oft vereinbart, dass der Wert der Wurzel der Hauptwert ist, also derjenige, der in der oberen komplexen Halbebene liegt, wird hier die *komplexe Vorzeichenfunktion* `csgn` verwendet. In diesem Kontext ist auch die Vereinfachung des Ergebnisses zu $|x|$, dem Betrag der komplexen Zahl x, fehlerhaft.

Für noch allgemeinere mathematische Strukturen, in denen Multiplikationen und deren Umkehrung definiert werden können, wie etwa Gruppen (quadratische Matrizen oder ähnliches), ist allerdings selbst diese Simplifikation nicht korrekt. MUPAD vereinfacht deshalb den Ausdruck auch unter `simplify` nicht.

Die dritte Beziehung $\ln(\exp(x)) = x$ ist wegen der Monotonie der beteiligten Funktionen dagegen für alle reellen Werte von x richtig. Für komplexe Argumente kommt dagegen, ähnlich wie für die Wurzelfunktion, die Mehrdeutigkeit der Logarithmusfunktion ins Spiel.

Weitere interessante Simplifikationsfälle sind die Ausdrücke

`sqrt(1/x)-1/sqrt(x)`	$\Rightarrow$	$\sqrt{\frac{1}{x}} - \frac{1}{\sqrt{x}}$
`sqrt(x*y)-sqrt(x)*sqrt(y)`	$\Rightarrow$	$\sqrt{x\,y} - \sqrt{x}\,\sqrt{y}$

die für solche Argumente, für die sie "sinnvoll" definiert sind (also hier etwa für positive reelle x), zu Null vereinfacht werden können. Für negative reelle Argumente haben wir in diesem Fall

allerdings nicht nur die Mehrdeutigkeit der Wurzelfunktion im Bereich der komplexen Zahlen zu berücksichtigen, sondern kollidieren mit anderen, wesentlich zentraleren Annahmen, wie etwa der automatischen Ersetzung von $\sqrt{-1}$ durch die imaginäre Einheit I. Setzen wir in obigen Ausdrücken $x = y = -1$ und führen diese Ersetzung aus, so erhalten wir im ersten Fall $I - 1/I = 2\,I$ und im zweiten Fall $\sqrt{1} - I^2 = 2$. Solche Inkonsistenzen tief in komplexen Berechnungen versteckt können zu vollkommen falschen Resultaten führen, ohne dass der Grund dafür offensichtlich wird.

Aus ähnlichen Gründen sind übrigens auch die oben betrachteten Transformationen der Logarithmusfunktion mathematisch nicht allgemeingültig:

$$\texttt{ln((-1)*(-1))=ln(-1)+ln(-1)} \quad \Rightarrow \quad 0 = 2i\,\pi$$

Natürlich sind Simplifikationssysteme mit zu rigiden Annahmen für die meisten Anwendungszwecke untauglich, wenn sie derart simple Umformungen "aus haarspalterischen Gründen" nicht oder nur nach gutem Zureden ausführen. Jedes der CAS muss deshalb für sich entscheiden, auf welchen Grundannahmen seine Simplifikationen sinnvollerweise basieren, um ein ausgewogenes Verhältnis zwischen mathematischer Exaktheit einerseits und Praktikabilität andererseits herzustellen.

In der folgenden Tabelle sind die Simplifikationsergebnisse der verschiedenen CAS (in der Grundeinstellung) auf einer Reihe von Beispielen zusammengestellt ($*$ bedeutet unsimplifiziert):

Ausdruck	Derive	Maxima	Maple	MMa	MuPAD	Reduce
	5	5.9.0	8	4.1	2.5	3.7
$\lvert -\pi \cdot x\rvert$	$\pi \cdot \lvert x\rvert$	$\pi \cdot \lvert x\rvert$	$\pi \cdot \lvert x\rvert$	$\pi \cdot \lvert x\rvert$	$\pi \cdot \lvert x\rvert$	$\pi \cdot \lvert x\rvert$
$\arctan(\tan(x))$	$*$	x	$*$	$*$	$*$	$*$
$\arctan(\tan(\frac{25}{4}\pi))$	$\pi/4$	$\pi/4$	$\pi/4$	$\pi/4$	$\pi/4$	$\pi/4$
$\sqrt{x^2}$	$\lvert x\rvert$	$\lvert x\rvert$	$*$	$*$	$*$	$\lvert x\rvert$
$\sqrt{x\,y} - \sqrt{x}\,\sqrt{y}$	$*$	$*$	$*$	$*$	$*$	$*$
$\sqrt{\frac{1}{z} - \frac{1}{\sqrt{z}}}$	$*$	0	$*$	$*$	$*$	$*$
$\ln(\exp(x))$	$*$	x	$*$	$*$	$*$	x
$\ln(\exp(10 * I))$	-2 i (2π-5)	10 i	$*$	10 i-4π i	-4πi+10 i	10 i

Tabelle 1: Simplifikationsverhalten der verschiedenen Systeme an ausgewählten Beispielen

MACSYMA, MAPLE und nun auch MUPAD besitzen mit dem `assume`-Mechanismus die Möglichkeit, einzelnen Symbolen *Eigenschaften* zuzuordnen, die beim Simplifizieren in die Entscheidungsfindung mit einbezogen werden. Solche Eigenschaften können Annahmen über die Natur einer Variablen (`assume(x,integer)`, `assume(x,real)`), die Zugehörigkeit zu einem reellen Intervall (`assume(x>0)`), die spezielle Natur einer Matrix (`assume(m,quadratic)`) oder andere Eigenschaften sein. Symbole, für die Eigenschaften definiert sind, werden in MAPLE mit einer Tilde versehen. Mit speziellen Funktionen (MACSYMA: `facts`, `properties`, MAPLE: `about`, MUPAD: `getprop`) können die gültigen Eigenschaften ausgelesen werden.

```
assume(y,real);
about(y);
Originally y, renamed y~:
  is assumed to be: real

assume(x>0);
about(x);
Originally x, renamed x~:
  is assumed to be: RealRange(Open(0),infinity)
```

MAPLE führt unter die unter zusätzlichen Annahmen gültigen mathematischen Umformungen nun automatisch aus:

```
abs(-Pi*x)                              ⇒  π x~
sqrt(x^2)                               ⇒  x~
simplify(sqrt(x*y)-sqrt(x)*sqrt(y))     ⇒  0
sqrt(1/x)-1/sqrt(x)                     ⇒  0
ln(exp(y))                              ⇒  y~
```

Ähnlich ist das Vorgehen in MuPAD

```
assume(y,Type::Real);
getprop(y);
                              Type::Real
assume(x<0);
getprop(x);
                                  < 0
```

oder Macsyma

```
declare(y,real);
assume(x<0);
facts(x);
                            [0 > x]
facts(y);
                         [KIND(y, REAL)]
```

```
abs(-PI*x)                              ⇒  −xπ
sqrt(x^2)                               ⇒  −x
simplify(sqrt(x*y)-sqrt(x)*sqrt(y))     ⇒  √(xy) − √x √y
```

Dieses Verhalten (MuPAD) ist hier korrekt, da wir $x < 0$ angenommen hatten. Expandieren der Wurzel ist nur für positive reelle x korrekt[5].

```
assume(x>0);
```

```
simplify(sqrt(x*y)-sqrt(x)*sqrt(y))     ⇒  0
sqrt(1/x)-1/sqrt(x)                     ⇒  0
ln(exp(y))                              ⇒  y
```

`assume` überschreibt in Maple und MuPAD die bisherigen Eigenschaften, während diese Funktion in Macsyma kumulativ wirkt. Die (zusätzliche) Annahme $x > 0$ führt in diesem Fall zu einem inkonsistenten System von Eigenschaften, weshalb zunächst `forget(x<0)` eingegeben werden muss.

Einzelne Funktionen sind in der Lage, aus Eigenschaften ihrer Argumente Eigenschaften der Funktionswerte abzuleiten und weiterzugeben. Allerdings ist dieses Inferenzproblem schwierig zu behandeln und über einer einigermaßen aussagekräftigen Menge von Eigenschaften algorithmisch nicht lösbar.

In Derive können ebenfalls Annahmen (positiv, nichtnegativ, reell, komplex, aus einem reellen Intervall) über die Natur einzelner Variablen getroffen werden.

In Macsyma und Reduce lässt sich außerdem die Strenge der mathematischen Umformungen durch verschiedene Schalter verändern. So kann man etwa in Reduce über den (allerdings nicht dokumentierten) Schalter `reduced` die klassischen Vereinfachungen von Wurzelsymbolen, die für komplexe Argumente zu fehlerhaften Ergebnissen führen können, zulassen. In Macsyma können die entsprechenden Schaltern wie `logexpand, radexpand` oder `triginverses` sogar drei Werte

[5] Macsyma liefert die ebenfalls korrekte Vereinfachung $\sqrt{-x}\,\sqrt{-y} - \sqrt{x}\,\sqrt{y}$.

annehmen: `false` (keine Simplifikation), `true` (bedachtsame Simplifikation) oder `all` (ständige Simplifikation).

Die mathematische Strenge der Rechnungen, die mit einem CAS allgemeiner Ausrichtung möglich sind, ist in den letzten Jahren stärker ins Blickfeld der Entwickler geraten. Die Konzepte der verschiedenen Systeme sind unter diesem Blickwinkel mehrfach geändert worden, was man an den differierenden Ergebnissen verschiedener Vergleiche, die zu unterschiedlichen Zeiten angefertigt wurden ([17, 16, 21, 22]), erkennen kann. Allerdings werden selbst innerhalb eines Systems an unterschiedlichen Stellen manchmal unterschiedliche Maßstäbe der mathematischen Strenge angelegt, insbesondere bei der Implementierung komplexerer Verfahren. Ein abschließendes Urteil ist deshalb nicht möglich.

Für eine umfassendere Betrachtung dieser Problematik, die auch tiefergehende mathematische Fähigkeiten der verschiedenen Systeme in einen Vergleich einbezieht, sei dazu auf [16] verwiesen.

4.4 Das allgemeine Simplifikationsproblem

Wir hatten gesehen, dass sich das allgemeine Simplifikationsproblem grob als **das Erkennen der semantischen Gleichwertigkeit syntaktisch verschiedener Ausdrücke** charakterisieren lässt. Wir wollen nun versuchen, dies begrifflich genauer zu fassen.

4.4.1 Die Formulierung des Simplifikationsproblems

Wir betrachten dazu die Menge $\mathcal{E}$ linguistischer Objekte, etwa regulär zusammengesetzter Ausdrücke, welche den (syntaktischen) Eingaben entsprechen, die ein CAS "versteht". Die semantische Gleichwertigkeit solcher Ausdrücke kann dann als eine (nicht notwendig effektiv berechenbare) Äquivalenzrelation $\sim$ auf $\mathcal{E}$ verstanden werden.

Als *Simplifikator* wollen wir eine effektiv berechenbare Funktion $S : \mathcal{E} \longrightarrow \mathcal{E}$ bezeichnen, die die beiden Bedingungen

$$(I) \quad S(S(t)) = S(t) \quad \text{(Idempotenz)} \qquad \text{und} \qquad (E) \quad S(t) \sim t \quad \text{(Äquivalenz)}$$

für alle $t \in \mathcal{E}$ erfüllt.

Wir hatten gesehen, dass man die Simplifikationsvorgänge in einem konkreten CAS als fortgesetzte Transformationen verstehen kann, wobei ein Arsenal von gewissen Transformationsregeln rekursiv immer wieder auf den entstehenden Zwischenausdruck angewendet wird, bis keine Ersetzungen mehr möglich sind. Diese Ersetzungen erfolgen nicht auf den eingegebenen Ausdrücken, also Elementen aus $\mathcal{E}$, sondern ihren Darstellungen als geschachtelte Listen.

Solche (geschachtelten) Listen kann man als *Binärbäume* auffassen, die rekursiv durch eine Operation `cons` aus atomaren Objekten (Funktionssymbolen, Variablen und Konstanten) und der leeren Liste `()` aufgebaut werden. `M:=cons(A,L)` erzeugt dabei einen Binärbaum M, dessen linker Teilbaum A ein atomares Objekt oder eine nicht leere Liste ist, während der rechte Teilbaum L eine beliebige (evtl. auch leere) Liste $L = (A_2\ A_3\ \ldots\ A_n)$ sein muss. Das neue Objekt M steht dann für die Liste $(A\ A_2\ A_3\ \ldots\ A_n)$.

Obwohl in praxi in diesen Listen identische Teilausdrücke an verschiedenen Stellen präsent sein können, also die wirkliche Datenstruktur die eines gerichteten kreisfreien Graphen ist, wollen wir uns der Einfachheit halber im Folgenden auf Binärbäume beschränken, indem wir solche identischen Teilausdrücke ggf. durch Duplikate ersetzen. Atomare Objekte und Binärbäume, die sich auf diesem Wege konstruieren lassen, wollen wir im Weiteren unter dem Begriff des *Parse-Baums* zusammenfassen. Atomare Objekte sind genau die Parse-Bäume der Tiefe 0. Ein Binärbaum der Tiefe > 0 ist genau dann ein Parse-Baum, wenn seine linken Blätter atomare Objekte und seine rechten Blätter leere Listen sind.

Die (oBdA umkehrbar eindeutige) Zuordnung zwischen der Menge $\mathcal{E}$ der zulässigen Zeichenketten und der Menge $\mathcal{A}$ der Parse-Bäume erfolgt durch eine (effektiv berechenbare) Funktion $p : \mathcal{E} \longrightarrow \mathcal{A}$, die *Parse-Funktion.*

Aus der Definition lassen sich sofort zwei Eigenschaften herleiten:

(1) Jeder (vollständige) Teilbaum eines Parse-Baums ist selbst wieder ein Parse-Baum.

(2) Ersetzt man in einem Parse-Baum den linken Sohn eines Knotens durch einen anderen Parse-Baum (insbesondere durch eine Variable), so erhält man wieder einen Parse-Baum.

In diesem Kontext lässt sich ein Simplifikator durch ein (endliches) Regelsystem $\mathcal{R}$ beschreiben, dessen Regeln die Gestalt

$$L(u) \longrightarrow R(u)$$

haben mit $L(u), R(u) \in \mathcal{A}$, wobei u eine Liste von formalen Parametern ist, die als linke Blätter in L bzw. R auftreten, und für alle Variablenbelegungen $u = (u_1, \ldots, u_m) \vdash U = (U_1, \ldots, U_m)$ mit Parse-Bäumen $U_1, \ldots, U_m \in \mathcal{A}$

$$p^{-1}\, L(U) \sim p^{-1}\, R(U)$$

in $\mathcal{E}$ gilt. Betrachtet man die Anwendung der einzelnen Regeln als elementaren Simplifikationsschritt, so

ist der zugehörige Simplifikator $S : \mathcal{E} \longrightarrow \mathcal{E}$ der mit p bzw. p^{-1} gekoppelte transitive Abschluss der Ersetzungsrelationen aus $\mathcal{R}$.

4.4.2 Termination

S ist somit *effektiv*, wenn die Implementierung von $\mathcal{R}$ die folgenden beiden Bedingungen erfüllt:

(Matching) Es lässt sich effektiv entscheiden, ob es zu gegebenem Parse-Baum B und Regel $(L(u) \longrightarrow R(u))$ eine solche Variablenbelegung $u \vdash U$ gibt, dass $L(U)$ ein Teilbaum von B ist (und die Ersetzung $B^{(1)} := B|_{L(U) \longrightarrow R(U)}$ kann effektiv vorgenommen werden).

(Termination) Nach endlich vielen Schritten $B, B^{(1)}, B^{(2)}, \ldots, B^{(N)}$ ist keine Ersetzung mehr möglich.

Während sich die erste Bedingung offensichtlich durch entsprechendes Absuchen des Baumes B unabhängig vom gegebenen Regelsystem erfüllen lässt, hängt die zweite Bedingung wesentlich vom Regelsystem $\mathcal{R}$ ab.

Am einfachsten lässt sich die Termination sichern, wenn es eine (teilweise) Ordnungsrelation $\leq$ auf $\mathcal{E}$ gibt, bzgl. derer die rechten Seiten der Regeln "kleiner" als die linken Seiten sind. In diesem Sinne ist dann auch $S(t)$ "einfacher" als t, d.h. es gilt

$$(S) \quad S(t) \leq t \qquad \text{für alle } t \in \mathcal{E}.$$

Die Termination ist gewährleistet, wenn zusätzlich gilt:

(1) $\leq$ ist wohlfundiert.

(2) Für jede Regel $(L(u) \to R(u)) \in \mathcal{R}$ und jede Belegung $u \vdash U$ gilt $p^{-1}\, L(U) > p^{-1}\, R(U)$.

Die zweite Eigenschaft sichert, dass in einem elementaren Simplifikationsschritt die Vereinfachung zu einem "kleineren" Ausdruck führt, die erste, dass eine solche Simplifikationskette nur endlich viele Schritte haben kann. Jede Ordnung, die (2) erfüllt und für die ein Kompliziertheitsmaß $l : \mathcal{E} \longrightarrow \mathbf{N}$ mit

$$e_1 > e_2 \;\Rightarrow\; l(e_1) > l(e_2) \quad \text{für } e_1, e_2 \in \mathcal{E}$$

(wie z.B. Anzahl der verwendeten Zeichen, Zahl der Klammern, der Additionen usw.) existiert, ist wohlfundiert und führt damit zu einem terminierenden Simplifikator. Obwohl solche Ordnungen auf $\mathcal{E}$ (die ja insbesondere mit der Unifikation konsistent sein müssen) nur für sehr einfache Regelsysteme explizit angegeben werden können, spielen solche Kompliziertheitsmaße eine zentrale Rolle beim Beweis der Termination einfacher Regelsysteme.

4.4.3 Abhängigkeit des Ergebnisses vom Simplifikationspfad

Ein Simplifikationsprozess kann wesentlich von den gewählten Simplifikationsschritten abhängen, wenn für einen (Zwischen-)Ausdruck mehrere Simplifikationsmöglichkeiten und damit Simplifikationspfade existieren. Dies kann nicht nur Einfluss auf die Rechenzeit, sondern auch auf das Ergebnis selbst haben.

Als *kanonische Form* bezeichnet man einen Simplifikator, der zusätzlich der Bedingung

$$(C) \quad s \sim t \Rightarrow S(s) = S(t) \qquad \text{für alle } s, t \in \mathcal{E}$$

genügt. Für einen solchen Simplifikator führen wegen (E) alle möglichen Simplifikationspfade zum selben Ergebnis. $S(t)$ bezeichnet man deshalb auch als **die** *kanonische Form* des Ausdrucks $t \in \mathcal{E}$. Ein solcher Simplifikator hat eine in Bezug auf unsere Ausgangsfrage starke Eigenschaft:

> Ein kanonischer Simplifikator erlaubt es, eine Klasse semantisch äquivalenter Ausdrücke an Hand des (syntaktischen) Aussehens der entsprechenden kanonischen Form eindeutig zu identifizieren.

Ein solcher kanonischer Simplifikator kann deshalb insbesondere dazu verwendet werden, die semantische Äquivalenz von zwei gegebenen Ausdrücken zu prüfen, d.h. das **Identifikationsproblem** zu lösen: Zwei Ausdrücke sind genau dann äquivalent, wenn ihre kanonischen Formen (Zeichen für Zeichen) übereinstimmen.

Solche Simplifikatoren existieren nur für spezielle Klassen von Ausdrücken $\mathcal{E}$. Deshalb ist auch die folgende Abschwächung dieses Begriffs von Interesse. Nehmen wir an, dass $\mathcal{E}/\sim$, wie in den meisten Anwendungen in der Computeralgebra, die Struktur einer additiven Gruppe trägt, d.h. ein spezielles Symbol $0 \in \mathcal{E}$ für das Nullelement sowie eine Funktion $M : \mathcal{E} \times \mathcal{E} \longrightarrow \mathcal{E}$ existiert, die die Differenz zweier Ausdrücke (effektiv syntaktisch) aufzuschreiben vermag. Letzeres bedeutet insbesondere, dass

$$s \sim t \Leftrightarrow M(s,t) \sim 0$$

gilt. Als *Normalformoperator* bezeichnen wir dann einen Simplifikator S, für den zusätzlich

$$(N) \quad t \sim 0 \Rightarrow S(t) = 0 \qquad \text{für alle } t \in \mathcal{E}$$

gilt, d.h. jeder Nullausdruck $t \in \mathcal{E}$ wird durch S auch als solcher erkannt.

Diese Forderung ist schwächer als die der Existenz einer kanonischen Form: Es kann sein, dass für zwei Ausdrücke $s, t \in \mathcal{E}$, die keine Nullausdrücke sind, zwar $s \sim t$ gilt, diese jedoch zu verschiedenen Normalformen simplifizieren.

> Gleichwohl können wir mit einem solchen Normalform-Operator S ebenfalls das Identifikationsproblem lösen, denn zwei **vorgegebene** Ausdrücke s und t sind offensichtlich genau dann semantisch äquivalent, wenn ihre Differenz zu Null vereinfacht werden kann:
>
> $$s \sim t \Leftrightarrow S(M(s,t)) = 0.$$

4.5 Simplifikation polynomialer und rationaler Ausdrücke

Wir hatten bei der Betrachtung der Fähigkeiten eines CAS bereits gesehen, dass sie besonders gut in der Lage sind, polynomiale und rationale Ausdrücke zu vereinfachen. Der Grund dafür ist die Tatsache, dass in der Menge dieser Ausdrücke kanonische bzw. normale Formen existieren.

4.5.1 Polynome in distributiver Darstellung

Betrachten wir dazu den Polynomring $S := R[x_1, \ldots, x_n]$ in den Variablen $X = (x_1, \ldots, x_n)$ über dem Grundring R. Wir hatten gesehen, dass solche Polynome $f \in S$ in expandierter Form als Summe von Monomen $f = \sum_a c_a X^a$ dargestellt werden, wobei $a = (a_1, \ldots, a_n) \in \mathbf{N}^n$ ein Multiindex ist und X^a kurz für $x_1^{a_1} \cdots x_n^{a_n}$ steht. Eine solche Darstellung kann in den meisten CAS durch die Funktion `expand` erzeugt werden. REDUCE verwendet sie standardmäßig für die Darstellung von Polynomen.

Diese Darstellung ist eindeutig, d.h. eine kanonische Form für Polynome $f \in S$, wenn für die Koeffizienten, also die Elemente aus R, eine solche kanonische Form existiert und die Reihenfolge der Summanden festgelegt ist. Zur Festlegung der Reihenfolge definiert man gewöhnlich eine Ordnung auf $T(X) = \{X^a : a \in \mathbf{N}^n\}$, dem *Monoid der Terme* in $(x_1, \ldots, x_n)$.

Als *distributive Darstellung* eines Polynoms $f \in S$ bzgl. einer solchen Ordnung bezeichnet man eine Darstellung $f = \sum_a c_a X^a$, in der die Summanden paarweise verschiedene Terme enthalten, diese in fallender Reihenfolge angeordnet sind und die einzelnen Koeffizienten in ihre kanonische Form gebracht wurden. In dieser Darstellung ist die Addition von Polynomen besonders effizient ausführbar. Ist die gewählte Ordnung darüberhinaus *monoton*, d.h. gilt

$$s < t \Rightarrow s \cdot u < t \cdot u \quad \text{für alle } s, t, u \in T(X),$$

so kann man auch die Multiplikation recht effektiv ausführen, da dann beim gliedweisen Multiplizieren einer geordneten Summe mit einem Monom die Summanden geordnet bleiben. Wohlfundierte Ordnungen mit dieser Zusatzeigenschaft bezeichnet man als *Termordnungen.*

Zusammenfassend können wir folgenden Satz formulieren:

Satz 3 *Existiert auf R eine kanonische Form, dann ist die distributive Darstellung von Polynomen $f \in S$ mit Koeffizienten in kanonischer Form eine kanonische Form auf S.*

Der Beweis ist offensichtlich. Damit kann in Polynomringen über gängigen Grundbereichen wie den ganzen oder rationalen Zahlen, für die kanonische Formen existieren, effektiv gerechnet werden.

4.5.2 Polynome in rekursiver Darstellung

Als *rekursive Darstellung* bezeichnet man die Darstellung von Polynomen aus S als Polynome in x_n mit Koeffizienten aus $R[x_1, \ldots, x_{n-1}]$, wobei die Summanden nach fallenden Potenzen von x_n angeordnet und die Koeffizienten rekursiv nach demselben Prinzip dargestellt sind. Da die rekursive Darstellung für $n = 1$ mit der distributiven Darstellung zusammenfällt, führt die rekursive Natur des bewiesenen Satzes unmittelbar zu folgendem

Korollar 1 *Existiert auf R eine kanonische Form, dann ist auch die rekursive Darstellung von Polynomen $f \in S$ mit Koeffizienten in kanonischer Form eine kanonische Form auf S.*

Die rekursive Darstellung des in distributiver kanonischer Form gegebenen Polynoms

$$f := 2\,x^3\,y^2 + 3\,x^2\,y^2 + 5\,x^2\,y - x\,y^2 - 3\,x\,y + x - y - 2$$

als Element von $R[y][x]$ ist

$$(2y^2)\,x^3 + (3y^2 + 5y)\,x^2 + (-y^2 - 3y + 1)\,x + (-y - 2)$$

4.5.3 Polynome in faktorisierter Darstellung

Der Leser wird sich fragen, ob Ähnliches auch für die *faktorisierte Darstellung* von Polynomen zutrifft. Schließlich gibt es einzelne polynomiale Ausdrücke, wie etwa

$$u := \left(x - xy + x^2 + (x - y + 1)\left(x^2 + 3\right)\right)\left(-9y - 3y^2 + y^3 + x^4\left(2y + y^2 + 1\right) - 5\right)$$

deren distributive Darstellung

```
normal(u);
```

$$\begin{aligned}&-20x - 12y - 31xy - 10x^2 - 5x^3 + 18y^2 + 3x^4 + 12y^3 + 4x^5 - 3y^4 + 2x^6 + x^7 - 3xy^2\\&\quad -13x^2y + 7xy^3 - 9x^3y - xy^4 + 3x^4y + 7x^5y + 3x^6y + 2x^7y + 3x^2y^2 + 5x^2y^3\\&\quad -3x^3y^2 - x^2y^4 + x^3y^3 - 3x^4y^2 - 3x^4y^3 + 2x^5y^2 - x^5y^3 - x^6y^3 + x^7y^2 - 15\end{aligned}$$

deutlich umfangreicher als eine vollständige Faktorzerlegung ist:

```
factor(u);
```

$$\left(x + x^2 + 3\right)(x - y + 1)(y + 1)^2\left(y + x^4 - 5\right)$$

Zunächst ist festzuhalten, dass das Rechnen mit Polynomen in faktorisierter Form mit wesentlich mehr Rechenaufwand verbunden ist als in distributiver oder rekursiver Form, da bei *jeder* Addition zweier Polynome eine (wenigstens teilweise) Faktorisierung notwendig wird. Das kann eine sehr zeitaufwändige Prozedur sein. Da sich außerdem Primfaktoren nur eindeutig bis auf eine Einheit des jeweiligen Integritätsbereichs bestimmen lassen, wird zur Herstellung einer kanonischen Form selbst bei Gültigkeit des Satzes von der eindeutigen Primfaktorzerlegung eine zusätzliche Normierung notwendig.

4.5.4 Rationale Funktionen

Wir hatten bereits mehrfach gesehen, dass CAS hervorragend in der Lage sind, auch rationale Funktionen zu vereinfachen. Allerdings ist es in einigen Beispielen besser, die spezielle Simplifikationsstrategie `normal` zu verwenden als den allgemeinen Ansatz `simplify`, wie folgendes Beispiel (MuPAD) demonstriert:

```
u:= a^3/((a-b)*(a-c)) + b^3/((b-c)*(b-a)) + c^3/((c-a)*(c-b));
```

$$\frac{a^3}{(a-b)(a-c)} + \frac{b^3}{(b-a)(b-c)} + \frac{c^3}{(c-a)(c-b)}$$

```
simplify(u);
```

$$\frac{a^3}{-ab - ac + bc + a^2} - \frac{b^3}{ab - ac + bc - b^2} + \frac{c^3}{ab - ac - bc + c^2}$$

```
normal(u);
```

$$a + b + c$$

`normal` verfährt nach folgendem einfachen Schema: Es bestimmt einen gemeinsamen Hauptnenner und überführt dann das entstehende Zählerpolynom in seine distributive Form. Auf diese Weise entsteht zwar keine kanonische Form wie im Fall der Polynome, da ja Zähler und Nenner nicht unbedingt teilerfremd sein müssen, allerdings eine Normalform, denn auf diese Weise werden Null-Ausdrücke sicher erkannt. Es gilt damit folgender

Satz 4 *Existiert auf R eine Normalform, dann liefert die beschriebene Normalisierungsstrategie eine Normalform auf dem Ring $R(x)$ der rationalen Funktionen über R.*

Eine solche Darstellung bezeichnet man deshalb auch als *rationale Normalform*. Von ihr gelangt man zu einer kanonischen Form, indem man den gcd von Zähler und Nenner ausdividiert und dann die beiden Teile des Bruches noch geeignet normiert, vgl. etwa [3]. Da die Berechnung des gcd aber im Vergleich zu den anderen arithmetischen Operationen aufwändiger sein kann, verzichtet z.B. REDUCE auf diese Berechnung, wenn nicht der Schalter `gcd` gesetzt ist:

```
(x^5-1)/(x^3-1);
```

$$\frac{x^5-1}{x^3-1}$$

```
on gcd; ws;
```

$$\frac{x^4+x^3+x^2+x+1}{x^2+x+1}$$

Die Funktionen zur Berechnung rationalen Normalformen in MAPLE, MUPAD (`normal`), MACSYMA (`rat`) und MATHEMATICA (`Together`) teilen dagegen den gcd von Zähler und Nenner aus.

4.5.5 Verallgemeinerte Kerne

Auf Grund der guten Eigenschaften, die die beschriebenen Simplifikationsverfahren polynomialer und rationaler Ausdrücke besitzen, werden sie auch auf Ausdrücke angewendet, die nur teilweise eine solche Struktur besitzen.

Betrachten wir etwa die Vereinfachung, die REDUCE automatisch an einem trigonometrischen Ausdruck vornimmt, und vergleichen ihn mit einem analogen polynomialen Ausdruck:

```
u1:=(sin(x)+cos(x))^3;
u2:=(a+b)^3;
```

$$\cos(x)^3+3\cos(x)^2\sin(x)+3\cos(x)\sin(x)^2+\sin(x)^3$$

$$a^3+3\,a^2\,b+3\,a\,b^2+b^3$$

Die innere Struktur beider Ausdrücke ist ähnlich, einzig statt der Variablen a und b stehen verallgemeinerte Variablen (sin x) und (cos x) (die Lisp-Notation von sin(x) und cos(x)).

```
lisp prettyprint prop 'u2;
((avalue scalar
    (!*sq
       ((((a . 3) . 1)
           ((a . 2) ((b . 1) . 3))
           ((a . 1) ((b . 2) . 3))
           ((b . 3) . 1))
          . 1)
       t)))

lisp prettyprint prop 'u1;
((avalue scalar
    (!*sq
       (((((cos x) . 3) . 1)
           (((cos x) . 2) (((sin x) . 1) . 3))
           (((cos x) . 1) (((sin x) . 2) . 3))
           (((sin x) . 3) . 1))
          . 1)
       t)))
```

Ein ähnliches Ergebnis liefert die Funktion `expand` bei "konservativeren" Systemen wie z.B. MuPAD:

```
u:=expand((sin(x)+cos(x))^3);
```

$$\cos(x)^3 + \sin(x)^3 + 3\cos(x)\ \sin(x)^2 + 3\cos(x)^2\ \sin(x)$$

Die interne Struktur als rationaler Ausdruck kann mit `rationalize` bestimmt werden:

```
rationalize(u);
```

```
   3      3           2       2
D9  + D10  + 3 D9 D10  + 3 D9  D10, {D9 = cos(x), D10 = sin(x)}
```

In beiden Fällen entstehen polynomiale Ausdrücke, aber nicht in (freien) Variablen, sondern in allgemeineren Ausdrücken, wie in diesem Fall `sin(x)` und `cos(x)`, die für die vorzunehmende Normalform-Expansion als algebraisch unabhängig angenommen werden. Die zwischen ihnen bestehende algebraische Relation $\sin(x)^2 + \cos(x)^2 = 1$ muss ggf. durch andere Simplifikationsmechanismen zur Wirkung gebracht werden. Solche allgemeineren Ausdrücke werden als *Kerne* bezeichnet.

Ein Kern kann dabei ein Symbol oder ein Ausdruck mit einem nicht-arithmetischen Funktionssymbol als Kopf sein.

Hier noch ein etwas komplizierteres Beispiel. Mit der Funktion `indets` (Maple und MuPAD) können Sie die Kerne bestimmen, nach denen der folgende Ausdruck als rationale Funktion betrachtet wird.

```
u:=(exp(x)*cos(x)+cos(x)*sin(x)^4+2*cos(x)^3*sin(x)^2+cos(x)^5)/
(x^2-x^2*exp(-2*x))-(exp(-x)*cos(x)+cos(x)*sin(x)^2+cos(x)^3)/
(x^2*exp(x)-x^2*exp(-x));
```

$$\frac{\cos x\ \sin^4 x + 2\ \cos^3 x\ \sin^2 x + \cos^5 x + e^x\ \cos x}{x^2 - x^2\, e^{-2\,x}} - \frac{\cos x\ \sin^2 x + \cos^3 x + e^{-x}\ \cos x}{x^2\, e^x - x^2\, e^{-x}}$$

```
indets(u,RatExpr); # MuPAD #
```

```
{x, cos(x), sin(x), exp(x), exp(-x), exp(-2 x)}
```

Die Wirkung von `normal` ist entsprechend. Insbesondere werden die Terme $\exp(x), \exp(-x)$ und $\exp(-2x)$ als eigenständige Kerne behandelt und deshalb nicht zusammengefasst.

```
v:=normal(u);
```

$$\frac{\left(\begin{array}{c}\cos(x)^3 + \cos(x)\exp(-x) + \cos(x)\sin(x)^2 - \cos(x)\exp(x)^2 - \cos(x)^5\exp(x) + \\ \cos(x)\exp(x)\exp(-x) - \cos(x)\sin(x)^4\exp(x) - \cos(x)^3\exp(-2x) + \\ \cos(x)^5\exp(-x) - \cos(x)\exp(-x)\exp(-2x) - \cos(x)\sin(x)^2\exp(-2x) + \\ \cos(x)\sin(x)^4\exp(-x) - 2\cos(x)^3\sin(x)^2\exp(x) + 2\cos(x)^3\sin(x)^2\exp(-x)\end{array}\right)}{x^2\exp(-x) - x^2\exp(x) + x^2\exp(x)\exp(-2x) - x^2\exp(-x)\exp(-2x)}$$

Maple belässt dabei den Nenner in faktorisierter Form, was aus algorithmischer Sicht vorteilhafter ist und die Normalformeigenschaft nicht zerstört. `normal(u1,expanded)` expandiert wie `expand` den Nenner, aber auch die verschiedenen exp-Kerne, was die Zahl der Kerne reduziert:

$$\frac{\left(\begin{array}{c}\cos(x)(e^x)^3 + \cos(x)(\sin(x))^4(e^x)^2 + (\cos(x))^5(e^x)^2 + \\ 2\,(\cos(x))^3(\sin(x))^2(e^x)^2 - \cos(x)(\sin(x))^2e^x - (\cos(x))^3e^x - \cos(x)\end{array}\right)}{-x^2 + x^2(e^x)^2}$$

Dasselbe Ergebnis erreicht man in MuPAD durch explizite Anwendung von `expand` auf Zähler und Nenner.

```
v2:=normal(map(u1,expand));
```

$$\frac{\left(\begin{array}{c} e^{2\,x}\cos(x)^5+2\,e^{2\,x}\cos(x)^3\sin(x)^2-e^x\cos(x)^3+ \\ e^{3\,x}\cos(x)+e^{2\,x}\cos(x)\sin(x)^4-e^x\cos(x)\sin(x)^2-\cos(x) \end{array}\right)}{e^{2\,x}\,x^2-x^2}$$

Ersetzt man noch zusätzlich $\sin(x)^2$ durch $1-\cos(x)^2$, so ergibt sich eine besonders einfache Form des Ausdrucks:

```
simplify(v2);
```

$$\frac{e^x\cos(x)+\cos(x)}{x^2}$$

Dasselbe Ergebnis liefert MuPAD mit `simplify(u)` auch aus dem Originalausdruck, während Maple die Kerne nicht alle eliminieren kann:

```
simplify(u);
```

$$-\frac{\cos(x)\left(e^{2\,x}-2+e^x-2\,e^{-x}+e^{-3\,x}+e^{-2\,x}\right)}{x^2\left(-1+e^{-2\,x}\right)\left(e^x-e^{-x}\right)}$$

Polynomiale und rationale Ausdrücke in solchen Kernen spielen eine wichtige Rolle im Simplifikationsdesign aller CAS. Sowohl die Herstellung rationaler Normalformen und anschließende Anwendung weitergehender Vereinfachungen wie in obigem Beispiel als auch die gezielte Umformung anderer funktionaler Abhängigkeiten in rationale wie etwa beim Expandieren trigonometrischer Ausdrücke werden angewendet.

System	Rationale Normalform bilden
Macsyma	`rat(u)`
Maple	`normal(u)`
Mathematica	`Together[u]`
MuPAD	`normal(u)`
Reduce	wird standardmäßig verwendet

Tabelle 2: Zur Berechnung von rationalen Normalformen in verschiedenen CAS

Macsyma und Reduce unterteilen ihre Simplifikationssysteme sogar in zwei Teile; einen fest eingebrannten Teil, der die als "standard quotients" (Reduce) bezeichneten rationalen Ausdrücke in Kernen verwaltet und einen regelbasierten, der die Beziehungen zwischen nicht-polynomialen Funktionssymbolen erfasst. Rationale Ausdrücke werden dabei in einer teilweise compilierten Form abgelegt wie wie im Beispiel oben gesehen haben.

In Macsyma liegt eine spezielle Schicht zwischen dieser als *Canonical Rational Expression* (CRE) bezeichneten Darstellung und der defaultmäßig verwendeten, so dass die interne Repräsentation meist vor dem Nutzer verborgen bleibt. Liegt ein Ausdruck als CRE vor, so erkennt man das an einer speziellen Markierung `/R/` in der Ausgabe.

Die anderen CAS verfahren intern ähnlich, geben aber Details nicht so offenherzig preis.

4.6 Simplifizieren trigonometrischer Ausdrücke und Regelsysteme

Die verschiedenen Additionstheoreme zeigen, dass zwischen den trigonometrischen Funktionen eine große Zahl von Abhängigkeiten bestehen. Die trigonometrischen Ausdrücke bilden damit eine Klasse, in der besonders viele verschiedene Darstellungen möglich und für die verschiedensten Zwecke auch nützlich sind.

Wir wollen diesen Effekt zunächst an einigen Beispielen studieren, in denen vom entsprechenden CAS selbst solche Umformungen eingesetzt werden. Dazu betrachten wir verschiedene Ausdrücke, die trigonometrische Funktionen enthalten, integrieren diese, bilden die Ableitung der so erhaltenen Stammfunktionen und untersuchen, ob die Systeme in der Lage sind zu erkennen, dass die so produzierten Ausdrücke mit den ursprünglichen Integranden zusammenfallen. Neben unserer eigentlichen Problematik erlaubt die Art der Antwort der einzelnen Systeme zugleich einen gewissen Einblick, wie diese die entsprechenden Integrationsaufgaben lösen.

Als Beipiele betrachten wir die Funktionen

$$\begin{aligned}
f_1 &:= \sin(3x)\,\sin(5x),\\
f_2 &:= \cos\left(\tfrac{x}{2}\right)\,\cos\left(\tfrac{x}{3}\right),\\
f_3 &:= \sin\left(2x-\tfrac{\pi}{6}\right)\,\cos\left(3x+\tfrac{\pi}{4}\right),\\
f_4 &:= \frac{1}{\cos^4(x)},\\
f_5 &:= \frac{1}{\sin^2(x)\,(1-\cos(x))},\\
f_6 &:= \frac{1}{\sin(2x)\,(3\tan(x)+5)}
\end{aligned}$$

Eine typische Antwort von MAPLE lautet etwa

$$g_1 := \int f_1\,dx = \frac{\sin(2x)}{4} - \frac{\sin(8x)}{16}$$

$$g_1' := \frac{\cos(2x)}{2} - \frac{\cos(8x)}{2}\ ,$$

woran wir das Vorgehen gut erkennen: Zur Berechnung des Integrals wurde das Produkt von Winkelfunktionen durch Anwenden der entsprechenden Additionstheoreme in eine Summe einzelner Winkelfunktionen mit Vielfachen von x als Argument umgewandelt. Eine solche Summe kann man leicht integrieren. Wollen wir aber jetzt prüfen, dass $f_1 = g_1'$ gilt, so sind diese Winkelfunktionen mit verschiedenen Argumenten zu vergleichen, wozu es notwendig ist, die Vielfachen von x als Argument wieder aufzulösen, also die entsprechenden Regeln in der anderen Richtung anzuwenden.

Welcher Art von Simplifikationsregeln werden in beiden Fällen verwendet? Für die erste Aufgabe sind Produkte in Summen zu verwandeln. Das erreicht man durch (mehrfaches) Anwenden der Regeln **Produkt-Summe**

$$\begin{aligned}
\sin(x)\sin(y) &\Rightarrow \tfrac{1}{2}(\cos(x-y)-\cos(x+y))\\
\cos(x)\cos(y) &\Rightarrow \tfrac{1}{2}(\cos(x-y)+\cos(x+y))\\
\sin(x)\cos(y) &\Rightarrow \tfrac{1}{2}(\sin(x-y)+\sin(x+y))
\end{aligned}$$

Diese Regeln sind invers zu den Regeln **Summe-Produkt**, die man beim Expandieren von Winkelfunktionen, deren Argumente Summen oder Differenzen sind, anwendet:

$$\begin{aligned}
\sin(x+y) &\Rightarrow \sin(x)\cos(y)+\cos(x)\sin(y)\\
\cos(x+y) &\Rightarrow \cos(x)\cos(y)-\sin(x)\sin(y)\\
\sin(-x) &\Rightarrow -\sin(x)\\
\cos(-x) &\Rightarrow \cos(x)
\end{aligned}$$

Bei der Formulierung des letzten Regelsatzes haben wir bereits berücksichtigt, dass REDUCE (wie im übrigen auch die anderen CAS) eine Differenz $x-y$ als Summe $x+(-y)$ darstellt, wir also keine Simplifikationsregeln für eine *binäre* Operation MINUS, wohl aber für die *unäre* Operation MINUS angeben müssen.

Zur Anwendung der Produkt-Summe-Regeln ist, wie oben beschrieben, der Parse-Baum des zu vereinfachenden Ausdrucks darauf zu untersuchen, ob in ihm Muster der Form

```
        (* (sin x) (cos y))
oder    (* (cos x) (cos y))
oder    (* (sin x) (sin y))
(oder   (* (cos y) (sin x)))
```

vorkommen.

In einer solchen *Regeldefinition* können x und y sowohl für die Literale selbst stehen als auch als formale Parameter Platzhalter für beliebige Teilausdrücke sein, die bei der Regelanwendung in einem Unifikationsprozess entsprechend zu matchen sind. Regeln ohne formale Parameter bezeichnet man als *literale* (MACSYMA) oder *spezielle* (MATHEMATICA), andere Regeln als *semantische* (MACSYMA) oder *allgemeine* (MATHEMATICA) Regeln. Im folgenden Regelsystem für `log` in REDUCE wird dieser Unterschied deutlich:

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

Die ersten beiden Regeln sind spezielle Regeln, denn dort steht `e` für das Symbol e, die Basis des natürlichen Logarithmus. Die beiden letzten Regeln dagegen sind allgemeine Regeln, in denen `~x` für einen formalen Parameter steht, wobei in der letzten Regel x sogar an zwei Stellen des Musters mit demselben Teilausdruck matchen muss.

Diese Feinheiten von Regelsystemen sind in der Form besonders gut in REDUCE und MATHEMATICA ausgeprägt. Wir wollen deshalb für diesen Abschnitt das System REDUCE zu Grunde legen.

Formulieren wir zunächst die obigen Regeln in REDUCE-Notation. Die Tilde vor einer Variablen bedeutet, dass sie als formaler Parameter verwendet wird.

```
trigsum0:={ % Produkt-Summen-Regeln
        cos(~x)*cos(~y) => 1/2 * ( cos(x+y) + cos(x-y)),
        sin(~x)*sin(~y) => 1/2 * (-cos(x+y) + cos(x-y)),
        sin(~x)*cos(~y) => 1/2 * ( sin(x+y) + sin(x-y))};
trigexpand0:={ % Summen-Produkt-Regeln
        sin(~x+~y) => sin x * cos y + cos x * sin y,
        cos(~x+~y) => cos x * cos y - sin x * sin y};
```

Die Regeln $\sin(-x) = -\sin(x)$ und $\cos(-x) = \cos(x)$ werden nicht benötigt, da dieser Teil der Simplifikation (gerade/ungerade Funktionen) als spezielle *Eigenschaft* der jeweiligen Funktion vermerkt ist[6] und genau wie die Vereinfachung rationaler Funktionen gesondert behandelt wird.

REDUCE erlaubt es – analog dem Substitutionsoperator für lokale Wertzuweisungen – mit dem Infixoperator `where` solche Regelsysteme lokal auf einzelne Ausdrücke anzuwenden.

Wenden wir die Regeln `trigsum0` auf einen polynomialen Ausdruck in den Kernen `sin(x)` und `cos(x)` an, so sollten am Ende alle Produkte trigonometrischer Funktionen zugunsten von Mehrfachwinkelargumenten aufgelöst sein, womit der Ausdruck in eine besonders einfach zu integrierende Form überführt wird. Die Termination dieser Simplifikation ergibt sich daraus, dass bei jeder Regelanwendung in jedem Ergebnisterm die Zahl der Faktoren um Eins geringer ist als im Ausgangsterm.

Leider klappt das nicht so, wie erwartet, wie etwa das Beispiel

```
sin(x)*sin(2x)*cos(3x) where trigsum0;
```

[6] wie man mit `flagp('sin,'odd)` und `flagp('cos,'even)` erkennt

$$\frac{-\cos(6\,x)+\cos(4\,x)-2\,\sin(x)^2}{4}$$

zeigt. Hier ist der Ausdruck $\sin(x)^2$ nicht weiter vereinfacht worden, weil er nicht die Gestalt `(* (sin A) (sin B))`, sondern `(expt (sin A) 2)` hat. Wir müssen also noch Regeln hinzufügen, die es erlauben, auch $\sin(x)^n$ und analog $\cos(x)^n$ zu vereinfachen.

Solche Regeln können wir aus der Beziehung $\sin^2(x) = \frac{1-\cos(2x)}{2}$ (analog für $\cos(x)^2$) ableiten, indem wir einen solchen Faktor von $\sin(x)^n$ abspalten und auf die verbleibende Potenz $\sin(x)^{n-2}$ dieselbe Regel rekursiv anwenden. Ein derartiges rekursives Vorgehen ist typisch für Regelsysteme und erlaubt es, Simplifikationen zu definieren, deren Rekursionstiefe von einem der formalen Parameter (wie hier n) abhängig ist. Allerdings müssen wir dazu *konditionierte Regeln* formulieren, denn obige Ersetzung darf nur für ganzzahlige $n > 1$ angewendet werden, um den Abbruch der Rekursion zu sichern. Auch das ist in REDUCE möglich. Eine entsprechende Erweiterung der Regeln `trigsum` sähe dann so aus:

```
trigsum1:={
        sin(~x)^(~n) => (1-cos(2x))/2*sin(x)^(n-2) when fixp n and (n>1),
        cos(~x)^(~n) => (1+cos(2x))/2*cos(x)^(n-2) when fixp n and (n>1)};
```

In REDUCE können wir diese Regeln jedoch weiter vereinfachen, wenn wir den

Unterschied zwischen algebraischen und exakten Ersetzungsregeln

beachten. Betrachten wir dazu die distributive Normalform von $(a+1)^5$, also den Ausdruck

$$A = a^5 + 5\,a^4 + 10\,a^3 + 10\,a^2 + 5\,a + 1,$$

und ersetzen in ihm a^2 durch x. MATHEMATICA liefert als Ergebnis

```
A /. (a^2->x)
```

$$1 + 5\,a + 10\,a^3 + 5\,a^4 + a^5 + 10\,x$$

während REDUCE

```
(a+1)^5 where (a^2 => x);
```

$$a\,x^2 + 10\,a\,x + 5\,a + 5\,x^2 + 10\,x + 1$$

zurückgibt.

Wir sehen, dass MATHEMATICA nur solche Muster ersetzt hat, die *exakt* auf den Ausdruck a^2 passen, während REDUCE erkennt, dass auch in einem Ausdruck $a^k, k > 2$ der Faktor a^2 enthalten ist, also die eine Regel durch eine Serie von Regeln $(a^{k+2} \Rightarrow x \cdot a^k), k \geq 0$ ersetzt hat.

Unser gesamtes Regelsystem `trigsum` für REDUCE lautet damit:

```
trigsum:={ % Produkt-Summen-Regeln
        sin(~x)*sin(~y) => 1/2*(cos(x-y)-cos(x+y)),
        sin(~x)*cos(~y) => 1/2*(sin(x-y)-sin(x+y)),
        cos(~x)*cos(~y) => 1/2*(cos(x-y)+cos(x+y)),
        cos(~x)^2 => (1+cos(2x))/2,
        sin(~x)^2 => (1-cos(2x))/2};
```

Die Anwendung dieses Regelsystems **Produkt-Summe** führt auf eine kanonische Darstellung polynomialer trigonometrischer Ausdrücke, ist also für Simplifikationszwecke hervorragend geeignet. Genauer gilt folgender

Satz 5 *Sei R ein Unterkörper von $\mathbf{C}$. Dann kann jeder polynomiale Ausdruck $P(\sin(x), \cos(x))$ mit Koeffizienten aus R mit obigem Regelsystem* `trigsum` *in einen Ausdruck der Form*

$$\sum_{k>0} (a_k \sin(kx) + b_k \cos(kx)) + c$$

mit $a_k, b_k, c \in R$ verwandelt werden.

Diese Darstellung ist eindeutig. Genauer: Hat R eine kanonische oder Normalform, so kann diese Darstellung zu einer kanonischen bzw. Normalform für die Klasse der betrachteten Ausdrücke verfeinert werden.

BEWEIS : Da das Regelsystem alle Produkte und Potenzen von Winkelsystemen ersetzt und sich in jedem Transformationsschritt die Anzahl der Multiplikationen verringert, ist nur die Eindeutigkeit der Darstellung zu zeigen. Die Koeffizienten a_k, b_k, c in einer Darstellung

$$f(x) = \sum_{k>0} (a_k \sin(kx) + b_k \cos(kx)) + c$$

kann man aber wie in der Theorie der Fourierreihen zurückgewinnen. Berechnen wir etwa für ein festes ganzzahliges $n > 0$ das Integral

$$\begin{aligned}
&2 \int_0^{2\pi} f(x) \sin(nx)\, dx \\
&\quad = \textstyle\sum_{k>0} a_k \int_0^{2\pi} 2 \sin(kx) \sin(nx)\, dx \\
&\qquad\qquad + \textstyle\sum_{k>0} b_k \int_0^{2\pi} 2 \cos(kx) \sin(nx)\, dx + 2c \int_0^{2\pi} \sin(nx)\, dx \\
&\quad = \textstyle\sum_{k>0} a_k \int_0^{2\pi} (\cos((k-n)x) - \cos((k+n)x))\, dx \\
&\qquad\qquad + \textstyle\sum_{k>0} b_k \int_0^{2\pi} (\sin((n-k)x) + \sin((n+k)x))\, dx \\
&\quad = 2\pi\, a_n,
\end{aligned}$$

so sehen wir, dass $f(x)$ jeden Koeffizienten a_n eindeutig bestimmt. Wir haben dabei von unseren Formeln Produkt-Summe sowie den Beziehungen

$$\begin{aligned}
\int_0^{2\pi} \sin(m\,x)\, dx &= 0 \\
\int_0^{2\pi} \cos(m\,x)\, dx &= \begin{cases} 0 & \text{für } m \neq 0 \\ 2\pi & \text{für } m = 0 \end{cases}
\end{aligned}$$

Gebrauch gemacht. Analog ergeben sich die Koeffizienten b_n und c aus $\int_0^{2\pi} f(x) \cos(nx)\, dx$ bzw. $\int_0^{2\pi} f(x)\, dx$. □

Neben der Umwandlung von Potenzen der Winkelfunktionen in Summen mit Mehrfachwinkel-Argumenten ist an anderen Stellen, etwa beim Lösen goniometrischer Gleichungen, auch die umgekehrte Transformation von Interesse, da sie die Zahl der verschiedenen Funktionsausdrücke, die als Kerne in diesen Ausdruck eingehen, verringert. Auf diese Weise liefert die anschließende Berechnung der rationalen Normalform oft ein besseres Ergebnis.

Mathematisch kann man die entsprechenden Formeln aus der *Moivreschen Formel* für komplexe Zahlen und den Potenzgesetzen herleiten: Aus

$$\cos(nx) + i \cdot \sin(nx) = e^{inx} = (e^{ix})^n = (\cos(x) + i \cdot \sin(x))^n$$

und den binomischen Formeln ergibt sich durch Vergleich der Real- und Imaginärteile unmittelbar

$$\cos(nx) = \sum_{2k \leq n} (-1)^k \binom{n}{2k} \sin(x)^{2k} \cos(x)^{n-2k}$$

und

$$\sin(nx) = \sum_{2k<n} (-1)^k \binom{n}{2k+1} \sin(x)^{2k+1} \cos(x)^{n-2k-1}$$

Der Umbau eines Ausdrucks nach dem Unifizieren ist nicht auf einfache arithmetische Kombinationen beschränkt, sondern kann beliebige, auch selbst definierte Funktionen heranziehen, mit welchen die rechte Seite der Regelanwendung aus den unifizierten Teilen zusammengebaut wird. In REDUCE etwa könnte man für die zweite Formel eine Funktion `Msin` als

```
procedure Msin(n,x);
  begin scalar k,l;
  while (2*k<n) do
  << l:=l+(-1)^k*binomial(n,2k+1)*sin(x)^(2k+1)*cos(x)^(n-2k-1);
     k:=k+1;
  >>;
  return l;
end;
```

vereinbaren und in der Regel

```
sin(~n*~x) => Msin(n,x) when fixp n and (n>1)
```

einsetzen.

Einfacher ist es allerdings auch in diesem Fall, die Regeln aus dem Ansatz

$$\sin(kx) = \sin\left((k-1)x + x\right) = \sin\left((k-1)x\right)\cos(x) + \cos\left((k-1)x\right)\sin(x)$$

herzuleiten, den wir wieder rekursiv zur Anwendung bringen. Unser gesamtes Regelsystem hat dann die Gestalt:

```
trigexpand:={ % Produkt-Summen-Regeln
        sin(~x+~y) => sin x * cos y + cos x * sin y,
        cos(~x+~y) => cos x * cos y - sin x * sin y,
        sin(~k*~x) => sin((k-1)*x)*cos x+cos((k-1)*x)*sin x
                      when fixp k and k>1,
        cos(~k*~x) => cos((k-1)*x)*cos x-sin((k-1)*x)*sin x
                      when fixp k and k>1};
```

Diese Umformungsregeln erlauben es, komplizierte trigonometrische Ausdrücke, in deren Argumenten Summen und Mehrfachwinkel auftreten, durch trigonometrische Ausdrücke mit nur noch wenigen verschiedenen und einfachen Argumenten zu ersetzen. Hängen die Argumente nur ganzzahlig von einer Variablen x ab, so können wir das System wiederum zu einer kanonischen oder Normalform erweitern. Genauer gesagt gilt der folgende Satz:

Satz 6 *Sei R ein Unterring von $\mathbf{C}$. Dann kann man jeden polynomialen Ausdruck in $\sin(k\,x)$ und $\cos(k\,x)$, $k \in \mathbf{Z}$, mit Koeffizienten aus R durch obiges Regelsystem* `trigexpand` *und die zusätzliche Regel $\{\sin(x)^2 \Rightarrow 1 - \cos(x)^2\}$ in einen Ausdruck der Form*

$$P(\cos(x)) + \sin(x)\,Q(\cos(x))$$

verwandeln, wobei $P(z)$ und $Q(z)$ Polynome mit Koeffizienten aus R sind.

Diese Darstellung ist eindeutig. Genauer: Hat R eine kanonische oder Normalform, so kann diese Darstellung zu einer kanonischen bzw. Normalform für die Klasse der betrachteten Ausdrücke verfeinert werden.

BEWEIS : Es ist wiederum nur die Eindeutigkeit zu zeigen.

$$P_1(\cos(x)) + \sin(x)\,Q_1(\cos(x)) = P_2(\cos(x)) + \sin(x)\,Q_2(\cos(x))$$

gilt aber genau dann, wenn $(P_1 - P_2)(\cos(x)) + \sin(x)\,(Q_1 - Q_2)(\cos(x))$ identisch verschwindet. Wir haben also nur zu zeigen, dass aus der Tatsache, dass $P(\cos(x)) + \sin(x)\,Q(\cos(x))$ identisch verschwindet, bereits folgt, dass $P(z)$ und $Q(z)$ beides Nullpolynome sind. Aus $P(\cos(x)) + \sin(x)\,Q(\cos(x)) = 0$ folgt aber nach der Substitution $x = -x$ auch $P(\cos(x)) - \sin(x)\,Q(\cos(x)) = 0$ und schließlich $P(\cos(x)) = Q(\cos(x)) = 0$. Da ein nichttriviales Polynom aber nur endlich viele Nullstellen besitzt, folgt die Behauptung. □

Mit diesen beiden Strategien lassen sich die ersten drei Integrationsaufgaben, mit denen wir diesen Abschnitt begonnen hatten, zufriedenstellend lösen. Einzige Schwierigkeit ist die Vereinfachung von $f_2 - g_2'$, da

$$g_2 := \int f_2\,dx = \frac{3}{5}\sin\left(\frac{5x}{6}\right) + 3\sin\left(\frac{x}{6}\right)$$

Winkel enthält, von denen nicht auf den ersten Blick sichtbar ist, dass es sich um ganzzahlige Vielfache eines gemeinsamen Winkels handelt. Die explizite Substitution $x = 6y$, die das System darauf hinweist, hilft hier aber weiter.

Die klare Struktur der beiden Simplifikationssysteme verwendet implizit die Tatsache, dass REDUCE standardmäßig polynomiale Ausdrücke in ihre distributive Normalform transformiert. Dies muss dem Regelsystem hinzugefügt werden, wenn eine solche Simplifikation nicht automatisch erfolgt. Wir wollen diesen Aspekt an MATHEMATICA demonstrieren. Vorab sei auf einige Besonderheiten der Syntax dieses Systems hingewiesen, die zum Verständnis der Formeln unerlässlich sind. Für eine detailliertere Einführung verweisen wir auf einschlägige Systemliteratur.

MATHEMATICA unterscheidet im Gegensatz etwa zu REDUCE streng zwischen Groß- und Kleinschreibung. Die meisten Funktionsnamen beginnen jede "Silbe" mit einem Großbuchstaben. Ihre Argumentliste wird im Gegensatz zu anderen Systemen in eckige Klammern eingeschlossen. Runde Klammern dienen ausschließlich der Gruppierung von Teilausdrücken. Einstellige Funktionen wie etwa `Simplify, Together` oder `Expand` können auch in der Postfix-Notation `<Expr>//Expand` statt `Expand[<Expr>]` geschrieben werden, was der Auswertungsreihenfolge (erst die Argumente, dann der Funktionsaufruf) stärker entspricht als die übliche Präfixnotation. Für viele zweistellige Funktionen existieren Infix-Notationen in Operatorform, wie etwa für die Substitutionsoperatoren `/.` (einmalige Substitution) und `//.` (rekursive Substitution; beide unterscheiden sich genau wie Evaluationstiefe 1 und maximale Evaluationstiefe). Formale Parameter werden durch einen Unterstrich, etwa als `x_`, gekennzeichnet, dem im Gegensatz zu REDUCE weitere Information über den Typ oder eine Default-Initialisierung folgen können. Deshalb dürfen, ebenfalls im Gegensatz zu REDUCE, Unterstriche nicht in Variablennamen auftreten.

Obige Regelsysteme hätten damit in MATHEMATICA folgende Gestalt:

```
trigexpand={ (* Expandiert Winkelfunktionen *)
        Cos[x_+y_] -> Cos[x]*Cos[y] - Sin[x]*Sin[y],
        Sin[x_+y_] -> Sin[x]*Cos[y] + Cos[x]*Sin[y],
        Cos[n_Integer*x_]/; (n>1)  ->
                Cos[(n-1)*x]*Cos[x]-Sin[(n-1)*x]*Sin[x],
        Sin[n_Integer*x_]/; (n>1)  ->
                Sin[(n-1)*x]*Cos[x]+Cos[(n-1)*x]*Sin[x]};

trigsum0={ (* Verwandelt Produkte in Summen von Mehrfachwinkeln *)
        Cos[x_]*Cos[y_] -> 1/2*(Cos[x+y]+Cos[x-y]),
        Sin[x_]*Sin[y_] -> 1/2*(-Cos[x+y]+Cos[x-y]),
        Sin[x_]*Cos[y_] -> 1/2*(Sin[x+y]+Sin[x-y]),
        Sin[x_]^(n_Integer)/; (n>1) -> (1-Cos[2*x])/2*Sin[x]^(n-2) ,
        Cos[x_]^(n_Integer)/; (n>1) -> (1+Cos[2*x])/2*Cos[x]^(n-2) };
```

Wenden wir das zweite System auf $\sin(x)^7$ an, so erhalten wir nicht die aus unseren Rechnungen mit REDUCE erwartete Normalform, sondern einen teilfaktorisierten Ausdruck:

```
Sin[x]^7//.trigsum0
```

$$\frac{(1-\cos(2\,x))^3\,\sin(x)}{8}$$

Dieser Ausdruck muss erst expandiert werden, damit die Regeln erneut angewendet werden können.

```
%//Expand;
%//.trigsum0
```

$$\frac{\sin(x)}{8}+\frac{3\,(1+\cos(4\,x))\,\sin(x)}{16}-\frac{3\,(-\sin(x)+\sin(3\,x))}{16}-\frac{(1+\cos(4\,x))\,(-\sin(x)+\sin(3\,x))}{32}$$

usw., ehe nach vier solchen Schritten schlielich das Endergebnis

$$\frac{35\,\sin(x)}{64}-\frac{21\,\sin(3\,x)}{64}+\frac{7\,\sin(5\,x)}{64}-\frac{\sin(7\,x)}{64}$$

feststeht.

Wir benötigen also nach jedem Simplifikationsschritt noch die Überführung in die rationale Normalform, also die Funktionalität von `Expand`. Allerdings kann man nicht einfach

```
expandrule={x_ -> Expand[x]}
```

hinzufügen, denn diese Regel würde ja immer passen und damit das Regelsystem nicht terminieren. In Wirklichkeit ist es sogar noch etwas komplizierter. Zunächst muss für einen Funktionsaufruf wie `Expand` genau wie bei Zuweisungen unterschieden werden, ob der Aufruf bereits während der Regeldefinition (etwa, um eine komplizierte rechte Seite kompakter zu schreiben) oder erst bei der Regelanwendung auszuführen ist. Das spielte bisher keine Rolle, weil auf der rechten Seite stets Funktionsausdrücke standen. MATHEMATICA kennt zwei Regelarten, die mit `->` (Regeldefinition mit Auswertung) und `:>` (Regeldefinition ohne Auswertung) angeschrieben werden. Hier benötigen wir die zweite Sorte von Regeln:

```
expandrule={x_ :> Expand[x]}
```

Jedoch ist das Ergebnis nicht zufriedenstellend:

```
Sin[x]^7//.Join[trigsum0, expandrule]
```

$$\frac{\sin(x)^5}{2}-\frac{\cos(2\,x)\,\sin(x)^5}{2}$$

Zwar wird expandiert, aber dann mitten in der Rechnung aufgehört. Der Grund liegt in der Art, wie MATHEMATICA Regeln anwendet. Um Unendlichschleifen zu vermeiden, wird die Arbeit beendet, wenn sich nach Regelanwendung am Ausdruck nichts mehr ändert. Außerdem werden Regeln erst auf den Gesamtausdruck angewendet und dann auf Teilausdrücke. Da auf den Gesamtausdruck des Zwischenergebnisses (nur) die expandrule-Regel passt, deren Anwendung aber nichts ändert, hört MATHEMATICA deshalb auf und versucht sich gar nicht erst an Teilausdrücken.

Wir brauchen also statt der bisher betrachteten Lösungen eine zusätzliche Regel, die genau dem Distributivgesetz für Produkte von Summen entspricht und auch nur in diesen Fällen greift. Das kann man durch eine einzige weitere Regel erreichen:

```
trigsum={ (* Verwandelt Produkte in Summen von Mehrfachwinkeln *)
        Cos[x_]*Cos[y_] -> 1/2*(Cos[x+y]+Cos[x-y]),
        Sin[x_]*Sin[y_] -> 1/2*(-Cos[x+y]+Cos[x-y]),
```

```
Sin[x_]*Cos[y_] -> 1/2*(Sin[x+y]+Sin[x-y]),
Sin[x_]^(n_Integer)/; (n>1) -> (1-Cos[2*x])/2*Sin[x]^(n-2) ,
Cos[x_]^(n_Integer)/; (n>1) -> (1+Cos[2*x])/2*Cos[x]^(n-2) ,
(a_+b_)*c_ -> a*c+b*c};
```

Nun zeigt die Simplifikation das erwünschte Verhalten:

```
Sin[x]^7//.trigsum
```

$$\frac{35\,\sin(x)}{64} - \frac{21\,\sin(3\,x)}{64} + \frac{7\,\sin(5\,x)}{64} - \frac{\sin(7\,x)}{64}$$

Da die beiden Regelsysteme `trigsum` und `trigexpand` zueinander entgegengesetzt sind, kann man nicht beide gleichzeitig in einem Simplifikationssystem zulassen, ohne Zirkelschlüsse zu verursachen. Dem Nutzer stehen in den verschiedenen Systemen dafür verschiedene fest eingebaute Simplifikationsstrategien zur Verfügung. Welche Freiheiten erlauben in dieser Richtung die einzelnen Systeme?

DERIVE: Über Knöpfe kann zwischen verschiedenen eingebauten Strategien gewechselt werden. Neben Expandieren und Faktorisieren polynomialer Ausdrücke (= verschiedene polynomiale Normalformen) kann zwischen Expandieren und Sammeln von Exponenten, von logarithmischen Ausdrücken und Winkelfunktionen gewählt werden. Letzteres entspricht den beiden hier vorgestellten Strategien, wobei noch gewählt werden kann, ob in Richtung $\sin(x)$ oder $\cos(x)$ simplifiziert werden soll.

Die vier "M-Systeme" MACSYMA, MAPLE, MATHEMATICA und MUPAD stellen eine ganze Reihe von Simplifikationsprozeduren mit verschiedenen Parametern zur Verfügung, die unterschiedliche Ziele ansteuern. In MAPLE sind dies insbesondere die Befehle `expand, combine, simplify` und `convert`.

Auf trigonometrische Ausdrücke angewendet bewirken `combine` (Produkte in Winkelsummen) und `expand` (Winkelsummen in Produkte) die obigen Umformungen, während man mit `convert` bzw. `rewrite` (u.a.) trigonometrische Funktionen und deren Umkehrfunktionen in Ausdrücke mit *exp* und *log* von komplexen Argumenten umformen kann. Dies entspricht dem REDUCE-Regelsatz

```
trigexp:={
        tan(~x) => sin(x)/cos(x),
        sin(~x) => (exp(i*x)-exp(-i*x))/(2*i),
        cos(~x) => (exp(i*x)+exp(-i*x))/2};
```

Diese Umformungen sind allerdings nur für solche Systeme sinnvoll, die mit `exp`-Ausdrücken gut rechnen können, etwa wenn sie die Regel

```
exp(n~*x~) => exp(x)^n
```

für $n \in \mathbf{Z}$ anwenden, um die Zahl unterschiedlicher `exp`-Kerne überschaubar zu halten. Dies kann am Ausdruck

$$\frac{\exp(5\,x) + \exp(3\,x)}{\exp(5\,x) - \exp(3\,x)}$$

getestet werden, der dann bei der Berechnung der rationalen Normalform zum Ausdruck

$$\frac{\exp(x)^2 + 1}{\exp(x)^2 - 1}$$

vereinfacht wird. REDUCE und MATHEMATICA führen diese Umformung automatisch aus, MAPLE und MUPAD nur mit der Transformationsfunktion `expand`.

In der folgenden Tabelle sind die Namen der Funktionen in den einzelnen Systemen einander gegenübergestellt, die dieselben Transformationen wie oben beschrieben bewirken.

Wirkung	Macsyma	Maple	Mathematica	MuPAD
Argumentsummen auflösen	trigexpand	expand	TrigExpand	expand
Produkte zu Mehrfachwinkeln	trigreduce	combine	TrigReduce	combine
Trig $\mapsto$ Exp	exponentialize	convert(u,exp)	TrigToExp	rewrite(u,exp)
Exp $\mapsto$ Trig	demoivre	convert(u,trig)	ExpToTrig	rewrite(u,sincos)

Tabelle 3: Ausgewählte Transformationsfunktionen für Ausdrücke, die trigonometrische Funktionen enthalten.

MAPLE erlaubt es auch, innerhalb des Simplify-Befehls eigene Regelsysteme anzugeben, kennt dabei aber keine formalen Parameter. Statt dessen kann man für einzelne neue Funktionen den simplify-Befehl erweitern, was aber ohne interne Systemkenntnisse recht schwierig ist. Ähnlich kann man in MUPAD die Funktionsaufrufe `combine` und `expand` durch die Definition entsprechender Attribute auf andere Funktionssymbole ausdehnen.

MACSYMA und MATHEMATICA erlauben die Definition eigener Regelsysteme, mit denen man die intern vorhandenen Transformationsmöglichkeiten erweitern kann. Allerdings kann man in MACSYMA Regelsysteme nur unter großen Schwierigkeiten lokal anwenden.

Auch REDUCE kennt nur wenige eingebaute Regeln, was sich insbesondere für die Arbeit mit trigonometrischen Funktionen als nachteilig erweist. Seit der Version 3.6 gibt es allerdings das Paket `trigsimp`, das Simplifikationsroutinen für trigonometrische Funktionen zur Verfügung stellt. Wir haben aber in diesem Abschnitt gesehen, dass es nicht schwer ist, solche Regelsysteme selbst zu entwerfen. Jedoch muss der Nutzer dazu wenigstens grobe Vorstellungen über die interne Datenrepräsentation besitzen. Besonders günstig ist, dass man eigene Simplifikationsregeln in Analogie zum Substitutionsbefehl auch als lokal gültige Regeln anwenden kann. Insbesondere letzteres erlaubt es, gezielt Umformungen mit überschaubaren Seiteneffekten auf Ausdrücken vorzunehmen.

Doch kehren wir zur Berechnung der 6 Beispielintegrale zurück. Hier sind die von DERIVE berechneten Ergebnisse im einzelnen aufgelistet:

$$g_1 := \frac{\sin(2x)}{4} - \frac{\sin(8x)}{16}$$

$$g_2 := \frac{3}{5}\sin\left(\frac{5x}{6}\right) + 3\sin\left(\frac{x}{6}\right)$$

$$g_3 := -\frac{\sqrt{6}+\sqrt{2}}{40}\cos(5x) + \frac{\sqrt{6}-\sqrt{2}}{40}\sin(5x) + \frac{\sqrt{6}-\sqrt{2}}{8}\cos(x) - \frac{\sqrt{6}+\sqrt{2}}{8}\sin(x)$$

$$g_4 := \frac{2}{3}\tan(x) + \frac{\sin(x)}{3\cos^3(x)}$$

$$g_5 := -\frac{2\cos^2(x) - 2\cos(x) - 1}{3\sin(x)\,(\cos(x)-1)}$$

$$g_6 := \frac{1}{10}\ln\left(\frac{\sin(x)}{5\cos(x)+3\sin(x)}\right)$$

REDUCE geht bei der Berechnung der ersten drei Integrale anders vor: Aus der Kenntnis der Form der Antwort wird diese mit unbestimmten Koeffizienten angesetzt und die konkreten Koeffizientenwerte werden ausgerechnet. Dies erkennen wir deutlich, wenn wir Integrale mit allgemeinen Koeffizienten eingeben

```
int(sin(a*x)*sin(b*x),x);
```

$$\frac{sin(a\,x)\,cos(b\,x)\,b\;-\;sin(b\,x)\,cos(a\,x)\,a}{a^2\;-\;b^2}$$

oder

```
int(sin(a*x)*sin(b*x)*sin(c*x),x);
```

$$\begin{aligned}&\big(\sin(ax)\sin(bx)\cos(cx)a^2c+\sin(ax)\sin(bx)\cos(cx)b^2c-\sin(ax)\sin(bx)\cos(cx)c^3\\&+\sin(ax)\sin(cx)\cos(bx)a^2b-\sin(ax)\sin(cx)\cos(bx)b^3+\sin(ax)\sin(cx)\cos(bx)bc^2\\&-\sin(bx)\sin(cx)\cos(ax)a^3+\sin(bx)\sin(cx)\cos(ax)ab^2+\sin(bx)\sin(cx)\cos(ax)ac^2\\&+2\cos(ax)\cos(bx)\cos(cx)abc\big)/(a^4-2a^2b^2-2a^2c^2+b^4-2b^2c^2+c^4)\end{aligned}$$

Damit vermeidet das System in den ersten drei Fällen, Winkelfunktionen mit neuen Argumenten einzuführen, was die nachfolgende Simplifikation natürlich wesentlich vereinfacht. Die Berechtigung zu solchem Vorgehen folgt aus

Satz 7 *Sei $\{m_1, m_2, \ldots, m_k\}$ eine endliche (Multi-)Menge reeller Zahlen. Wir betrachten die Menge der Ausdrücke*

$$T := \{t_1(m_1x) * \cdots * t_k(m_kx) \;:\; t_i \in \{\sin, \cos\}\}$$

und $L = L(T)$ die Menge der reellen Linearkombinationen von Produkten aus T.

Dann gibt es zu jedem $f(x) \in L$ einen Ausdruck $g(x) \in L$ und ein $c \in \mathbf{R}$ mit

$$\int f(x)\,dx = g(x) + c\,x,$$

d.h. das Integral von $f(x)$ lässt sich bis auf einen linearen Korrekturterm als polynomialer Ausdruck in im wesentlichen denselben Kernen darstellen wie $f(x)$.

BEWEIS : Wendet man auf $f(x) = t_1(m_1x)\cdots t_k(m_kx)$ unsere Regeln `trigsum` an, so erhält man eine Linearkombination von Ausdrücken $t(\sum_{i=1}^{k}\varepsilon_i m_i x)$ mit $t \in \{\sin, \cos\}$ und $\varepsilon_i = \pm 1$. Integration solcher Ausdrücke bleibt in der Klasse außer für $\sum \varepsilon_i m_i = 0$ und $t = \cos$. Expansion der Ausdrücke $t(\sum \varepsilon_i m_i x)$, aus denen das Integral zusammengesetzt ist, längs der "Sollbruchstellen" mit `trigexpand` führt wieder in die Klasse L zurück. □

REDUCE liefert damit die folgenden Antworten:

$$g_1 := \frac{3\sin(5x)\cos(3x) - 5\sin(3x)\cos(5x)}{16}$$

$$g_2 := \frac{6\big(-2\sin(\frac{x}{3})\cos(\frac{x}{2}) + 3\sin(\frac{x}{2})\cos(\frac{x}{3})\big)}{5}$$

$$g_3 := \frac{3\sin(\frac{12x-\pi}{6})\sin(\frac{12x+\pi}{4}) + 2\cos(\frac{12x-\pi}{6})\cos(\frac{12x+\pi}{4})}{5}$$

$$g_4 := \frac{\sin(x)\big(2\sin(x)^2 - 3\big)}{3\cos(x)\big(\sin(x)^2 - 1\big)}$$

$$g_5 := \frac{2\sin(x)^2 + 2\cos(x) - 1}{3\sin(x)\big(\cos(x) - 1\big)}$$

$$g_6 := \frac{-\log\big(3\tan(x) + 5\big) + \log\big(\tan(x)\big)}{10}$$

Die Berechnung der letzten drei Integrale erfolgte mit Hilfe eines allgemeinen Verfahrens, das darauf beruht, die Kerne $\sin(x)$, $\cos(x)$ durch $\tan(\frac{x}{2})$ auszudrücken. Dies ist mit folgendem Regelsatz möglich, nachdem alle anderen Winkelfunktionen allein durch `sin` und `cos` ausgedrückt sind:

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

Die dazu inverse Regel

```
invtrigtan:={ tan(~x) => sin(2x)/(1+cos(2x) };
```

erlaubt es, Halbwinkel im Ergebnis wieder so weit als möglich zu eliminieren. Grundlage dieses Vorgehens ist die Fähigkeit der Systeme, rationale Funktionen in der Integrationsvariablen zu integrieren sowie der

Satz 8 *Sei $R(z) = \frac{P(z)}{Q(z)}$ eine rationale Funktion. Dann kann*

$$\int R(\tan(\frac{x}{2}))\,dx$$

durch die Substitution $y = \tan(\frac{x}{2})$ auf die Berechnung des Integrals einer rationalen Funktion zurückgeführt werden.

Beweis: Es gilt $\tan(x)' = 1 + tan(x)^2$ und folglich $dx = \frac{2\,dy}{1+y^2}$, womit wir

$$\int R(\tan(\frac{x}{2}))\,dx = \int \frac{R(y)}{1+y^2}\,dy$$

erhalten.

4.7 Das allgemeine Simplifikationsproblem

Betrachten wir zum Abschluss dieses Kapitels, wie sich die verschiedenen CAS bei der Simplifikation komplexerer Ausdrücke verhalten.

1. Beispiel:

```
u1:=(exp(x)*cos(x)+cos(x)*sin(x)^4+2*cos(x)^3*sin(x)^2+cos(x)^5)/
(x^2-x^2*exp(-2*x))-(exp(-x)*cos(x)+cos(x)*sin(x)^2+cos(x)^3)/
(x^2*exp(x)-x^2*exp(-x));
```

Dieser nicht zu komplizierte Ausdruck enthält neben trigonometrischen auch Exponentialfunktionen. Hier sind offensichtlich die Regeln anzuwenden, die weitestgehend eine der Winkelfunktionen durch die andere ausdrücken. Als Ergebnis erhält man den Ausdruck

$$\frac{\cos(x)\,(e^x+1)}{x^2}$$

Eine genauere Analyse zeigt, dass hierfür (neben der Reduktion der verschiedenen exp-Kerne auf einen einzigen) nur die Regel $\cos(x)^2 \Rightarrow 1 - \sin(x)^2$ anzuwenden ist.

MAPLE hatte noch in der Version 3 Schwierigkeiten, den kleinsten gemeinsamen Nenner dieses Ausdrucks zu erkennen, der ja hinter vielen verschiedenen `exp`-Kernen verborgen ist. Selbst

```
simplify((exp(x)-exp(-x))/(1-exp(-2*x)));
```

lieferte

$$-\frac{e^x - e^{-x}}{-1 + e^{-2\,x}} \qquad \text{statt} \qquad e^x$$

Das war in den Versionen 4 – 6 behoben. Version 7 hat mit diesem Ausdruck die alten Schwierigkeiten. Ähnliches gilt für MuPAD 2.0, wobei wenigstens `simplify` eine Reduktion der Anzahl der Kerne vornimmt.

Reduce benötigt, nach unseren bisherigen Ausführungen nicht unerwartet, als Hilfestellung die oben angegebene Regel[7], während Derive und Mathematica ebenfalls bereits mit `Simplify` bzw. `Together` zu dem erwarteten Ergebnis kommen. Macsyma kann den Ausdruck mit einer speziellen Simplifikationsroutine für trigonometrische Funktionen ebenfalls zufriedenstellend vereinfachen.

2. Beispiel:

```
u2:=16*cos(x)^3*cosh(x/2)*sinh(x)-6*cos(x)*sinh(x/2)-
6*cos(x)*sinh(3/2*x)-cos(3*x)*(exp(3/2*x)+exp(x/2))*(1-exp(-2*x));
```

Hier sind die Simplifikationsregeln für trigonometrische und hyperbolische Funktionen anzuwenden. Das kann man etwa durch Umformung in Exponentialausdrücke erreichen. Dieses Zusatzwissen erlaubt es Reduce, mit den Regelsystemen

```
hyp2exp:={
tanh(~x) => sinh(x)/cosh(x),
coth(~x) => cosh(x)/sinh(x),
sinh(~x) => (e^x-e^(-x))/2,
cosh(~x) => (e^x+e^(-x))/2};

trig2exp:={
tan(~x) => sin(x)/cos(x),
cot(~x) => cos(x)/sin(x),
sin(~x) => (e^(I*x)-e^(-I*x))/2/I,
cos(~x) => (e^(I*x)+e^(-I*x))/2};
```

die Identität $u = 0$ zu erkennen.

Dasselbe Ergebnis erhielt man in Maple V.3 über

```
u2a:=convert(u2,exp);
u2b:=expand(u2a);
```

Um den Ausdruck auch in neueren Versionen zu Null zu vereinfachen, ist noch eine zusätzliche Anwendung der Potenzgesetze notwendig:

```
combine(u2b,power);
```

Ähnlich kann man in Macsyma und MuPAD vorgehen, während in Mathematica wiederum ein Aufruf der Funktion `Simplify` genügt. Derive genügt ein Aufruf von Simplify, wenn man vorher `Trigonometry:=Expand` gesetzt hat.

System	Beispiel `u1`	Beispiel `u2`
Derive	(S)implify	(S)implify mit `Trigonometry:=Expand`
Macsyma	`rat(trigreduce(u1))`	`expand(exponentialize(u2))`
Maple	`simplify(u1)`[8]	`combine(expand(convert(u2,exp)),power)`
Mathematica	`u1//Simplify`	`u2//Simplify`
MuPAD	`simplify(u1)`	`combine(expand(rewrite(u2,exp)),exp)`
Reduce	`u1 where trigsin`	`u2 where hyp2exp,trig2exp`

[7] Auch die Funktion `trigsimp(u);` aus dem Paket `trigsimp` liefert das obige Resultat

Tabelle 4: Simplifikation von u_1 und u_2 auf einen Blick

3. Beispiel (aus [5, S. 81])

```
u3:=log(tan(x/2)+sec(x/2))-arcsinh(sin(x)/(1+cos(x)));
```

Diesen Ausdruck vermag keines der Systeme zu Null zu vereinfachen, obwohl auch hier das Vorgehen für einen Mathematiker mit geübtem Blick sehr transparent ist: Wegen $\operatorname{arcsinh}(a) = \log(a+\sqrt{a^2+1})$ ist arcsinh durch einen logarithmischen Ausdruck zu ersetzen und dann die beiden Logarithmen zusammenzufassen. Dies wird in DERIVE beim Vereinfachen automatisch vorgenommen, ist in MACSYMA, MAPLE und MUPAD durch eingebaute Funktionen (`convert(u3,ln)` oder `rewrite(u3,ln)`) und in den anderen Systemen durch Angabe einer einfachen Regel realisierbar, etwa in MATHEMATICA als

```
Arch2Log = { ArcSinh[x_] -> Log[x+Sqrt[1+x^2]] }
```

Vereinfacht man die Differenz A dieser beiden Logarithmen mit `simplify`, so wartet nur MATHEMATICA mit einem einigermaßen passablen Ergebnis auf:

$$\log\left(\sec(\frac{x}{2})+\tan(\frac{x}{2})\right)-\log\left(\sqrt{\sec(\frac{x}{2})^2}+\tan(\frac{x}{2})\right)$$

Die Simplifikation endet an der Stelle, wo $\sqrt{x^2}$ nicht zu x vereinfacht wird, obwohl dies hier aus dem Kontext heraus erlaubt wäre (wenn man voraussetzt, dass es sich um reelle Funktionen handelt): $\log(\sec(\frac{x}{2})+\tan(\frac{x}{2}))$ hat als Definitionsbereich $D = \{x : \cos(\frac{x}{2}) > 0\}$. Wir können MATHEMATICA mit der Regel

```
SimpSqrt = { Sqrt[x_^2] -> x }
```

veranlassen, diese Vereinfachung trotzdem vorzunehmen und erhalten danach schließlich 0.

Zugleich sehen wir an dieser Stelle, dass die Vereinfachung von u3 zu Null ohne weitere Annahmen über x mathematisch nicht korrekt wäre, da für negative Werte des ersten Logarithmanden eine imaginäre restgröße stehenbliebe. Beschränken wir deshalb, wenn möglich, x auf das reelle Intervall $-1 < x < 1$, in dem alle beteiligten Funktionen definiert und reellwertig sind.

In MAPLE 7 setzen wir dazu etwa

```
assume(-1<x,x<1);
```

in MUPAD 2.0

```
assume(-1<x):
assume(x<1,_and):
```

was in der folgenden Eigenschaft resultiert:

```
getprop(x);
                              ]-1, 1[ of Type::Real
```

Um in den anderen Systemen ebenfalls die Nulläquivalenz des betrachteten Ausdrucks A zu erkennen, ist es sinnvoll, die beiden Logarithmen zusammenzufassen. In DERIVE erfolgt dies automatisch und ist mit `logcontract` (MACSYMA), `combine(A,ln)` (MAPLE und MUPAD) oder dem Schalter `combinelogs` (REDUCE) auch in den anderen Systemen auf einfache Weise möglich.

[8] Nicht mehr in MAPLE 7.

Ohne die Voraussetzung an x weigert sich MAPLE allerdings hartnäckig, die Zusammenfassung $\ln(u)-\ln(v) \mapsto \ln(u/v)$ vorzunehmen, da die Werte des Logarithmus sonst ja komplex werden und das Ergebnis verfälscht sein könnte. Da regelbasiertes Programmieren von MAPLE nicht unterstützt wird, müssen wir uns an dieser Stelle damit behelfen, die geforderte Funktionstransformation "per Hand" vorzunehmen, indem wir A auseiandernehmen und anders wieder zusammenbauen:

```
A1 := combine(A,ln);
A2 := ln(op([1,1],A1)/op([2,2,1],A1));
```

Seit Version 5 kann diese Funktionstransformation allerdings auch über den Befehl

```
combine(A,ln,anything,symbolic)
```

erreicht werden.

Nachdem diese Hürde genommen ist entsteht als Argument der Logarithmusfunktion ein komplizierter trigonometrischer Ausdruck

$$B := \frac{\tan(\frac{x}{2}) + \sec(\frac{x}{2})}{\frac{\sin(x)}{1+\cos(x)} + \sqrt{\frac{(\sin(x))^2}{(1+\cos(x))^2}+1}},$$

der mit den bisher betrachteten Regeln weiter vereinfacht werden kann. MATHEMATICA liefert bereits mit `Simplify[B]` ein verwertbares Ergebnis

$$\frac{1+\sin(\frac{x}{2})}{\cos(\frac{x}{2})\sqrt{\sec(\frac{x}{2})^2}+\sin(\frac{x}{2})},$$

das mit der Regel `simpsqrt` zu Eins vereinfacht.

Die anderen Systeme haben Schwierigkeiten, die Winkelargumente x und $x/2$ zueinander in Beziehung zu setzen. Ersetzen wir in B deshalb noch $x \mapsto 2y$:

```
C:=subs(x=2*y,B);
```

MAPLE liefert nun mit

```
C1:=simplify(expand(C));
```

$$\frac{\sin(y)+1}{\sin(y)+\mathrm{csgn}(\cos(\overline{y}))}$$

ein erstes Ergebnis, das nach Einschränkung des Definitionsbereichs von y mit

```
assume(-1/2<y,y<1/2);
```

zum korrekten Ergebnis 1 vereinfacht werden kann.

Dasselbe Ergebnis erhält man mit DERIVE, wenn `Trigonometry := Expand` einstellt wird.

MACSYMA, das eine Einschränkung des Definitionsbereich von x nicht unterstützt, liefert mit

```
trigsimp(trigexpand(C));
```

$$\frac{\cos y \sin y + \cos y}{|\cos y| + \cos y \sin y};$$

ein Ergebnis ähnlicher Qualität. Man beachte, dass dieser einfache Ausdruck nur entsteht, wenn zuerst die Mehrfachwinkel aufgelöst werden, ehe die allgemeine Simplifikationsroutine gestartet wird.

Mit REDUCE kann man dieselben Umformungen durch geeignete Regelsysteme erreichen, die in der Datei `trig` enthalten sind. Zunächst erzeugen wir den Ausdruck B. In Version 3.6 funktioniert die Befehlsfolge

```
on combinelogs$
A:=(u3 where archtolog)$
B:=part(A,1)$
```

In Version 3.7 dagegen werden die Logarithmen selbst für den Fall, dass `on combinelogs` gesetzt ist, nicht zusammengefasst, so dass wir wie oben die entsprechende Funktionstransformation "per Hand" vornehmen müssen:

```
B:=part(A,1,1)/part(A,2,1,1)$
C:=sub(x=2*y,B)$
C where trig;
```

In Version 3.6 erhalten wir als (vorsichtigeres) Ergebnis korrekterweise noch

$$\frac{\cos(y)\sin(y)+\cos(y)}{\mathrm{abs}(\cos(y))+\cos(y)\sin(y)},$$

während Version 3.7. (in der Version vom 15 April 1999) wieder die Vereinfachung $\sqrt{x^2} = x$ vornimmt und so den Wert 1 zurückliefert.

Bei MUPAD 2.0 führt dieselbe Strategie (mit der Annahme $-1 < x < 1$) bereits beim Zusammenfassen der beiden logarithmischen Terme zu einer Fehlermeldung:

```
A:=rewrite(u3,ln):
combine(A,ln);
  Error: Illegal operand [_power]; during evaluation of 'property::IVnat::_power'
```

Setzt man B wie oben selbst zusammen (Achtung: Die Reihenfolge der Summanden muss nicht mit der ausgegebenen Reihenfolge übereinstimmen. `op(A)` bringt sie ans Licht!)

```
B:=op(A,[2,1])/op(A,[1,1,1]);
```

und wendet `simplify` an, so wird wieder mit einem Fehler abgebrochen. Auch weitere Manipulationen führen zu keinem verwertbaren Ergebnis, da `simplify` die mit `expand` erreichte Reduktion der Kerne mit notorischem Starrsinn wieder rückgängig macht.

Wir sehen hieran bereits, dass man sich offensichtlich stets genügend komplizierte Simplifikationsprobleme ausdenken kann, die mit dem vorhandenen Instrumentarium nicht aufgelöst werden können. Es gilt jedoch noch mehr:

Satz 9 *Aus den Funktionssymbolen* sin, abs *und* $*$, $+$ *sowie der Konstanten* π *und ganzen Zahlen kann man Ausdrücke zusammenstellen, für die das Simplifikationsproblem algorithmisch unlösbar ist.*

Das allgemeine Simplifikationsproblem gehört damit zur Klasse der algorithmisch unlösbaren Probleme.

Der Beweis dieses Satzes (der hier nicht geführt wird) wird zurückgeführt auf die negative Antwort zum 10. Hilbertschen Problem, die Matiyasewitsch und Roberts Ende der 60er Jahre gefunden haben.

Literaturverzeichnis

[1] *Brockhaus Enzyklopädie in 26 Bänden.* F.A. Brockhaus, Mannheim, 1994.

[2] B. Buchberger. Symbolisches Rechnen. In P. Rechenberg and H. Pomberger, editors, *Informatik-Handbuch*, chapter E5, pages 799 – 817. Hanser, München, 1997.

[3] B. Buchberger and R. Loos. Algebraic simplification. In B. Buchberger, G.E. Collins, and R. Loos, editors, *Computer Algebra - Symbolic and Algebraic Computation*, pages 11–43. Springer, Wien, second edition, 1983.

[4] H. Engesser, editor. *Duden Informatik.* Dudenverlag, Mannheim, 1993.

[5] K.O. Geddes, S.R. Czapor, and G. Labahn. *Algorithms for Computer Algebra.* Kluwer Acad. Publisher, 2 edition, 1992.

[6] J. Grabmeier. Computeralgebra – eine Säule des Wissenschaftlichen Rechnens. *it + ti*, 6:5 – 20, 1995.

[7] J. Grabmeier, E. Kaltofen, and V. Weispfenning, editors. *Computer Algebra Handbook. Foundations – Applications – Systems.* Springer, Berlin, 2003.

[8] D. Gruntz and M. Monagan. Introduction to Gauss. *Maple Technical Newsletter*, 9, 1993.

[9] D.E. Knuth. *The art of computer programming.* Addison Wesley, 1991.

[10] R. Loos. Introduction. In B. Buchberger, G.E. Collins, and R. Loos, editors, *Computer Algebra - Symbolic and Algebraic Computation*, pages 1–10. Springer, Wien, second edition, 1983.

[11] T. Ottmann and P. Widmeyer. *Algorithmen und Datenstrukturen.* BI Wissenschaftsverlag, 2 edition, 1993.

[12] R. Pavelle, M. Rothstein, and J.P. Fitch. Computer algebra. *Scientific American*, 245(6):102 – 113, dec 1981.

[13] J.K. Prentice and M. Wester. Code generation using Computer Algebra Systems. In M. Wester, editor, *Computer Algebra Systems: A Practical Guide*, chapter 13, pages 233 – 254. Wiley, Chichester, 1999.

[14] P. Rechenberg and H. Pomberger, editors. *Informatik-Handbuch.* Hanser, München, 1997.

[15] Schneider, editor. *Lexikon Informatik.* Oldenbourg, München, 4.0 edition, 1997.

[16] U. Schwardmann. *Computeralgebrasysteme.* Addison-Wesley, 1995.

[17] B. Simon. Comparative CAS review. *Notices AMS*, 39:700 – 710, sept 1992.

[18] B. Stroustrup. *Die C++-Programmiersprache.* Addison-Wesley, 2 edition, 1992.

[19] J.A. van Hulzen and J. Calmet. Computer Algebra Systems. In B. Buchberger, G.E. Collins, and R. Loos, editors, *Computer Algebra - Symbolic and Algebraic Computation*, pages 221 – 243. Springer, Wien, second edition, 1983.

[20] J. Weizenbaum. *Die Macht der Computer und die Ohnmacht der Vernunft*, volume 274 of *Taschenbuch Wissenschaft*. Suhrkamp, 9 edition, 1994.

[21] M. Wester. A review of CAS mathematical capabilities. *CAN Nieuwsbrief*, 13:41–48, dec 1994.

[22] M. Wester, editor. *Computer Algebra Systems: A Practical Guide*. Wiley, Chichester, 1999.

[23] N. Wirth. *Algorithmen und Datenstrukturen*. B.G. Teubner Verlag Stuttgart, 4 edition, 1986.

[24] S. Wolfram. *The Mathematica Book*. Cambridge University Press, 1996.