# Der Funktionsbegriff im symbolischen Rechnen

Hans-Gert Gräbe, Leipzig

Version vom 16. November 2011

**Zusammenfassung**

Der – allein propädeutisch mögliche – Einsatz von Computeralgebra im Schulunterricht stellt hohe Anforderungen an Lehrende, Konzepte und Lehrmaterialien. In diesem Aufsatz diskutiere ich verschiedene Aspekte dieser Problematik am Beispiel des Funktionsbegriffs im Spannungsfeld zwischen Mathematik und Informatik. Der Schwerpunkt der Ausführungen liegt auf der sachlichen Darstellung der verschiedenen Facetten des Funktionsbegriffs und der praktischen Umsetzung dieser Facetten in den programmiersprachlichen Möglichkeiten eines CAS. Die prinzipielle kulturelle Bedeutung des Zugangs zu diesen (programmier)sprachlichen Mitteln bereits im Schulunterricht im Kontrast zur heute verbreiteten Art der Taschenrechnernutzung wird besonders hervorgehoben.

## 1 Einführende Bemerkungen

Wenn heute über Computeralgebra (CA) – präziser eigentlich *Computermathematik* – und den Einsatz von Computeralgebra-Systemen (CAS) in der Ausbildung gesprochen wird, dann bewegen sich die Diskussionen meist zwischen zwei Polen. Einerseits wird der ungeheure Zuwachs an Möglichkeiten betont, den mathematische Argumentation, Modellierung und Rechnung durch die neuen Werkzeuge erfahren und bereits erfahren haben. Und andererseits werden die didaktischen Potenzen beschworen, welche sich aus dem Einsatz CAS-gestützter Technik im bisherigen Curriculum ergeben.

Dabei wird gelegentlich die Frage übersehen, ob *Computeralgebra in der Ausbildung* ausschließlich Mittel zum Erreichen bisheriger Zwecke im Spannungsfeld zwischen Mathematik- und Informatik-Ausbildung ist oder ob es – jenseits aller Begeisterung – in einer technik- und algorithmen-dominierten Welt auch eine eigenständige Bedeutung etwa für die Allgemeinbildung hat, Erfahrungen mit computeralgebraischen Werkzeugen zu sammeln und kritisch zu verarbeiten.

Ich möchte einer solchen Frage im Weiteren an einem scheinbar einfachen Gegenstand nachgehen – dem Funktionsbegriff. Dessen Klippen und Untiefen im Spannungsfeld zwischen Mathematik und Informatik sind gut bekannt. Während die Mathematik „existenziellen“ Fragen ganzer Funktionenklassen nachgeht, sind in der Informatik Funktionen vor allem als handhabbares Konzept der Wiederverwendung im Einsatz, mit zentralen Begriffen wie Spezifikation, Parameterbelegung und Programmablaufstrukturen.

Dass sich diese verschiedenen Sichten realweltlich bei Fragen des angemessenen Einsatzes von CA-Instrumenten im beruflichen Alltag in voller Breite bemerkbar machen, versteht sich von selbst. Deshalb wird von Lehrerinnen und Lehrern auch jenseits explizit interdisziplinärer

Unterrichtsansätze ein Problembewusstsein für derartige Klippen und sicheres Beherrschen der relevanten Begrifflichkeiten und Konzepte verlangt, wenn dies schülergerecht und ohne Sinn entstellende Verkürzungen für das eigene Unterrichtsgeschehen aufbereitet werden soll.

Derartige Überlegungen sind eingebettet in eine umfassendere Debatte, von der ich zwei Aspekte hervorheben möchte:

1. Die Problematik ist Teil der erbitterten Debatten um die Integration informatisch-algorithmischer, also letztlich technischer Aspekte in das Schulcurriculum. Hier kann gerade die *praktische* Befassung mit Mathematik – ob mit oder ohne Computer – Wunder bewirken für das Verständnis der prinzipiellen Möglichkeiten einer *lingua franca* der Wissenschaft und damit auch für das Interesse, deren Potenzen für sich selbst praktisch zu erschließen.

2. Die Frage nach technik- und algorithmenzentrierten allgemeinbildenden Inhalten ist eingebettet in die Debatten um das Thema „technical literacy“ und allgemeiner um die Frage, was es bedeutet, die Auszubildenden auf ein angemessenes, eigenständiges und verantwortungsvolles Agieren in einem technisch und kulturell geprägten Umfeld vorzubereiten.

Die Mathematik als *lingua franca* hat ihr Pendant in der Programmiersprache des eingesetzten CAS, mit deren Hilfe Schülerinnen und Schüler in die Lage versetzt werden (können), mathematische Probleme so weit und so stringent sprachlich aufzubereiten, dass sie sogar ein „dummer“ Computer versteht. Die hierfür zu entwickelnde und zu erwerbende Ausdrucksfähigkeit besteht aus zwei Komponenten – der (möglichst raschen) Beherrschung eines kleinen Instrumentariums nützlicher und immer wiederkehrender Kontrollfluss-Strukturen und der (inkrementellen) Erschließung der mathematischen und syntaktischen Bedeutung konkreter Funktionen. Der Einsatz von Taschenrechnern und CAS ohne derartige ausgebaute Programmierfähigkeit oder Abstinenz in der Nutzung derselben verstellt hier gründlich den Blick auf Mögliches und Wünschenswertes.

Für die weiteren Ausführungen ist deshalb die Verfügbarkeit einer solchen CAS-Programmiersprache vorausgesetzt. *Praktisch* lassen sich die darzustellenden Aspekte des Funktionsbegriffs im Spannungsfeld zwischen Mathematik und Informatik dann oft an der Frage eines *guten Programmierstils* fest machen. Dabei geht es mir nicht um Fragen des Geschmacks – über den sich bekanntlich (nur) streiten lässt –, sondern darum, wie sprachlich zum Ausdruck kommt, ob ein Werkzeug nicht irgendwie, sondern problem- und technologieadäquat eingesetzt wird.

Dies ist eine sehr ausbildungsrelevante Frage, denn erst und nur so lassen sich einerseits die Potenzen von Wissenschaft und Technik voll ausnutzen und besteht andererseits die Chance, die Technik in der Rolle des dienenden Werkzeugs zu belassen bzw. zu halten. Eine hoch entwickelte Technik bedarf eines hoch entwickelten Intellekts, um angemessen eingesetzt zu werden. Viele unserer Informatikstudenten haben damit selbst noch im Masterstudium ihre Probleme und geben sich zufrieden, wenn sie etwas Lauffähiges „zusammengehackt“ haben.

Ich fordere auf, von Anfang an Wert zu legen auf einen sowohl dem Problem als auch dem Werkzeug *angemessenen* praktischen Zu- und Umgang mit Computeralgebra. Die Codebeispiele im Text beziehen sich mehrheitlich auf das freie CAS Maxima[1], lassen sich aber ohne

[1] `http://maxima.sourceforge.net`

viel Mühe auch auf andere CAS, namentlich die kommerziellen Systeme MATHEMATICA[2] und MAPLE[3], übertragen. Wer mehr über die einheitlichen Prinzipien erfahren möchte, die sich hinter diesen Ähnlichkeiten im Design verbergen, sei auf [2] verwiesen.

## 2 Der Funktionsbegriff in Mathematik und Informatik

Funktionen bezeichnen im mathematischen Sprachgebrauch *Abbildungen* $f : X \to Y$ von einem Definitionsbereich $X$ in einen Wertevorrat $Y$. In der Informatik geht es um deren konstruktiven Aspekt, genauer um den *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. Dabei ist zwischen *Funktionsdefinitionen* und *Funktionsaufrufen* zu unterscheiden.

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 eindeutig bestimmte positive reelle Zahl, deren Quadrat gleich $x$ ist“.

## 3 Symbolisches und numerisches Rechnen

Eine solche Differenz ist charakteristisch für den Unterschied zwischen symbolischem und numerischem Rechnen. Im klassischen Programmieren wird genau zwischen der *Designzeit*, zu der eine Funktionsdefinition vereinbart wird, und der *Laufzeit*, zu der eine solche Funktion ein- oder mehrmals abgearbeitet wird, unterschieden. Das ist im symbolischen Rechnen nicht anders. Im klassischen Programmieren numerischer Algorithmen treten allerdings Variablenbezeichner und Ausdrücke ausschließlich zur Designzeit auf. Bei der Übersetzung in ein Maschinenprogramm werden diese Ausdrücke nach wohldefinierten Regeln aufgelöst (geparst), den Variablenbezeichnern entsprechende Speicherbereiche zugewiesen und die Hochsprachenalgorithmen in Maschinensprachalgorithmen übersetzt, welche dann zur Laufzeit mit konkreten Werten abgearbeitet werden, die an den markierten Adressen gespeichert sind. Alle symbolischen Bezeichnungen, die zur Beschreibung des Algorithmus zur Designzeit verwendet wurden, verschwinden in diesem Übersetzungsprozess komplett. Zur Laufzeit wird auf der Ebene der Maschinensprache allein mit Speicheradressen und Speicherinhalten gerechnet.

Eine solche Reduktion ist im symbolischen Rechnen nicht mehr möglich, da sich zwar die entsprechenden Ausdrücke auch hier nach (denselben) wohldefinierten Regeln parsen lassen, aber die Übersetzung in Maschinensprache daran scheitert, dass einzelnen Variablenbezeichnern kein Wert, ja nicht einmal ein Typ, zugewiesen ist. Symbolische Systeme müssen also

[2] `http://www.wolfram.com/mathematica`
[3] `http://maplesoft.com/products/Maple`

auch zur Laufzeit mit teilausgewerteten Ausdrücken weiterrechnen und *verhalten sich deshalb zur Laufzeit ähnlich wie klassische Systeme zur Designzeit.* Dazu werden die Variablenbezeichner in einer *Symboltabelle* aufgesammelt, und es ist zu jedem Zeitpunkt der Rechnung (also zur *Laufzeit* des CAS) zu unterscheiden, welchen Bezeichnern ein Wert zugewiesen ist (Bezeichner im Wertmodus) und welchen nicht (Bezeichner im Symbolmodus).

Ausdrücke spielen damit eine zentrale Rolle im symbolischen Rechnen und stehen im Mittelpunkt des Designs jedes CAS. Einer der zentralen Merksätze im MATHEMATICA-Handbuch heißt deshalb auch *Everything is an Expression.* Symbolische Ausdrücke wie $a + b \cdot c$ werden intern sofort in Ausdrücke `plus(a,times(b,c))` umgesetzt, in denen *Funktionssymbole* `plus` oder `times`, d. h. Funktionen ohne (ausführbare) Funktionsdefinition wie das oben betrachtete `sqrt`, einen wichtigen Bestandteil bilden.

Ein Anmerkung zu den didaktischen Potenzen des Anschreibens von Berechnungen, die für numerische und symbolische Anwendungen gleichermaßen gilt: Die strengen Anforderungen einer Programmiersprache an die Genauigkeit der Syntax zwingen die Nutzer solcher Systeme, auf die Genauigkeit und Aussagekraft ihrer Formulierungen besondere Obacht zu geben. Programmierumgebungen sind dabei streng und helfend zugleich – streng, da sie in der Regel extrem fehlerintolerant sind, und helfend, da die meisten in der Lage sind, wenigstens Orte von Syntaxfehlern zu lokalisieren.

# 4 Funktionen und Funktionsbezeichner

Wir hatten gesehen, dass im symbolischen Rechnen neben Variablenbezeichnern, die im klassischen Programmieren einzig zum Aufbewahren von Werten eingesetzt sind, auch Funktionsbezeichner eine Rolle spielen. Für diese gilt die Unterscheidung zwischen Symbolmodus (Bezeichner ohne Funktionsdefinition) und Wertmodus (Bezeichner mit Funktionsdefinition) analog, und die meisten CAS verwenden sogar einen gemeinsamen Namensraum für Variablen- und Funktionsbezeichner.

Allerdings hängt der Modus eines Funktionsbezeichners – der zu jedem Moment der Laufzeit neu zu bestimmen ist – zusätzlich von den jeweiligen Aufrufparametern ab. Es wird grundsätzlich zunächst versucht, den Bezeichner als Funktionsaufruf zu interpretieren. Ein solcher Funktionsaufruf folgt (von wenigen Ausnahmen abgesehen) dem klassischen Aufrufschema *call by value* und wird mit den symbolischen Ausdrücken abgearbeitet, die sich durch Auswertung der Aufrufparameter ergeben haben. Existiert keine Funktionsdefinition, so wird allerdings nicht mit einer Fehlermeldung abgebrochen, sondern – wie in dem oben betrachteten Beispiel `sqrt(2)` – aus den Werten der formalen Parameter und dem Funktionssymbol als „Kopf“ ein neuer symbolischer Ausdruck als teilausgewertetes Artefakt gebildet. Derartige Ausdrücke bezeichnen wir im Weiteren als *Funktionsausdrücke.*

Normalerweise ist damit eine Funktion nur partiell (im informatischen Sinne) definiert, d. h. für spezielle Argumente findet ein Funktionsaufruf statt, für andere dagegen wird ein Funktionsausdruck gebildet. So bleibt etwa bei der Auswertung der MAXIMA-Konstrukte `sin(x)` und `sin(2)` alles unverändert – für symbolische oder ganzzahlige Argumente wird das Funktionssymbol `sin` zur Konstruktion von Funktionsausdrücken verwendet, die für exakte Sinus-Funktionswerte stehen wie oben `sqrt(2)` für einen exakten Wurzelwert. Im Gegensatz dazu erfolgt bei der Eingabe von `sin(2.55)` ein klassischer Funktionsaufruf, der für dieses Float-

Argument eine Funktionsdefinition von `sin` zur näherungsweisen Berechnung für einen reellwertigen Parameter verwendet und den Zahlenwert `0.55768371739142` zurückliefert.

Hierbei ist – wie in klassischen Programmiersprachen auch – genau zwischen einem Integerwert wie `2` und einem Floatwert wie `2.0` zu unterscheiden, die auch computerintern vollkommen anders dargestellt sind. Bei symbolischen Rechnungen mit Float-Werten wird fast immer angenommen, dass auch ein Näherungswert zurückgegeben werden soll, und damit der Bereich des symbolischen Rechnens verlassen. Anderes ergibt auch wenig Sinn. Sind dagegen die Eingabeparameter exakt, so wird exakt gerechnet und ein Ergebnis derselben Qualität wie das oben bereits diskutierte `sqrt(2)` zurückgegeben. Um auch in diesem Fall einen Näherungswert berechnen, ist `sqrt(2.0)` oder `float(sqrt(2))` einzugeben.

Ähnlich den Bezeichnern für Variablen können auch neue Funktionsbezeichner eingeführt werden, von denen zunächst nichts bekannt ist und die damit im Symbolmodus als Funktionssymbole ohne Funktionsdefinition behandelt werden. Die MAXIMA-Zuweisung

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

registriert ein neues Funktionssymbol $f$ und gibt $f(x) + 2\,x + 1$ zurück. Nach der Substitutionsanweisung

```
ev(u,x=2)
```

wird $f(x) + 5$ zurückgegeben, und mit $f(x)$ und $f(2)$ existieren in der bisherigen Rechnung zwei Funktionsausdrücke mit dem Kopf $f$.

# 5 Transformationen und Transformationsfunktionen

Von besonderer Art sind MAXIMA-Funktionen wie `expand`, `trigexpand` oder `ratsimp`, die symbolische Ausdrücke transformieren. Die Wirkung dieser Funktionsaufrufe ist auf den Umbau der als Parameter übergebenen Funktionsausdrücke ausgerichtet. Solche *Transformationen* ersetzen gewisse Kombinationen von Funktionssymbolen und evtl. speziellen Funktionsargumenten durch andere, mathematisch gleichwertige Kombinationen.

Transformationen sind eines der zentralen Designelemente von CAS, da auf diesem Wege syntaktisch verschiedene, aber mathematisch (semantisch) gleichwertige Ausdrücke produziert werden können. So analysiert MAXIMA für das Ergebnis $\frac{1}{\sqrt{2}}$ der Berechnung `sin(%pi/4)` das Zusammentreffen der Symbole `sin` und `%pi` und löst eine solche Transformation automatisch aus. Auch Vereinfachungen wie etwa `sqrt(2)^2` zu 2 liegen solche Transformationen zu Grunde.

Da Transformationen in einem breiten und in seiner Gesamtheit widersprüchlichen Spektrum möglich sind, können sie in den meisten Fällen aber nicht automatisch vorgenommen werden. Für einzelne Transformationsaufgaben gibt es deshalb spezielle *Transformationsfunktionen*, die aus dem Gesamtspektrum möglicher (genauer: im jeweiligen CAS implementierter) Transformationen eine konsistente Teilmenge auf einen als Parameter übergebenen Ausdruck *lokal* anwenden.

Betrachten wir die Wirkung einer solchen Transformationsfunktion, der MAXIMA-Funktion `expand`, näher. Der Aufruf

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

verwandelt ein Produkt von zwei Summen in eine Summe nach dem Distributivgesetz und gibt $x^2 + 3\,x + 2$ zurück.

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*. Dabei werden komplexere Teilstrukturen der Argumente durch die aufrufende Funktion analysiert.

So muss die Transformationsfunktion beim Aufruf

```
expand(sum1*sum2)
```

etwa erkennen, dass ihr Argument `sum`$_1$`*sum`$_2$ ein Produkt zweier Summen ist, die Listen $l_1$ und $l_2$ der Summanden beider Faktoren extrahieren und seinerseits einen weiteren Funktionsaufruf

```
expandproduct(l1,l2)
```

auslösen, der aus $l_1$ und $l_2$ alle paarweisen Produkte bildet und diese danach aufsummiert.

Transformationen sind manchmal nur über Umwege zu realisieren, da beim Aufruf einer Funktion deren Argumente ausgewertet werden, was deren Struktur so verändern kann, dass die syntaktischen Bestandteile, deren Zusammentreffen ausgewertet werden soll, zur Bearbeitungszeit der äußeren Funktion gar nicht mehr vorhanden sind.

Dabei sind verschiedene Feinheiten zu beachten, auf die ich am folgenden MAPLE-Beispiel zu sprechen komme. Möchte man das Polynom $f = x^3 + x^2 - x + 1$ nicht über den ganzen Zahlen faktorisieren, sondern modulo 2, also im Ring $\mathbb{Z}_2[x]$, so führen weder die Eingabe

```
factor(f) mod 2
```

noch

```
factor(f mod 2)
```

zum richtigen Ergebnis. Im ersten Fall ergibt sich $x^3 + x^2 + x + 1$, im zweiten $(x + 1)\,(x^2 + 1)$. Das zweite Ergebnis kommt der korrekten Antwort schon nahe – Ausmultiplizieren ergibt $x^3 + x^2 + x + 1 \equiv f \pmod 2$ –, ist aber wegen $(x^2 + 1) \equiv (x + 1)^2 \pmod 2$ noch immer falsch, weil nicht vollständig faktorisiert.

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 ist) wird vor dem Aufruf von `mod` das Polynom $f$ über den ganzen Zahlen faktorisiert. Das Ergebnis enthält die Information `factor` nicht mehr, so dass `mod` als Reduktionsfunktion operiert und die Koeffizienten dieser ganzzahligen Faktoren modulo 2 reduziert. Im zweiten Fall dagegen wird das Polynom $f$ erst modulo 2 reduziert. Das Ergebnis enthält die Information `mod` nicht mehr und wird als ganzzahliges Polynom $x^3 + x^2 + x + 1$ betrachtet und auch so

faktorisiert. MAPLE hat also in beiden Fällen nicht die Faktorisierung über einem modularen Bereich, sondern die über den ganzen Zahlen aufgerufen.

Da es sich beim Faktorisieren von Polynomen über $\mathbb{Z}$ und über Restklassenkörpern $\mathbb{Z}_p$ um vollkommen verschiedene Algorithmen handelt, sind auf der Ebene der letztlich auszulösenden Funktions*aufrufe* dafür auch verschiedene Funktionsnamen, etwa `factorinteger` und `factormod`, erforderlich. Um dies wenigstens in jenem Teil zu vermeiden, mit dem der Nutzer unmittelbar zu tun hat, führt MAPLE ein weiteres *Funktionssymbol* `Factor` ein, das sein Argument unverändert zurückgibt. Der Befehl

```
Factor(f) mod 2
```

liefert das korrekte Ergebnis $(x+1)^3$.

Der Aufruf wird als `mod(Factor(f),2)` umgesetzt, beim Auswerten der Argumente von `mod` bleibt `Factor(f)` als Funktionsausdruck unverändert und am Zusammenstoßen der Funktionsbezeichner `mod` und `Factor` wird nun erkannt, dass modulares Faktorisieren aufzurufen ist. Dazu wird eine Funktions*transformation* ausgelöst, die die Argumente $f$ und 2 extrahiert und daraus den Funktions*aufruf*

```
`mod/Factor`(f,2)
```

generiert. Der Bezeichner `` `mod/Factor` `` dieser MAPLE-Funktion wird dabei aus den Bezeichnern `mod` und `Factor` zusammengebaut.

Wir sehen an diesem Beispiel, dass es die symbolischen Möglichkeiten eines CAS erlauben, zur Laufzeit einer Funktion neue Bezeichner sowohl für Variablen als auch Funktionen zu generieren. Dabei kann es sich sowohl um neue als auch dem System bereits bekannte Bezeichner handeln. Dieser Mechanismus kann mit einiger Perfektion auch zur Simulation von Polymorphie eingesetzt werden.

Funktionssymbole wie `Factor` 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.

Solche Hilfskonstruktionen sind für diesen Zweck allerdings nicht zwingend erforderlich. In MATHEMATICA kann das modulare Faktorisieren mit `Factor[f, Modulus -> 2]` aufgerufen werden. Da das zweite Argument beim Auswerten nicht verändert wird, wird am Zusammentreffen der Bezeichner `Factor` und `Modulus` erkannt, dass die modulare Faktorisierung zu verwenden ist. Ähnlich geht MAXIMA vor. Hier können Optionen als lokale Werte von Kontextvariablen komma-separiert angegeben werden. Der entsprechende Aufruf lautet

```
factor(f), modulus=2;
```

Dieser Zugang ließe sich leicht auch in MAPLE realisieren.

Derselbe Funktionsbezeichner kann in unterschiedlichem Kontext in verschiedenen Rollen auftreten, wie folgendes MAXIMA-Beispiel für die `exp`-Funktion zeigt. Beim Aufruf

```
exp(x)
```

erhalten wir mit `exp(x)` einen Funktions*ausdruck* mit dem Symbol `exp` als Kopf, beim Aufruf

```
exp(2.0)
```

die mit einem entsprechenden Näherungsverfahren in einem Funktions*aufruf* berechnete Float-Zahl 7.389056099, beim Aufruf

```
exp(log(x+y))
```

dagegen das Ergebnis $x + y$, welches über eine Funktions*transformation* berechnet wurde, die das Zusammentreffen von `exp` und `log` erkannt hat.

# 6 Funktionen von Funktionen

Auch die Resultate einer auf den ersten Blick stärker „algorithmischen“ Funktion wie etwa `diff` zum Berechnen von Ableitungen sind oft das Ergebnis von Transformationen. Für die folgenden Beispiele verwende ich zunächst MAPLE, da in MAXIMA einige Besonderheiten zu beachten sind, auf die ich weiter unten zu sprechen komme.

Im ersten Beispiel

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

beobachten wir das erwartete Verhalten. Das Zusammentreffen von `diff` und `sin` löst eine Transformation aus, bei der diese Kombination durch `cos` ersetzt und $\cos(x)$ zurückgegeben wird.

Auch im zweiten Beispiel

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

entspricht das Ergebnis $\frac{d}{dx} f(x)$ (fast) den Erwartungen. Bei genauerer Analyse stellt man fest, dass hier gar nichts geschehen ist, denn das Ergebnis ist nur die zweidimensionale Ausgabeform des Funktionsausdrucks `diff(f(x),x)`, wie mit `lprint` leicht festgestellt werden kann. Der Ausdruck konnte nicht vereinfacht werden, weil über $f$ nichts weiter bekannt ist.

Einzig die Kettenregel wird auf entsprechende Funktionsausdrücke angewendet, wie das dritte Beispiel

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

zeigt. Die entsprechende Transformation wird automatisch ausgeführt und in diesem Fall

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

zurückgegeben. Zur Anwendbarkeit der Kettenregel-Transformation wird dabei aus der syntaktischen Struktur geschlossen, dass es sich bei den Bezeichnern $f$ und $g$ um Funktionssymbole handelt.

Hinter der zweidimensionalen Ausgabe verbirgt sich der Ausdruck `D(f)(g(x))*diff(g(x),x)` mit dem Funktionssymbol $D$. $D(f)$ bezeichnet hier offensichtlich die Ableitung $f'$ der Funktion $f$. $D$ steht damit für eine Funktion, deren Aufrufargument selbst eine Funktion ist. Dies mag auf den ersten Blick ungewöhnlich klingen, ist aber mathematisch korrekt, denn nach der Kettenregel ist der erste Faktor ja als Funktionswert von $f'$ an der Stelle $g(x)$ zu berechnen. Dies kann nicht mit `diff` angeschrieben werden, da `diff` Ableitungen von Funktions*ausdrücken*, nicht aber von Funktionen berechnet.

Ersetzen wir $f$ durch das „bekannte“ Funktionssymbol `sin`, so erhalten wir die aus der Kettenregel gewohnten Formeln.

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

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

Das zusätzliche `eval` ist erforderlich, da `subs` sehr eingeschränkt reagiert und nicht alle sonst automatisch verwendeten Regeln anwendet.

Solche *Funktionen von Funktionen* entstehen in verschiedenen mathematischen Zusammenhängen auf natürliche Weise, da auch Funktionen Gegenstand mathematischer Kalküle (etwa der Analysis) sind. Derartige Funktionen von Funktionen sind auch im informatischen Kontext, etwa im funktionalen Programmieren, gut bekannt.

Die Ableitung von $f$ an der Stelle $x$, die hier in Maple als `D(f)(x)` angeschrieben ist, kann in einigen CAS näher an der üblichen mathematischen Notation als `f'(x)` eingegeben werden. `f'` ist dabei allerdings keine neue syntaktische Einheit, sondern ebenfalls eine Funktion von Funktionen – das Ergebnis der Anwendung des Postfix-Operators `'` auf $f$.

Es ist wichtig und nicht nur aus mathematischer Sicht korrekt, zwischen der Ableitung $D(f)$ der Funktion $f$ und $D(f)(x)$ des Ausdrucks $f(x)$ zu unterscheiden; gerade dieser Umstand wird durch die beschriebene Notation berücksichtigt. Mathematisch ist

$$D : (\mathbb{R} \to \mathbb{R}) \to (\mathbb{R} \to \mathbb{R})$$

eine Funktion, welche die reelle Funktion $f$ auf die reelle Funktion $D(f)$ abbildet, $D(f) : \mathbb{R} \to \mathbb{R}$ dagegen selbst eine (reelle) Funktion.

Ein solches Konzept erlaubt es, Variablen- und Funktionsbezeichner auf einheitliche Weise zu behandeln. Dies ist in historisch älteren CAS-Designkonzepten wie etwa bei Maxima noch nicht in der Deutlichkeit berücksichtigt. Dort muss für komplexere Anwendungen mit Funktionsbezeichnern auch auf der Nutzerebene sehr genau zwischen Wert und Symbol unterschieden werden. In der Vorstellung eines Bezeichners `otto` als Box für den Wert steht in der Maxima-Notation `otto` für den Inhalt der Box und `'otto` für den Aufkleber auf der Box. Funktionen von Funktionen führen außerdem kein Eigenleben, sondern müssen stets durch weitere formale Parameter zu Funktionsausdrücken im klassischen Sinne ergänzt werden.

Ich werde dies am Beispiel der Berechnung der Ableitung von $\sin(\log(x))$ mit Maxima genauer darstellen. Dazu schalten wir mit `display2d:false` zunächst auf eindimensionale Ausgabe, um Formatierungseffekte auszublenden. Bei der Berechnung der Ableitung

```
u:diff(f(g(x)),x);
```

```
'diff(f(g(x)),x,1)
```

wird die Kettenregel nicht angewendet, sondern das Ergebnis als Ausdruck mit dem Kopf `'diff` zurückgegeben. Es wird also ein Funktionsausdruck mit diesem Kopf konstruiert, obwohl dem Bezeichner `diff` eine Funktionsdefinition zugeordnet ist. Das vorangestellte Zeichen `'` verhindert, dass diese Definition angewendet wird. Allerdings führt die Substitution

```
u1:ev(u,f=sin,g=log);
```

```
'diff(sin(log(x)),x,1)
```

selbst mit nachgeschalteter nochmaliger Auswertung auch nur zu einem Funktionsausdruck mit dem Kopf `'diff`. Es ist notwendig dem CAS mitzuteilen, dass die mit `diff` verbundene Funktionsdefinition an dieser Stelle *doch* angewendet werden soll.

```
u1, nouns;
```

```
cos(log(x))/x
```

Details über `nouns` und `verbs` zum Verständnis des Geschehens können an dieser Stelle nicht dargestellt werden. Ich verweise dazu auf das MAXIMA-Handbuch[4], das inzwischen auch vollständig ins Deutsche übersetzt wurde.

Die meisten Fehler in diesem Bereich sind auch nicht auf diese Feinheiten zurückzuführen, sondern auf die unreflektierte Verwendung derselben Bezeichner sowohl für Funktionen als auch für Funktionsausdrücke. So ist etwa nach der Eingabe

```
h:diff(x^2,x);
```

$h$ mitnichten die Ableitung der *Funktion* $x \to x^2$, sondern die Ableitung des *Ausdrucks* $x^2$ nach $x$. Der „Wert“ von $h$ an der Stelle 2 ergibt sich nicht durch Anschreiben von `h(2)`, sondern als `ev(h,x=2)` durch die Ersetzung $x = 2$ in $h$.

Eine Vereinbarung von $f$ als Ableitung der Funktion $x \to x^2$ ist mit keinem der Kommandos

```
f(x):=diff(x^2,x);
```

```
f(x)::=diff(x^2,x);
```

möglich, da zwar stets `f(y)` und auch `f(sin(z))` korrekt berechnet wird, aber der Aufruf `f(2)` mit einer Fehlermeldung abbricht. Das Problem ist, dass `:=` und `::=` die rechten Seiten nicht auswertet und damit `f(2)` stets als `diff(4,2)` (miss)-verstanden wird.

Eine solche Nichtauswertung der rechten Seite von Funktionsdefinitionen ist allerdings sehr sinnvoll, denn normalerweise soll ja die zu definierende Funktion erst zum Aufruf abgearbeitet werden und nicht schon während der Definition. Allerdings nicht in diesem Beispiel, denn hier soll die Ableitung bereits zur *Definitionszeit* der Funktion berechnet und diese andere Funktionsdefinition unter dem Bezeichner $f$ gespeichert werden. Das ist mit dem MAXIMA-Kommando `define` möglich:

```
define(f(x), diff(x^2,x));
```

definiert die gewünschte Funktion $f : x \to 2\,x$ korrekt, wie man am Aufruf von `f(z)` und `f(2)` leicht nachprüft.

[4] `http://maxima.sourceforge.net/documentation.html`

# 7 Rechnen mit Funktionen

Funktionen von Funktionen, sofern im entsprechenden CAS verfügbar, können auch ein Eigenleben führen, wie die folgenden MAPLE-Beispiele zeigen. Leider lassen sich diese Beispiele in MAXIMA aus den oben dargelegten Gründen eines anderen Zugangs zu Funktionen, in dem Funktionen von Funktionen kein Eigenleben führen können, so nicht anschreiben.

Die Ableitung `D(sin)` der Sinusfunktion wird erwartungsgemäß als `cos`, also korrekt als Kosinusfunktion zurückgegeben.

Interessanter ist das Ergebnis für die Ableitung `f:=D(ln)` der Logarithmusfunktion. Hier steht die Frage, wie das Ergebnis genau anzuschreiben ist, denn die Ableitung der Logarithmusfunktion hat keinen allgemein anerkannten Namen. Jedenfalls wären „Umkehrfunktion“ oder „inverse Funktion“ höchst mehrdeutige Begriffe. MAPLE gibt deshalb $\left(a \mapsto a^{-1}\right)$ zurück. Das Ergebnis ist eine Funktion, von der zwar die Anwendungsvorschrift bekannt ist, die aber keinen eigenen Namen besitzt.

Eine solche Funktion wird auch als *namenlose Funktion* (pure function) bezeichnet. Sie ist das Gegenteil eines Funktionssymbols, von dem umgekehrt der Name, aber keine Anwendungsvorschrift bekannt war. Namenlose Funktionen entstehen auf natürliche Weise in Antworten des CAS auf Problemstellungen, in denen Funktionen zu konstruieren sind, wie etwa beim Integrieren und allgemeiner beim Lösen von Differenzialgleichungen.

Weist man solchen Ausdrücken einen Bezeichner zu, so lassen sich diese Funktionen auch in gewohnter Weise aufrufen.

```
f(z)
```

$$\frac{1}{z}$$

Es ist sogar denkbar, dass nicht nur der Name, sondern auch die genaue Zuordnungsvorschrift nicht bekannt ist und nur eine semantische Spezifikation der Funktion als „Black Box“ existiert wie im Ergebnis der folgenden Interpolationsaufgabe in MATHEMATICA:

```
points = {{0,0},{1,1},{2,3},{3,4},{4,3},{5,0}};
ifun = Interpolation[points]
```

$$\text{InterpolatingFunction}[\{\{0, 5\}\}, <>]$$

`ifun` ist eine auf dem Intervall $[0, 5]$ definierte Interpolationsfunktion durch die angegebenen Punkte `points`, welche die im Handbuch angegebenen Eigenschaften hat, von der aber diesmal nicht einmal die genaue Funktiondefinition durch entsprechende Optionen angezeigt werden kann.

Solche Antworten entstehen zum Beispiel, wenn das Ergebnis Funktionen enthält, die nur in vorcompilierter Form vorliegen. Auch derartige Funktionen können einem Bezeichner zugewiesen und dann zur grafischen Visualisierung, hier zum Beispiel mit `Plot[ifun[x],{x,0,5}]`, aufgerufen werden.

# 8 Und nun etwas Praktisches

Es mag scheinen, dass die Ausführungen der letzten Abschnitte Spitzfindigkeiten sind, die mit dem alltäglichen Gebrauch von CAS in der Schule kaum etwas zu tun haben. Ich möchte dazu abschließend ein Beispiel präsentieren, um diese Frage *praktisch* zu erörtern. Es ist dem Sächsischen Abitur 2001[5], Leistungskurs Mathematik, entnommen. Die Originalaufgabe enthält zwei weitere Aufgabenteile a) und d).

> In einem kartesischen Koordinatensystem sind für jedes $t$ ($t \in \mathbb{R}$, $t > 0$) die Punkte $A(6,0,0)$, $B_t(8,t^2,0)$, $C_t(4,3\,t,0)$ und $D(2,2,0)$ gegeben.
>
> Jedes Viereck $AB_tC_tD$ ist die Grundfläche einer Pyramide mit der Spitze $S(5,3,6)$.
>
> b) Berechnen Sie für $t = 1$ den Schnittwinkel zwischen der Seitenflächenebene $AB_1S$ und der Grundflächenebene.
>
> c) Zeigen Sie, dass es genau einen Wert $t$ gibt, für den die zugehörige Pyramide eine quadratische Grundfläche besitzt, und bestimmen Sie diesen Wert.

Der Parameter $t$, der hier als Index der Eingangsgrößen $B_t$ und $C_t$ angeschrieben ist, lässt eine zweifache Interpretation zu – einmal als *symbolischer Parameter* der „einfachen Aufgabenstellung“ und zum anderen als *formales Funktionsargument*, wenn $B_t$ und $C_t$ als Funktionen $B(t)$ und $C(t)$ betrachtet werden. Für eine CA-gestützte Diskussion der Aufgabe sind beide Varianten des Anschreibens möglich, allerdings ist es wichtig, die beiden Zugänge genau zu unterscheiden und nicht zu vermischen, denn sonst geschehen schnell „wundersame Dinge“. Ich möchte im Weiteren nacheinander beide Varianten am Beispiel der Behandlung der Aufgabenstellung mit MAXIMA vorstellen und beginne mit der Variante, die $t$ als Symbol verwendet.

**Variante 1:** Die gegebenen Größen können als

```
A:[6,0,0]; B:[8,t^2,0]; C:[4,3*t,0]; D:[2,2,0]; S:[5,3,6];
```

eingeführt werden. Koordinaten von Punkten und Vektoren werden in MAXIMA nicht unterschieden und beides als Listen notiert. Im Weiteren wird das Skalarprodukt von Vektoren benötigt, das als `A . B` angeschrieben werden kann.

**b)** Der Schnittwinkel zwischen den Ebenen ist gleich dem Winkel zwischen den zugehörigen Normalenvektoren $n_0$ der Ebene $AB_1S$ und $[0,0,1]$, dem Einheitsvektor in $z$-Richtung.

Der mathematisch einfachste Zugang ist sicher die Berechnung von $n_0$ über das Kreuzprodukt

$$n_1 = \overrightarrow{AB} \times \overrightarrow{AS}\,|_{t=1}, \quad n_0 = \frac{n_1}{\sqrt{n_1 \cdot n_1}}\,.$$

Leider ist das Kreuzprodukt in MAXIMA in einer schwer verständlichen Form im Paket `vect` versteckt. In einem solchen Fall sollte stets die Alternative erwogen werden, ob es nicht schneller und einfacher ist, die Funktion selbst zu definieren. Das ist hier zweifelsohne der Fall; mit der Funktionsdefinition

[5] `http://www.sn.schule.de/~matheabi/01/ma01la.htm`

```
kreuzprodukt(a,b):=
    [a[2]*b[3]-a[3]*b[2],a[3]*b[1]-a[1]*b[3],a[1]*b[2]-a[2]*b[1]];
```

lässt es sich für zwei Vektoren $a$, $b$ bestimmen.

Die Koordinaten von $\overrightarrow{AS}$ ergeben sich als $S - A$. Hier wird verwendet, dass MAXIMA „mitdenkt“ und die „Differenz“ zweier Listen sehr nutzerfreundlich als Liste der Differenzen der jeweiligen Elemente interpretiert und berechnet. Natürlich ist klar, was wirklich passiert – MAXIMA führt an dieser Stelle eine entsprechende Funktionstransformation automatisch aus.

Die Lösung des Aufgabenteils b) ergibt sich nun aus folgender Rechnung, wobei im letzten Schritt die Umrechnung von Bogen- in Gradmaß vorgenommen wird.

```
n1:kreuzprodukt(A-subst(t=1,B),S-A);
n0:n1/sqrt(n1.n1);
r1:acos(n0.[0,0,1]);
r2:180-float(r1*180/%pi);
```

Dieser einfache geometrische Zugang ist nicht mehr gangbar, wenn Curricula das Kreuzprodukt nicht mehr enthalten. Er kann durch die Interpretation der Koeffizienten der Gleichung $a_1x+a_2y+a_3z+a_4 = 0$ der durch $AB_1S$ aufgespannten Ebene $\varepsilon$ als $n_1 = (a_1, a_2, a_3)$ ersetzen werden. Dazu sind aus den Bedingungen $A, B_1, S \in \varepsilon$ drei lineare Gleichungen abzuleiten und das entstehende homogene lineare Gleichungssystem zu lösen.

```
gl:a1*x+a2*y+a3*z+a4;
```

$$a_3z + a_2y + a_1x + a_4$$

```
s1:map("=",[x,y,z],A);
```

$$[x = 6, y = 0, z = 0]$$

```
s2:map("=",[x,y,z],B),t=1;
```

$$[x = 8, y = 1, z = 0]$$

```
s3:map("=",[x,y,z],S);
```

$$[x = 5, y = 3, z = 6]$$

```
sys:map(lambda([u],ev(gl,u)),[s1,s2,s3]);
```

$$[a_4 + 6\,a_1,\ a_4 + a_2 + 8\,a_1,\ a_4 + 6\,a_3 + 3\,a_2 + 5\,a_1]$$

```
sol:solve(sys,[a1,a2,a3,a4]);
```

$$\left[\left[a_1=-\frac{\%r}{6},\ a_2=\frac{\%r}{3},\ a_3=-\frac{7\,\%r}{36},\ a_4=\%r\right]\right]$$

Natürlich müssen die Rechnungen nicht so kompakt angeschrieben werden wie dies hier aus Gründen der praktischen Demonstration geschehen ist. Wir erhalten eine einparametrige Lösungsschar, aus der eine „schöne“ Lösung durch Einsetzen von `%r=36` gewonnen werden kann:

```
sol,%r=36;
```

$$[[a_1=-6,\ a_2=12,\ a_3=-7,\ a_4=36]]$$

Daraus gewinnen wir ebenfalls

```
n1:[-6,12,-7];
```

und können die Rechnungen weiter wie oben angegeben ausführen.

**c)** Hierfür ist zunächst zu überlegen, wie geprüft werden kann, ob ein Viereck ein Quadrat ist. Ein Viereck ist genau dann ein Quadrat, wenn es

- erstens ein Parallelogramm ist, also $\overrightarrow{AB}=\overrightarrow{DC}$ gilt,
- zweitens benachbarte Seiten gleichlang sind, also $\overrightarrow{AB}^2=\overrightarrow{AC}^2$ gilt,
- und drittens diese senkrecht aufeinander stehen, also $\overrightarrow{AB}\cdot\overrightarrow{AC}=0$ gilt.

Wir erhalten damit ein System von drei Gleichungen. Während die erste dieser Gleichungen eine Vektorgleichung ist, die in drei skalare Gleichungen expandiert werden kann, sind die anderen beiden skalare Gleichungen. Aus Gründen, die weiter unten klar werden, wollen wir die Bedingungen zu einer Funktionsdefinition `istQuadrat` zusammenfassen, mit der für vier übergebene Punkte geprüft werden kann, ob diese ein Quadrat aufspannen. Die Definition

```
istQuadrat(A,B,C,D):=
    [(A-B)=(D-C), (A-B).(A-B)=(A-D).(A-D), (D-A).(B-A)=0];
```

fasst die Gleichungen zusammen und für die vier Punkte

```
P1:[0,1]; P2:[1,0]; P3:[0,-1]; P4:[-1,0];
```

können wir prüfen, dass diese in der Tat ein Einheitsquadrat aufspannen:

```
sys:istQuadrat(P1,P2,P3,P4));
apply("and",sys);
```

liefert den Wert `true` zurück – mit dem zweiten Kommando werden die drei Einträge der Liste `sys` „und“-verknüpft.

Für unser Viereck $ABCD$ ist die Situation etwas komplizierter, da ja nicht zu prüfen ist, ob es sich um ein Quadrat handelt, sondern es die Werte $t$ zu bestimmen gilt, für die das so ist.

```
sys:istQuadrat(A,B,C,D);
```

liefert hierfür eine Liste von Bestimmungsgleichungen

$$\left[\left[-2,-t^2,0\right]=\left[-2,2-3\,t,0\right],\ t^4+4=20,\ 2\,t^2-8=0\right],$$

für die das Kommando `solve(sys,t)` leider keine Lösung findet. Der Grund hierfür liegt in der Mischung von Vektor- und skalaren Gleichungen in `sys`. Einige CAS wie zum Beispiel MATHEMATICA erlauben eine solche Mischung, MAXIMA dagegen nicht. Wir müssen also `istQuadrat` so definieren, dass alle Gleichungen skalar sind:

```
istQuadrat(A,B,C,D):=
    append( map("=",(A-B),(D-C)),
        [(A-B).(A-B)=(A-D).(A-D), (D-A).(B-A)=0]
    );
```

Nun liefern

```
sys:istQuadrat(A,B,C,D);
```

$$\left[-2=-2,\ -t^2=2-3\,t,\ 0=0,\ t^4+4=20,\ 2\,t^2-8=0\right]$$

und `solve(sys,t)` die einzige Lösung $t=2$ der Aufgabe. Mit den folgenden Kommandos lassen sich die Koordinaten der Eckpunkte der Grundfläche für $t=2$ berechnen und die Quadratkonditionen noch einmal hinschreiben – man überzeugt sich durch diese Probe, dass alles stimmt.

```
ev([A,B,C,D],t=2);
```

$$[[6,0,0],\ [8,4,0],\ [4,6,0],\ [2,2,0]]$$

```
ev(istQuadrat(A,B,C,D),t=2);
```

$$[-2=-2,\ -4=-4,\ 0=0,\ 20=20,\ 0=0]$$

Eigentlich müsste man alle Kombinationen von $ABCD$ prüfen, ob sie ein Quadrat ergeben können, wegen verschiedener Symmetrien also $ABCD$, $ABDC$, $ACBD$. Die anderen beiden Systeme enthalten aber Gleichungen $2=-2$ bzw. $2=-6$ als $x$-Koordinate im Vektorvergleich. Dies können wir schnell durch Berechnen von `istQuadrat(A,B,D,C)` bzw. `istQuadrat(A,C,B,D)` prüfen.

**Variante 2:** Schauen wir uns nun die Variante der Lösung an, wenn $B_t$ und $C_t$ als Funktionen $B(t)$ und $C(t)$ angeschrieben werden. Dazu müssen wir die Eingangsgrößen modifizieren:

```
A:[6,0,0]; B(t):=[8,t^2,0]; C(t):=[4,3*t,0]; D:[2,2,0]; S:[5,3,6];
```

Die Lösung für Aufgabenteil b) kann fast genauso wie oben angeschrieben werden, allein zwei Modifikationen sind erforderlich:

```
n1:kreuzprodukt(A-B(1),S-A);
s2:map("=",[x,y,z],B(1));
```

Für Teil c) definieren wir eine weitere Funktion

```
QuadratTest(t):=istQuadrat(A,B(t),C(t),D);
```

Der Aufruf von `QuadratTest(t)` liefert dann wie oben die Bestimmungsgleichungen für $t$, der Aufruf `QuadratTest(2)` zeigt, dass $t = 2$ wirklich eine Lösung ist.

# 9 Schluss

Ich komme abschließend auf meine Eingangsbemerkungen zurück und die Frage, ob es auch eine eigenständige Bedeutung für die Allgemeinbildung hat, Erfahrungen mit computeralgebraischen Werkzeugen zu sammeln und kritisch zu verarbeiten.

Ob es zu Fausts Zeiten anders gewesen ist, mag dahingestellt bleiben – heute jedenfalls, in einem hochtechnisierten Umfeld, hat das handlungsmächtige *Wort* ganz zentrale Bedeutung. Denn auch ohne natürlichsprachliche Eingabeeinheiten „gehorcht“ diese Technik (nur) aufs Wort. Genau so ist sie konstruiert, und es legt nicht zuletzt jede lange, in Bildersprache geschriebene IKEA-Montageanweisung Zeugnis davon ab. Ohne Lesekompetenz, Interpretationskompetenz, Methodenkompetenz und adäquaten Werkzeugeinsatz ist noch kein von dort gelieferter Schrank zusammengebaut worden.

Dies gilt für die technisierten Teile unserer Welt in Gänze – sie sind nach Worten (Konstruktionsbeschreibungen) konstruiert, nach Worten (Montagebescheibung) ist sie zusammenzubauen und Betriebsbereitschaft herzustellen und nach Worten (Betriebsanleitung) ist sie zu betreiben. Jeden einzelnen Wunsch muss man dieser technisierten Umgebung mit Worten – nach Worten eingeübte Handbewegungen eingeschlossen – vermitteln. Allerdings ist es nicht wie im Märchen, dass die Technik nur zu Gutem fähig wäre und das Böse, wenn es denn auftritt, allein den Bösewicht, wenigstens aber stets die anderen, trifft. In jedem Fall ist sie – das hat Fukushima ein weiteres Mal gelehrt – potenziell unbeherrschbar, weil *kollateralschadensfähig*. H.-P. Dürr thematisiert in [3] zum Ende des Kapitals 2 dieses Zusammenwachsen wissenschaftlichen, experimentellen und handwerklichen Tuns in der heutigen Zeit genauer, das mit Galilei, Newton und Leonardo da Vinci sowie dem Aufkommen der *western science* im 17. Jahrhundert seinen Anfang nahm.

Die Verwendung eines leistungsfähigen CAS in der Schule erzeugt prototypisch genau diese Situation: Die Schülerinnen und Schüler begegnen einem sehr mächtigen, dennoch von menschlichem Intellekt geschaffenen technischen Artefakt, dem man seine eigenen Wünsche antragen kann. Allerdings ist es eine Simulation – man kann dem CAS unaufgeregt begegnen und bei Nichtgelingen einfach den „roten Knopf“ drücken. Die Kollateralschadensfähigkeit *dieses* Werkzeugs hält sich in Grenzen.

Mit CAS (wie auch der Informatik) hat die Technik ihren Fuß in der Tür des (bundesdeutschen) Gymnasiums. Die mit solchen technischen Werkzeugen gesammelten positiven und

negativen Erfahrungen werden das Bild des Einzelnen vom Umgang mit Technik überhaupt nachhaltig prägen. Gut ausgewählte und motivierende Beispiele – ich hatte dies exemplarisch im letzten Abschnitt am Beispiel einer Abituraufgabe in Ansätzen erläutert – können diesen Prozess entscheidend befördern, den Schülerinnen und Schülern allgemein den Zugang zu derartigem Werkzeugeinsatz erschließen helfen und speziell den Wert einer „voll integrierten Umgebung für technisches Rechnen“ (Steven Wolfram über MATHEMATICA) für professionelles Arbeiten deutlich machen.

## Literatur


[1] M. Kofler, H.-G. Gräbe: *Mathematica 6. Einführung, Grundlagen, Beispiele.* Fünfte, aktualisierte Auflage. Pearson Studium, München 2007.

[2] H.-G. Gräbe: Skript zum Kurs *Einführung in das symbolische Rechnen.* `http://www.informatik.uni-leipzig.de/~graebe/skripte`

[3] H.-P. Dürr: *Warum es ums Ganze geht. Neues Denken für eine Welt im Umbruch.* oekom Verlag, München 2009.