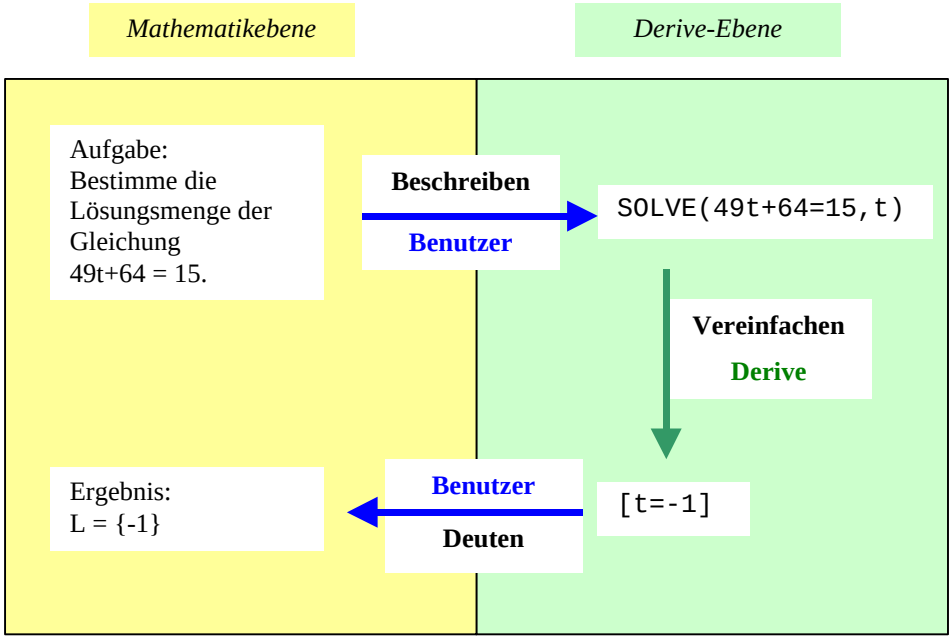
Problemlösen mit dem Computeralgebrasystem *DERIVE* informatisch betrachtet



Klaus Becker

Mai 2000

1. Informatisch fundierter Mathematikunterricht

Computeralgebrasysteme wie DERIVE verändern den Mathematikunterricht. Mit ihrer Hilfe können neue Inhalte erarbeitet und alte Inhalte methodisch neu erschlossen werden (vgl. hierzu etwa [Noll&Schmitt 1997]). Ziel dieser Arbeit ist es nicht, diese Veränderungsmöglichkeiten anhand weiterer Beispiele aufzuzeigen – in der fachdidaktischen Literatur finden sich hierzu eine Vielzahl von Unterrichtsvorschlägen (vgl. z. B. [Materialien 1995]). Vielmehr sollen in dieser Arbeit die beim Einsatz von DERIVE benötigten Denkstrukturen in den Vordergrund gestellt werden.

DERIVE ist ein Werkzeug, das dem Benutzer eine Vielzahl von „Diensten“ zur Verfügung stellt, mit deren Hilfe mathematische Operationen automatisiert ausgeführt werden können. Obwohl die Dienste recht benutzerfreundlich aufbereitet sind, so erfordert ihr Einsatz doch ein Grundverständnis für die ihnen zugrunde liegenden informatischen Strukturen. Die Anwendung von DERIVE-Diensten setzt immer eine funktional-deklarative Beschreibung der zu erledigenden Aufgabe voraus. Hierbei kommen eine Reihe informatischer Strukturen wie funktionale Modellierung und Datentypen zum Tragen. Die Ausführung der Dienste basiert auf dem Reduktionskonzept. Ein Verständnis dieses Konzepts kann so manches von DERIVE gelieferte merkwürdige Ergebnis erklären. Will man die Gestaltungsmöglichkeiten, die das Werkzeug DERIVE zur Verfügung stellt, voll nutzen, so erfordert dies auch ein beträchtliches Maß an informatischem Grundlagenwissen, das über reines Bedienerwissen hinausgeht. Dieses Grundlagenwissen soll im Folgenden zusammengestellt werden.

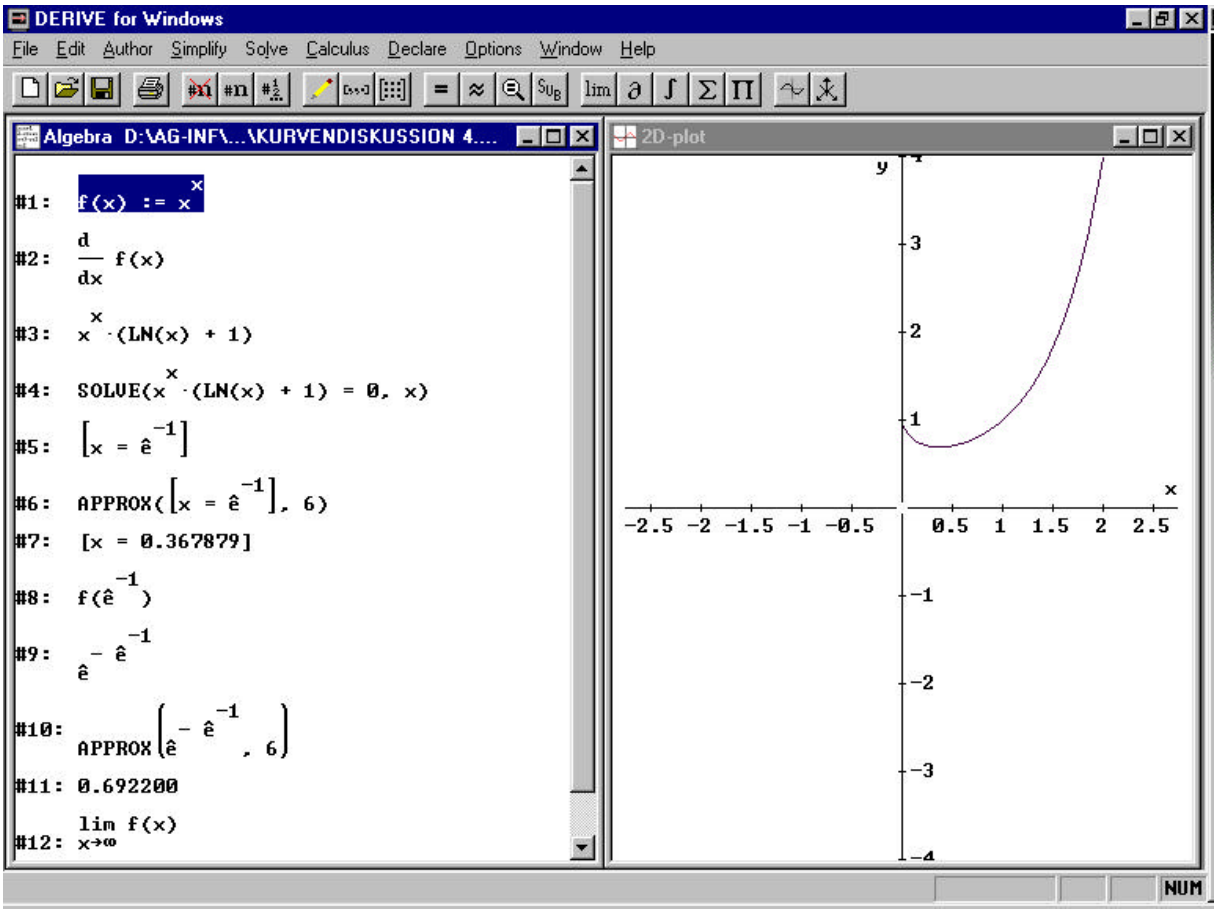
Für den Einsatz von DERIVE im Mathematikunterricht ergeben sich meiner Meinung nach folgende Konsequenzen: Ein mit Computeralgebrasystemen arbeitender Mathematikunterricht sollte teilweise fachübergreifend angelegt werden, indem auch Strukturen und Konzepte des Fachs Informatik thematisiert werden. So kann wohl am ehesten gewährleistet werden, dass die für den Einsatz von DERIVE benötigten Denkstrukturen auch tatsächlich entwickelt werden. Die folgenden Ausführungen werden anhand unterrichtsnaher Beispiele durchgeführt und liefern Material für einen informatisch fundierten Mathematikuntericht (vgl. auch [Wagenknecht 1999]).

2. Beschreiben, Vereinfachen, Deuten

Beispiel 1

Problem:
Wie verhält sich die Funktion $f(x) = x^x$ bei wachsendem bzw. kleiner werdendem x? Gibt es lokale Maxima und Minima? Wenn ja, an welchen Stellen?

Lösung:
Die folgende Abbildung zeigt eine typische DERIVE-Lösung zu diesem Problem.



Die graphische Veranschaulichung liefert erste Befunde: Für $x \to 0$ nähert sich der Funktionswert f(x) dem Wert 1. Für $x \to \infty$ geht f(x) gegen unendlich. In der Nähe von 0.4 liegt ein lokales Minimum vor. Einen genaueren Wert erhält man durch Anwendung der Differentialrechnung: Das lokale Minimum liegt an der Stelle $x = e^{-1}$.

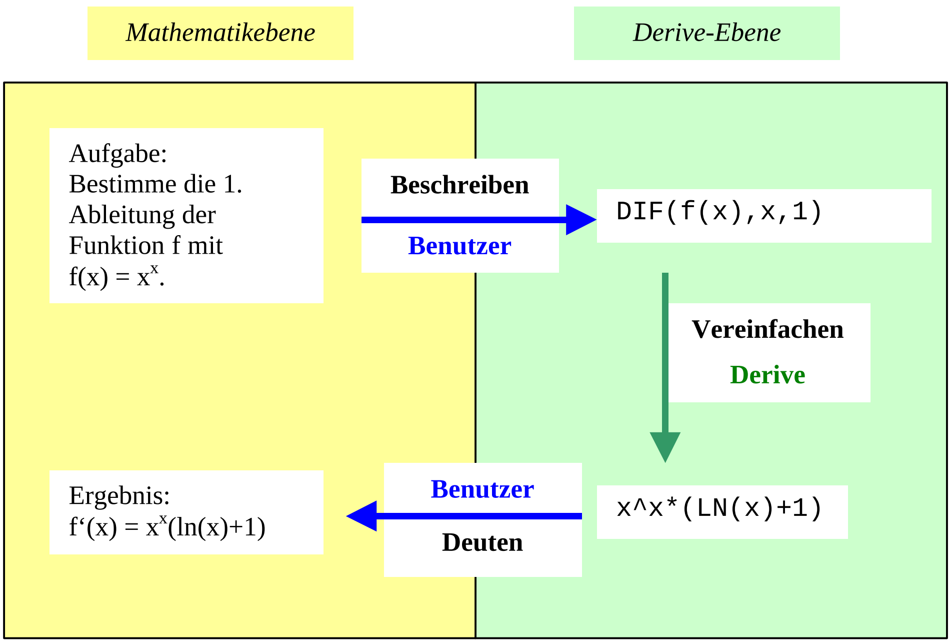
Analyse

Wie wird das Computeralgebrasystem DERIVE hier zum Lösen des Problems eingesetzt? Die erste fundamentale Strategie basiert auf Veranschaulichung. Viele qualitative Aussagen lassen sich direkt aus graphischen Abbildungen ablesen. Dabei ist natürlich Vorsicht geboten. Leicht kann man zu unsicheren oder auch fehlerhaften Vermutungen kommen (begrenzter Ausschnitt, begrenztes Auflösevermögen, etc.). Will man präzise und abgesicherte Aussagen treffen, so muss man den Gegenstand einer mathematischen Analyse unterziehen. Im obigen

Fall werden zu diesem Zweck die lokalen Extrema bestimmt. Der hierbei benutzte Kalkül ist Standard. Interessant ist, wie man mit DERIVE diesen Kalkül umsetzt.

Der erste Schritt besteht darin, die Aufgabe in die Sprache von DERIVE zu übersetzen, d. h. mit Hilfe eines DERIVE-Ausdrucks zu beschreiben. In einem zweiten Schritt wird der DERIVE-Ausdruck von DERIVE vereinfacht. Der von DERIVE erzeugte neue Ausdruck muss schließlich in einem dritten Schritt mathematisch gedeutet werden.

Die folgende Übersicht zeigt anhand eines Beispiels diese drei Schritte.

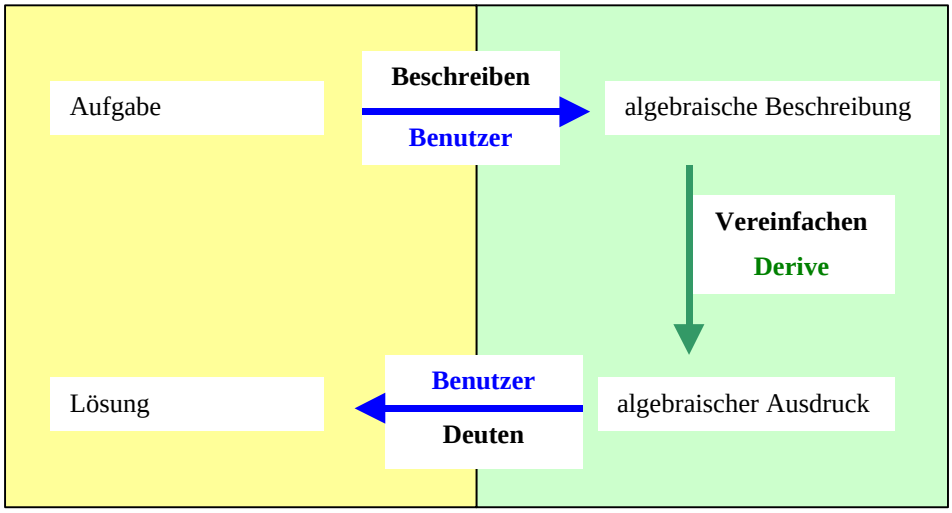


Besonders deutlich werden die Schritte „Beschreiben“ und „Vereinfachen“, wenn man sich ein – von DERIVE – kommentiertes Protokoll anschaut. Hier der Anfang eines solchen Protokolls:

```
                  x
#1:  f(x)  :=  x                                                     User

      ⎛d ⎞1
#2:  ⎜——⎟   f(x)                                                     User
      ⎝dx⎠

         x
#3:  x  ·(LN(x)  +  1)                                             Simp(#2)

               x
#4:  SOLVE(x  ·(LN(x)  +  1)  =  0,  x)                              User

      ⎡       -1⎤
#5:  ⎣x  =  ê   ⎦                                                  Simp(#4)
```

Man findet Beschreibungen von Aufgaben, die vom Benutzer (User) eingegeben wurden und Ergebnisausdrücke, die DERIVE durch Vereinfachen (Simplify) erzeugt hat.

Was ist hieran interessant und wichtig? DERIVE löst letztlich die Aufgabe im Vereinfachungsschritt. Der Benutzer überträgt somit das Ausführen von Routinen an DERIVE. Damit dies überhaupt möglich wird, muss der Benutzer die Aufgabe DERIVE mitteilen können und das von DERIVE gelieferte Ergebnis verstehen können. Die Grundstrategie besteht also in dem Dreischritt Beschreiben – Vereinfachen – Deuten.



Was liegt dieser Arbeitsweise aus informatischer Sicht zu Grunde? Die Informatik entwickelt u. a. Sprachen, mit deren Hilfe man Problemlöseverfahren erfassen kann. Sie unterscheidet dabei zwischen Kommandosprachen („mache erst das, dann das, dann das usw.“) und Beschreibungssprachen („dies sei so, das sei so usw.“). Kommandosprachen werden in der Informatik als ***imperative Sprachen***, Beschreibungssprachen als ***deklarative Sprachen*** bezeichnet. Da die Sprache, die bei einer Problemlösung benutzt wird, in natürlicher Weise das Denken beeinflusst, sollte man sich die „Grundstruktur“ der benutzten Sprache bewusst machen. DERIVE arbeitet mit einer Beschreibungssprache. Die zu lösende Aufgabe (Bestimme die 1. Ableitung der Funktion f) wird genau beschrieben (DIF(f(x),x,1)) und zur Lösung an DERIVE übertragen.

Didaktische und methodische Hinweise

Ein verständiges Arbeiten mit dem Werkzeug DERIVE setzt ein Grundverständnis dieses Werkzeugs voraus. Der Benutzer sollte wissen, wie man vorgeht, wenn man mit DERIVE ein Problem lösen will. Hierzu ist es erforderlich, adäquate mentale Modelle über die Arbeitsweise von DERIVE aufbauen. Dies kann meines Erachtens nur dann erfolgen, wenn der deklarative Charakter von DERIVE erkannt wird. Ungünstig ist das – in der Fachliteratur durchaus praktizierte – Verfahren, den Erklärungsmodellen imperative Strukturen zu Grunde zu legen.

Für Schülerinnen und Schüler, die bereits Erfahrungen im Problemlösen mit dem Computer gemacht haben und hierbei ausschließlich imperative Sprachkonzepte benutzt haben, könnten sich Umstellungsschwierigkeiten ergeben. Sie müssten eventuell umdenken lernen.

Wie entwickelt man nun adäquate Grundvorstellungen? Die wichtigste Strategie besteht darin, den Dreischritt „Beschreiben – Vereinfachen – Deuten“ zu thematisieren. Anlass hierzu kann ein kommentiertes DERIVE-Protokoll (siehe z. B. oben) sein, das verstanden werden soll. Ein

weiterer Anlass kann darin bestehen, eine DERIVE-Sitzung deklarativ zu dokumentieren. Die oben wiedergegebene DERIVE-Sitzung könnte etwa wie folgt dokumentiert werden:

Aufgabe	*algebraische Beschreibung*	*Ergebnis der (algebraischen) Vereinfachung*
Führe die Funktion ein.	`f(x):=x^x`	
Bestimme die 1. Ableitung.	`DIF(f(x),x)`	`x^x*(LN(x)+1)`
Bestimme die Nullstellen der Ableitungsfktn. (1. Ordnung)	`SOLVE(x^x*(LN(x)+1)=0,x)`	`[x=e^(-1)]`
Berechne einen Näherungswert.	`APPROX([x=e^(-1)],6)`	`[x=0.367879]`
Bestimme den Funktionswert an der Stelle e^(-1)	`f(e^(-1))`	`e^(-e^(-1))`
Berechne einen Näherungswert	`APPROX(e^(-e^(-1)),6)`	`0.692200`
Bestimme den Grenzwert von f(x) für x gegen Unendlich.	`LIM(f(x),x,inf,0)`	`inf`

3. Aufgaben beschreiben

Ziel dieses Abschnittes ist es genauer zu analysieren, wie eine Aufgabenbeschreibung mit DERIVE strukturiert ist. Wir betrachten zunächst ein Beispiel.

Beispiel 2

Problem:
Der Punkt A(2|0|10) soll an der Ebene E: $2x_1+3x_2+6x_3-15 = 0$ gespiegelt werden. Bestimme die Koordinaten des Spiegelpunktes B.

Dokumentation einer DERIVE-Lösung:

Aufgabe	*Beschreibung*	*Ergebnis*
Definiere Punkt A / Ortsvektor a	`a:=[2,0,10]`	
Definiere Punkt P / Ortsvektor p	`p:=[0,5,0]`	
Definiere Normalenvektor n	`n:=[2,3,6]`	
Definiere eine Gerade g	`g(t):=a+t*n`	
Definiere die Ebene E	`E(x):=(x-p)*n=0`	
Beschreibe den Schnittpunkt von g und E	`E(g(t))`	`49*t+64=15`
Bestimme die Lösung der resultierenden Gleichung	`SOLVE(49*t+64=15,t)`	`[t=-1]`
Bestimme den Schnittpunkt	`g(-1)`	`[0,-3,4]`
Benenne den Schnittpunkt	`s:=[0,-3,4]`	
Benenne und bestimme den Spiegelpunkt	`b:=a+2*(s-a)`	`[-2,-6,-2]`

Welche Erkenntnisse lassen sich hieraus über die Art und Weise gewinnen, wie eine Aufgabenbeschreibung mit DERIVE strukturiert ist? Hierzu müssen wir uns anschauen, wie mathematische Objekte und Operationen mit DERIVE repräsentiert werden.

Wir betrachten die folgenden Aspekte:
a) Mathematische Objekte müssen als Datenobjekte repräsentiert werden, damit eine maschinelle Bearbeitung erfolgen kann. DERIVE stellt hierzu Datentypen zur Verfügung.
b) Will man Datenobjekte weiterverarbeiten, so ist es günstig, Namen für die zu verarbeitenden Objekte einzuführen. DERIVE erlaubt es, mittels Deklaration solche Namen einzuführen.
c) DERIVE erlaubt es, komplexe Operationen durch funktionale Ausdrücke (Terme) zu repräsentieren.

Diese Aspekte sollen im Folgenden näher betrachtet werden.

Repräsentation elementarer mathematischer Objekte

Wie werden mathematische Objekte wie Zahlen, Zahlentupel oder Gleichungen in DERIVE dargestellt? Welche Operationen können in DERIVE mit diesen Objekten durchgeführt

werden? Das oben wiedergegebene Protokoll zeigt bereits, dass hier eine Reihe von Dingen zu beachten ist. So werden etwa Zahlentripel mit Hilfe von eckigen Klammern dargestellt. Zahlentripel können addiert werden, Zahlentripel können mit Zahlen multipliziert werden, Zahlentripel können aber auch selbst multipliziert werden. All diese Gegebenheiten können mit Hilfe des informatischen Konzepts „Datentyp“ beschrieben werden.

Informatisches Konzept: *Datentyp*

Datentypen dienen in der Informatik dazu, gleichartige bzw. gleich zu behandelnde Objekte zu beschreiben.

Ein ***Datentyp*** wird festgelegt durch eine Menge von Objekten sowie Operationen zur Bearbeitung der Objekte.

Für die Benutzung von Operationen benötigt man ihre Signaturen. Eine ***Signatur*** gibt die Eingabe- und Ausgabebereiche der Operation mit Hilfe von Datentypen an.

Einige grundlegende Datentypen, die in DERIVE zur Verfügung stehen, sollen hier kurz vorgestellt werden. Weitere Datentypen werden in den folgenden Abschnitten eingeführt.

Datentyp: ganze Zahl
Objekte: ..., -2, -1, 0, 1, 2, ...
Operationen: +, -, *, /, MOD, GCD, LCM, PRIME, NEXT_PRIME, ...

Eine genauere Beschreibung der Operationen erfolgt mit Hilfe ihrer Signaturen. Für die Operation „PRIME“ ist beispielsweise der Eingabebereich durch den Datentyp „ganze Zahl“ und der Ausgabebereich durch den Datentyp „Wahrheitswert“ festgelegt. Man beachte, dass ein Eingabebereich nicht identisch mit dem mathematischen Definitionsbereich einer Operation sein muss. So sind bei der Operation „MOD“ die Eigaben 1 und 0 durchaus erlaubt, auch wenn dies mathematisch zu keinem sinnvollen Ergebnis führt.

Signatur der Operation	*Beispiel für eine Anwendung*
+: ganze Zahl, ganze Zahl → ganze Zahl	3 + (-2) → 1
-, *: analog	
/: ganze Zahl, ganze Zahl → Bruchzahl	3 / 7 → 3/7
MOD: ganze Zahl, ganze Zahl → ganze Zahl	MOD(5, 2) → 1
GCD: ganze Zahl, ganze Zahl → ganze Zahl	GCD(56, 24) → 8
LCM: ganze Zahl, ganze Zahl → ganze Zahl	LCM(56, 24) → 168
PRIME: ganze Zahl → Wahrheitswert	PRIME(33) → false
NEXT_PRIME: ganze Zahl → ganze Zahl	NEXT_PRIME(33) → 37
...	

Datentyp: Dezimalzahl
Objekte: Dezimalbrüche der Gestalt 2.13
Operationen: +, -, *, /, ...
Die Signaturen der Operationen sind analog zu bilden.

Datentyp: Zahlentripel
Objekte: Zahlentripel bestehen aus drei Objekten vom Typ ganze Zahl oder Dezimalzahl; die Darstellung benutzt Listen (siehe Abschnitt 8); Bsp.: [0, 6.1, 5]
Operationen: +, -, * (skalare Multiplikation), * (Skalarprodukt), | | (Betrag), CROSS (Vektorprodukt)

Signatur der Operation	*Beispiel für eine Anwendung*
+: Zahlentripel, Zahlentripel → Zahlentripel	[3, 3, 4] + [5, 6, 2] → [8, 9, 6]
-: Zahlentripel, Zahlentripel → Zahlentripel	[3, 3, 4] – [5, 6, 2] → [-2, -3, 2]
*: Zahl, Zahlentripel → Zahlentripel	2 * [4, -1, 7] → [8, -2, 14]
*: Zahlentripel, Zahlentripel → Zahl	[3, 0, 4] * [1, 6, 2] → 11 (= 3*1 + 0*6 + 4*2)
| | : Zahlentripel → Zahl	| [1, 0, 0] | → 1
CROSS: Zahlentripel, Zahlentripel → Zahlentripel	CROSS([3,1,4], [1,2,3]) → [-5,-5,5]

Beachte:
Es findet eine Überladung des Zeichens „*“ statt: Dieses Zeichen wird – wie auch in der Mathematik – in verschiedenen Bedeutungen benutzt.

Datentyp: Gleichung
Objekte: Eine Gleichung besteht aus zwei Termen. Eine Gleichung wird in DERIVE – genau wie in der Mathematik – in der Form t1 = t2 dargestellt, wobei t1 und t2 die beiden die Gleichung konstituierenden Terme bezeichnen.
Bsp.: (x-p)*n=0
Operationen:
SOLVE (löst eine Gleichung), LHS (liefert die linke Seite der Gleichung), RHS (liefert die rechte Seite der Gleichung)

Signatur der Operation	*Beispiel für eine Anwendung*
SOLVE: Gleichung, Variable → Liste mit einer Gleichung	SOLVE(49*t+64=15,t) → [t=-1]
LHS: Gleichung → Term	LHS(49*t+64=15) → 49*t+64
RHS: Gleichung → Term	RHS(49*t+64=15) → 15

Gewinnung neuer mathematischer Objekte

DERIVE bietet die Möglichkeit, Objekten einen Namen zuzuodnen. Dies erfolgt mit Hilfe des Symbols „:=“.
Beispiele:

```
a:=[2,0,10]
g(t):=a+t*n
E(x):=(x-p)*n=0
```

Zur Klärung der Bedeutung des Symbols „:=“ erfolgt ein kurzer Exkurs in die Informatik.

Informatische Konzepte: *Wertzuweisung und Deklaration*

Das Symbol „:=“ wird in der Informatik in unterschiedlichen Bedeutungen benutzt.

In vielen Programmiersprachen (wie z. B. PASCAL) bewirkt das Symbol eine ***Wertzuweisung***. Einer Variablen wird ein Wert zugewiesen. Hier liegt die sog. ***Behältersemantik*** vor: Eine Variable dient als Abstraktion eines Speicherplatzes (Behälter, der Daten aufnehmen kann). Der Speicherplatz kann immer wieder neu belegt werden. So sind Wertzuweisungen der Form x := x+1 natürliche Darstellungen von Operationen, die den Wert der Variablen (Inhalt des Behälters) verändern.

Das Symbol „:=“ kann auch die Bedeutung einer ***Deklaration*** haben: Es wird ein Name eingeführt, mit dem ein Objekt bezeichnet werden soll. Mit Hilfe des Namens kann das Objekt jederzeit angesprochen werden, es sei denn, der Name wird neu vergeben. Dann verliert die ursprüngliche Bindung ihre Gültigkeit. Eine Deklaration erfolgt mit Hilfe einer Konstanten oder einer Funktion.
Bsp.: a := 1. Hier wird die Konstante „a“ eingeführt, um die Zahl „1“ abstrakt zu beschreiben.
Bsp.: f(x) := x+1. Hier wird die Funktion „f“ eingeführt, um die Zuordnung „x → x+1“ zu beschreiben. Der funktionale Ausdruck „f(x)“ repräsentiert dann den Term „x+1“.
Da Konstanten auch als nullstellige Funktionen (das sind Funktionen mit null Argumenten) angesehen werden können, kann man allgemein von ***Funktionsdeklarationen*** sprechen.

Die Bedeutung von „:=“ in DERIVE

In welcher Weise wird das Symbol „:=“ in DERIVE benutzt? Ist ein Ausdruck wie „`n:=[2,3,6]`“ eine Wertzuweisung, bei der der Variablen „n“ der Wert „[2,3,6]“ zugewiesen wird, oder handelt es sich um eine Konstantendeklaration, bei der eine Konstante mit Namen „n“ eingeführt wird, um das Tripel „[2,3,6]“ abstrakt zu repräsentieren? Diese Frage ist gar nicht so leicht zu beantworten. Baumann spricht in [Baumann 1998] von Wertzuweisungen. Zur Philosophie von DERIVE würde das Deklarationskonzept passen. In [Köhler 1995] wird aber gezeigt, wie man höchst merkwürdige Phänomene mit Hilfe des Symbols „:=“ erzeugen kann, die nicht mit dem Deklarationskonzept erklärt werden können. Diese schwer durchschauenbaren Phänomene sind aber eher unerwünscht. Meines Erachten sollte man den deklarativen Charakter hervorheben und durch einen disziplinierten Gebrauch die unerwünschten Phänomene vermeiden. Wie so etwas praktisch aussieht, wird anhand einiger Regeln im Folgenden aufgezeigt.

Regel 1:
Namen sollten nur einmal vergeben werden. Viele Fehler bzw. merkwürdige Ergebnisse entstehen dadurch, dass ein bereits vergebener Name nochmals vergeben wird.

Regel 2:
Will man einen Konstantenwert vorerst nicht festlegen, dann sollte man das so machen:
a :=
Hier wird eine neue Konstante namens „a“ eingeführt, deren Wert erst später festgelegt wird, z. B. durch eine Deklaration
a := 3.

Regel 3:
Will man eine Größe systematisch variieren, dann sollte man diese Größe nicht durch eine Konstante beschreiben, sondern eine neue Funktion einführen.
Bsp.: s. u.

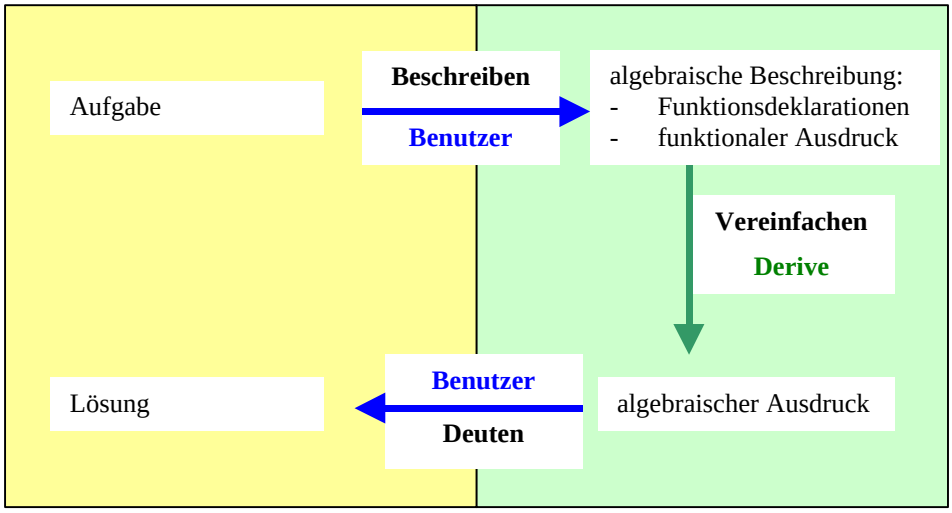
Funktionale Ausdrücke als Aufgabenbeschreibungen

Mit funktionalen Ausdrücken versucht man, Aufgabenbeschreibungen vorzunehmen. Einige Beispiele sollen dies aufzeigen:

verbale Beschreibung	*Beschreibung mit einem DERIVE-Ausdruck*
Schnittpunkt von g und E	`E(g(t))`
Lösung der resultierenden Gleichung	`SOLVE(49*t+64=15,t)` bzw. `SOLVE(E(g(t)),t)`
Punkt auf g	`g(-1)`

Die wesentlichen Bausteine einer Aufgabenbeschreibung sind die Operatoren bzw. Funktionen, mit denen der Ausdruck aufgebaut wird. Man benutzt hierzu vordefinierte Operatoren (wie z. B. den SOLVE-Operator) und die neu definierten Funktionen. Des weiteren können vordefinierte Objekte (wie z. B. Zahlen) und Variablen (die z. B. in Gleichungen vorkommen) benutzt werden. Der Aufbau erfolgt nach dem Schachtelungsprinzip bzw. Kompositionsprinzip.

Wir sind jetzt in der Lage, eine differenzierte Sichtweise des Dreischritts „Beschreiben – Vereinfachen – Deuten“ einzunehmen. Die algebraische Beschreibung der Aufgabe erfolgt mit funktionalen Ausdrücken und (Hilfs-) Deklarationen. Letztlich beschreiben die Ausdrücke die zu lösenden Aufgaben. Beim Vereinfachen wird schließlich der Wert des jeweiligen Ausdrucks ermittelt. Dieser Wert stellt die Lösung der Aufgabe dar, die nur noch richtig gedeutet werden muss. Wie DERIVE den jeweiligen Wert bestimmt, ist nicht ersichtlich und in den meisten Fällen auch für den Benutzer nicht wichtig – ein mögliches Erklärungsmodell wird dennoch in Abschnitt 11 vorgestellt.



Didaktisch-methodische Hinweise:

Das Beschreiben von Aufgaben mit Hilfe von funktionalen Ausdrücken ist für Schülerinnen und Schüler eher ungewohnt. Obwohl die Mathematik vielfach deklarative Züge aufweist, so sind doch operationale Vorgehensweisen im Mathematikunterricht vorherrschend (Ausführen von Lösungsroutinen). Beim Problemlösen mit DERIVE sollte man aber die deklarative Vorgehensweise in den Vordergrund stellen. Es bietet sich an, das Beschreiben von Aufgaben selbst im Unterricht zu thematisieren. Ein Beispiel: Die Verwendung vordefinierter Operatoren wie z. B. des Differenzierungsoperators wird von DERIVE dadurch erleichtert, dass der Aufbau des funktionalen Ausdrucks menügesteuert erfolgen kann. Lädt man die Beschreibung in den DERIVE-Editor, so kann man erkennen, dass intern der Ausdruck mit Hilfe des DIF-Operators aufgebaut wird.

4. Mit Funktionen modellieren

Beispiel 3 (Verallgemeinerung von Beispiel 2)

Problem:
Man möchte nicht nur den Punkt A an der Ebene bzgl. P und n spiegeln, sondern auch andere Punkte.

Wir zeigen zunächst eine zweite Lösung für das Problem aus Beispiel 2.

Dokumentation einer DERIVE-Lösung:

Aufgabe	*Beschreibung*	*Ergebnis*
Definiere Punkt A / Ortsvektor a	`a:=[2,0,10]`	
Definiere Punkt P / Ortsvektor p	`p:=[0,5,0]`	
Definiere den Normalenvektor n	`n:=[2,3,6]`	
Bestimme den Normaleneinheitsvektor zu n und nenne ihn n0	`n0:=n/ABS(n)`	`[2/7,3/7,6/7]`
Bestimme den orientierten Abstand von A zur Ebene bzgl. P und n	`d:=(a-p)*n0`	`7`
Benenne und bestimme den Spiegelpunkt	`b:=a-2*d*n0`	`[-2,-6,-2]`

Ziel ist es, diese Lösung zu verallgemeinern. Hierzu modellieren wir geeignete Funktionen, mit deren Hilfe aus jeweils gegebenen Objekten neue Objekte bestimmt werden können. Wir zerlegen zunächst das Ausgangsproblem in geeignete Teilprobleme.

Problem: Bestimme den Spiegelpunkt.
Teilprobleme: Bestimme den Normaleneinheitsvektor. Bestimme den (orientierten) Abstand.

Jetzt führen wir die Funktionen „sp“, „n0“ und „d“ zur Beschreibung der Probleme ein. Diese Funktionen werden zunächst genau spezifiziert. Hierzu benutzen wir sogenannte Schnittstellenbeschreibungen.

Informatisches Konzept: *Schnittstellenbeschreibung*

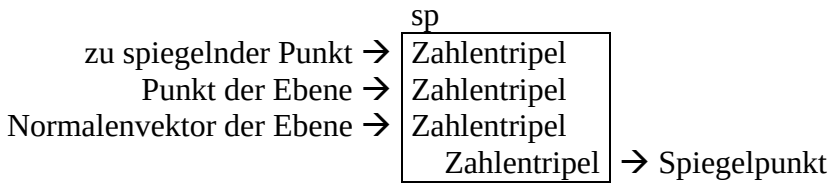
Eine ***Schnittstellenbeschreibung*** legt genau fest, welche Dienstleistungen ein System bzw. eine Funktionseinheit (hier in unserem Fall eine Funktion) der Umgebung zur Verfügung stellt. Schnittstellen müssen genau spezifiziert werden. Im vorliegenden Fall – Spezifikation einer Funktion – muss genau festgelegt werden, was die Funktion leistet. Wir können dies mit Hilfe einer Signaturangabe und einer Verhaltensbeschreibung erreichen.

Eine ***Signatur*** legt die Eingabebereiche und den Ausgabebereich genau fest. Eingabe- und Ausgabebereiche werden üblicherweise mit Hilfe der zur Verfügung stehenden Typen beschrieben.

Eine ***Verhaltensbeschreibung*** soll den funktionalen Zusammenhang zwischen Ein- und Ausgabewerten möglichst genau festlegen. Wir versuchen hier, diesen Zusammenhang informell und anhand von Beispiele zu beschreiben.

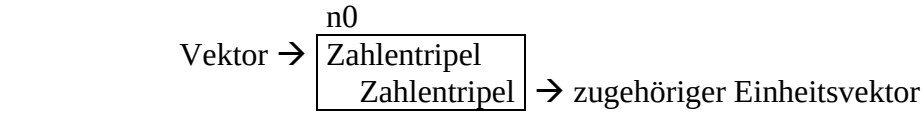
In Kurzform werden die Signatur und die Verhaltensbeschreibung durch eine graphische Box-Notation (eine Art Datenflussdiagramm) dargestellt.

Schnittstellenbeschreibung: sp



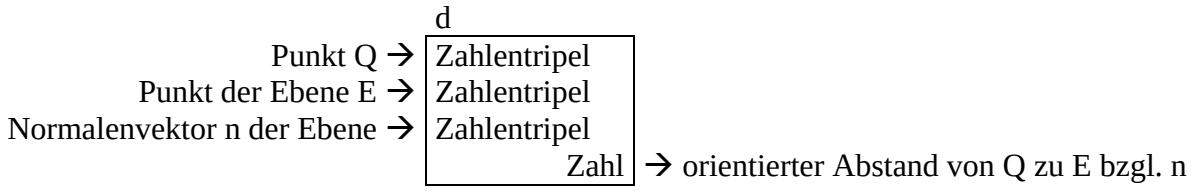
Beispiel:
sp([2,0,10], [0,5,0], [2,3,6]) beschreibt den Spiegelpunkt von A(2,0,10) bzgl. der Ebene mit Stützvektor $\vec{p}$ = (0,5,0) und Normalenvektor $\vec{n}$ = (2,3,6).

Schnittstellenbeschreibung: n0



Beispiele:
n0([2,3,6]) beschreibt liefert den Einheitsvektor zum Vektor (2,3,6).

Schnittstellenbeschreibung: d



Beispiele:
d([2,0,10], [0,5,0], [2,3,6]) beschreibt liefert den orientierten Abstand von Q(2,0,10) bzgl. der Ebene mit Stützvektor (0,5,0) und Normalenvektor (2,3,6).

Realisierung der Funktionen in DERIVE

Das folgende DERIVE-Protokoll zeigt, wie diese Funktionen in DERIVE dargestellt werden.

```
n0(v):=v/ABS(v)
d(q,p,n):=(q-p)*n0(n)
sp(a,p,n):=a-2*d(a,p,n)*n0(n)
```

Ein Test kann mit DERIVE wie folgt ausgeführt werden:

```
a1:=[2,0,10]
p1:=[0,5,0]
n1:=[2,3,6]
sp(a1,p1,n1)
[-2,-6,-2]
```

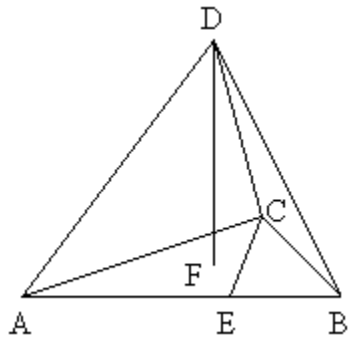
Didaktisch-methodische Hinweise

Die Entwicklung des funktionalen Denkens gehört zu den grundlegenden fachspezifisch allgemeinen Lernzielen des Mathematikunterrichts. Das Computeralgebrasystem DERIVE eignet sich besonders, dieses Denken zu fördern. Das oben aufgeführte Beispiel zeigt, wie man mit DERIVE den Funktionsbegriff nutzt, um komplexe Sachverhalte zu modellieren. Die funktionale Modellierung bereitet allerdings Schülerinnen und Schülern z. T. große Schwierigkeiten. Als Grund kann hier vermutet werden, dass die Schülerinnen und Schüler z. T. über verengte Funktionsbegriffe (nur eine Variable; Darstellung in der Form f(x) = ..., usw.) verfügen. Die Bearbeitung von Problemen der oben gezeigten Art könnte hier helfen, einen allgemeinen und tragfähigen Funktionsbegriff aufzubauen.

5. Automatisierung durch Funktionskomposition

Beispiel 4

Problem:
Es soll ein Berechnungsverfahren entwickelt werden, mit dessen Hilfe man das Volumen einer dreiseitigen Pyramide aus den Koordinaten der Eckpunkte bestimmen kann.
Beispiel:
Bei der Eingabe der Eckpunkte A(1,1,0), B(2,-4,5), C(6,7,11), D(3,1,2) wird das Volumen der entsprechenden Pyramide berechnet.



Es ergeben sich die folgenden Teilprobleme:
- Berechnung des Volumens einer Pyramide
- Berechnung des Flächeninhalts eines Dreiecks
- Berechnung des Abstands zweier Punkte
- Berechnung des Abstands Punkt-Gerade
- Berechnung des Abstands Punkt-Ebene

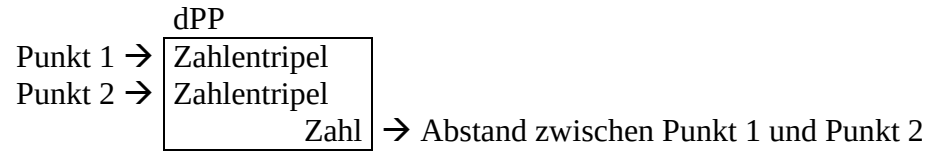
Zur Berechnung der jeweiligen Werte werden die folgenden Formeln benutzt:

Berechnungsformeln:

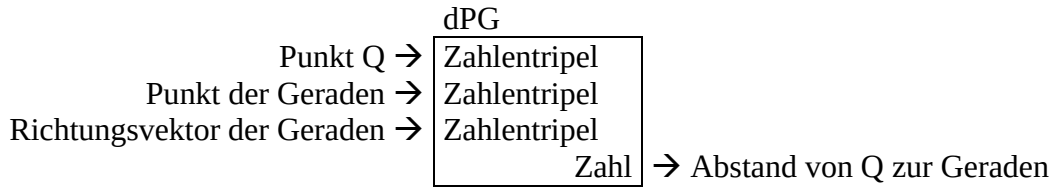
Abstand Punkt-Punkt:	$d_{PP} = \lvert \vec{q} - \vec{r} \rvert$
Abstand Punkt-Gerade:	$d_{PG} = \sqrt{(\vec{r} - \vec{p})^2 - \left((\vec{r} - \vec{p}) \cdot \frac{\vec{u}}{\lvert \vec{u} \rvert} \right)^2}$
Abstand Punkt-Ebene:	$d_{PE} = \left\lvert (\vec{r} - \vec{p}) \cdot \frac{\vec{n}}{\lvert \vec{n} \rvert} \right\rvert$
Flächeninhalt eines Dreiecks:	$A_D = \frac{1}{2} g \cdot h$
Volumen einer Pyramide:	$V_P = \frac{1}{3} G \cdot h$

Wir führen jetzt Funktionen ein, die zur Bearbeitung der Teilprobleme benutzt werden können. Diese Funktionen werden zunächst spezifiziert.

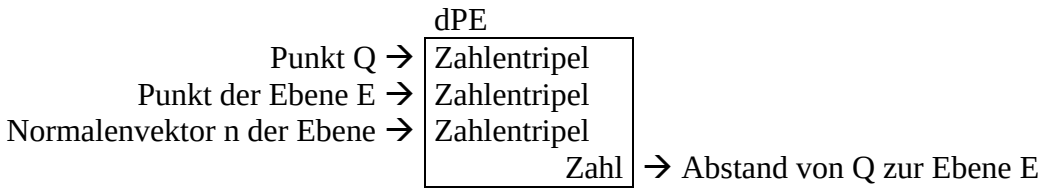
Schnittstellenbeschreibung: dPP



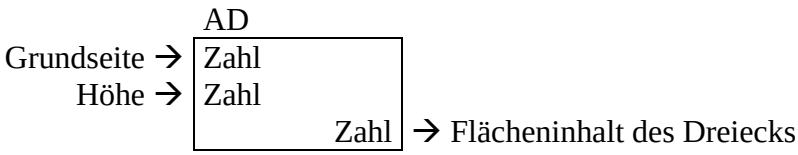
Schnittstellenbeschreibung: dPG



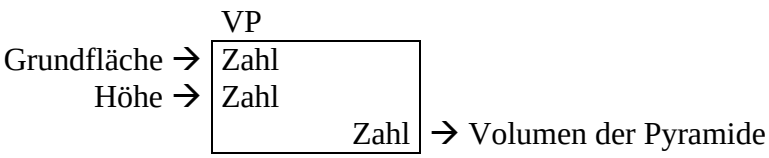
Schnittstellenbeschreibung: dPE



Schnittstellenbeschreibung: AD



Schnittstellenbeschreibung: VP



Wir implementieren diese Funktionen in DERIVE mit Hilfe der oben genannten Formeln wie folgt:

$$dPP(r, p) := |p - r|$$

$$dPG(r, p, u) := \sqrt{\left((r - p)^2 - \left((r - p)\cdot\frac{u}{|u|}\right)^2\right)}$$

$$dPE(r, p, n) := \left|(r - p)\cdot\frac{n}{|n|}\right|$$

$$AD(g, h) := \frac{1}{2}\cdot g\cdot h$$

$$VP(G,\ h)\ :=\ \frac{1}{3}\cdot G\cdot h$$

Mit Hilfe dieser Funktionen können jetzt die interessierenden Größen berechnet werden.

```
a := [1, 1, 0]
b := [2, -4, 5]
c := [6, 7, 11]
d := [3, 1, 2]
dPP(a, b)
√51
dPG(c, a, b - a)
 √47498
--------
   17
AD(dPP(a, b), dPG(c, a, b - a))
 √8382
-------
   2
dPE(d, a, CROSS(b - a, c - a))
 18·√8382
----------
   1397
VP(AD(dPP(a, b), dPG(c, a, b - a)), dPE(d, a, CROSS(b - a, c - a)))
18
```

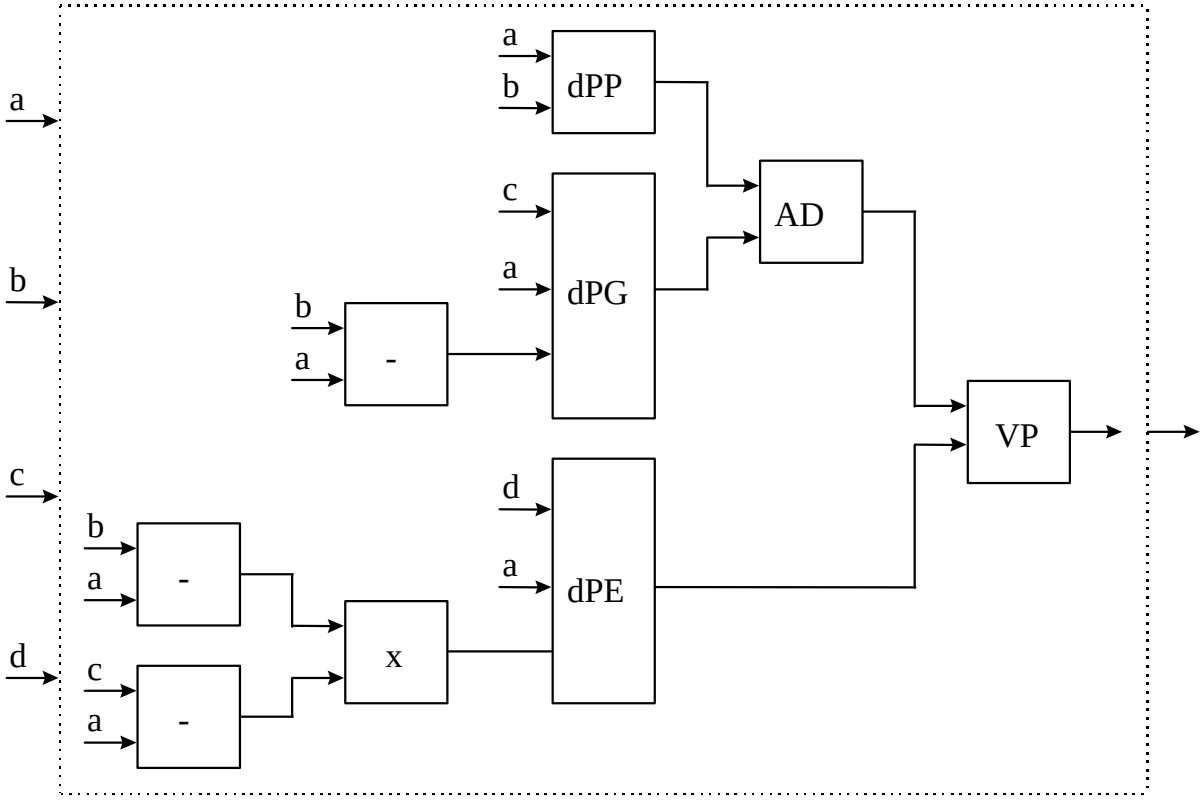
Es fällt auf, dass die interessierenden Größen hier mit Hilfe von zum Teil recht komplexen Ausdrücken beschrieben werden. Die folgende Auflistung zeigt noch einmal diese Ausdrücke, jetzt aber in die Sprache der Mathematik übersetzt.

$\overline{AB}$	$dPP(\vec{a},\vec{b})$
$\overline{CE}$	$dPG(\vec{c},\vec{a},\vec{b}-\vec{a})$
A_{ABC}	$AD(dPP(\vec{a},\vec{b}),dPG(\vec{c},\vec{a},\vec{b}-\vec{a}))$
$\overline{DF}$	$dPE(\vec{d},\vec{a},(\vec{b}-\vec{a})\times(\vec{c}-\vec{a}))$
V_{ABCD}	$VP(AD(dPP(\vec{a},\vec{b}),dPG(\vec{c},\vec{a},\vec{b}-\vec{a})),dPE(\vec{d},\vec{a},(\vec{b}-\vec{a})\times(\vec{c}-\vec{a})))$

($\vec{a},\vec{b},\vec{c},\vec{d}$ seien hier die Ortsvektoren der vorgegebenen Punkte.)

Die Ausdrücke sind alle mit Hilfe der Berechnungsfunktionen aufgebaut. Als Bauprinzip wird dabei das Kompositionsprinzip benutzt. Die folgende Abbildung visualisiert, wie die Berechnung des Volumens durch das Zusammensetzen der Funktionen erfolgt.

Visualisierung des Datenflusses bei der Funktionskomposition:



Wir können jetzt abschließend eine Funktion definieren, die das gesuchte Volumen automatisch aus den gegebenen Punkten berechnet.

```
V(a, b, c, d) := VP(AD(dPP(a, b), dPG(c, a, b - a)), dPE(d, a, CROSS(b -
a, c - a)))
```

Mit dieser Funktion lassen sich jetzt die oben gezeigte Volumenberechnung und eine Kontrollberechnung direkt durchführen.

```
V([1, 1, 0], [2, -4, 5], [6, 7, 11], [3, 1, 2])

18

V([0, 0, 0], [1, 0, 0], [0, 1, 0], [0, 0, 1])

 1
---
 6
```

Mathematisch-informatisches Konzept: *Kompositionsprinzip*

Funktionen können beliebig geschachtelt werden. Man spricht in diesem Zusammenhang auch von ***Funktionskomposition***. Funktionskomposition muss allerdings die Signaturen der beteiligten Funktionen beachten.
Bsp.:
Ein Ausdruck wie „`AD(dPP(a, b), dPG(c, a, b - a))`“ ist korrekt gebildet, da die Funktion „`AD`“ zwei Argumente vom Typ „Zahl“ erwartet und da die Funktionen „`dPP`“ und „`dPG`“ jeweils Ergebnisse vom Typ „Zahl“ liefern. Unzulässig (und hier natürlich auch unsinnig) wäre dagegen ein Ausdruck wie „`dPP(AD(a,b),AD(b,c))`“. Die Funktion „`AD`“ kann nicht mit Zahlentripeln als Argumenten aufgerufen werden. Ebenso kann die Funktion „`dPP`“ nicht mit Zahlen als Argumenten aufgerufen werden.

Durch Funktionskomposition können einzelne Berechnungsverfahren zu einem komplexeren Berechnungsverfahren zusammengesetzt werden. Funktionskomposition stellt demnach ein fundamentales Bauprinzip einer Programmierform dar, die als „funktionale Programmierung“ bezeichnet wird (mehr hierzu findet man in Abschnitt 12). Funktionale Programme bestehen aus Funktionsdeklarationen. Durch Funktionskomposition gewinnt man Programme, mit denen komplexe Berechnungsverfahren automatisiert werden können.

Didaktisch-methodische Hinweise

Bei der Entwicklung des Funktionsbegriffs ist in einem fortgeschrittenen Stadium anzustreben, dass die Schülerinnen und Schüler Funktionen als eigenständige Objekte erkennen. Erreicht werden kann dies z. B. durch die Erkenntnis, dass man Funktionen – genau wie andere mathematische Objekte auch – bestimmten Operationen unterwerfen kann. Eine solche Operation wäre z. B. die Verkettung von Funktionen. Durch Probleme der oben gezeigten Art kann man diesen Aspekt des Funktionsbegriffs gut aufzeigen. Durch das oben gezeigte Beispiel kann aber noch mehr erreicht werden. Es zeigt nämlich, dass man mit Funktionen auch programmieren kann. Der Funktionsbegriff kann so eine weitere Dimension gewinnen.

6. Mit Bausteinen arbeiten

Beispiel 5

Problem:
Ziel ist es, den Inhalt von Flächen „unter“ Funktionsgraphen automatisiert bestimmen zu können.

Entwicklung der Berechnungsformeln

Wir setzen zunächst voraus, dass f eine fest vorgegebene Funktion beschreibt (z. B. $x \rightarrow x^2$).
Sei $d = \frac{b-a}{n}$ die Streifenbreite (mit $n \in N$). Sei $s_i = a + i \cdot d$ (i = 0, ..., n-1) die i-te Unterteilungsstelle. Dann gilt für die Unter- bzw. Obersumme:

$$U_n = f(s_0) \cdot d + \cdots + f(s_{n-1}) \cdot d = \sum_{i=0}^{n-1} f(s_i) \cdot d$$

$$O_n = f(s_1) \cdot d + \cdots + f(s_n) \cdot d = \sum_{i=1}^{n} f(s_i) \cdot d$$

Analyse der Formeln aus informatischer Sicht
Zur mathematischen Beschreibung von Unter- und Obersumme sind vier Formeln entwickelt worden. Diese werden jetzt funktional modelliert. Hierzu werden die Funktionen „U“, „O“, „d“, „s“ eingeführt. Es folgt zunächst eine genaue Spezifikation dieser Funktionen mittels Schnittstellenbeschreibungen.

Schnittstellenbeschreibung: d
Die Funktion „d“ soll die Streifenbreite beschreiben.

d

untere Grenze →	Bruchzahl
obere Grenze →	Bruchzahl
Anzahl der Unterteilungen →	natürliche Zahl
	Bruchzahl → Streifenbreite

Beispiel:
d(0,1,10) beschreibt die Streifenbreite bzgl. des Intervalls [0; 1] bei 10 äquidistanten Unterteilungen, hier also den Wert 0.1.

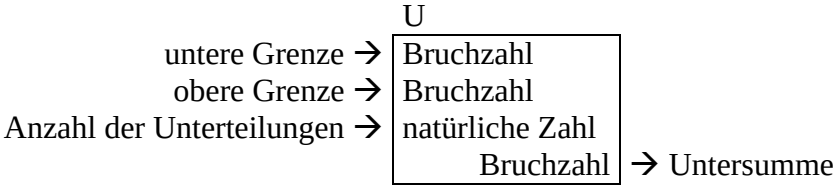
Schnittstellenbeschreibung: s
Die Funktion „s“ dient dazu, die Unterteilungsstellen zu beschreiben.

s

untere Grenze →	Bruchzahl
obere Grenze →	Bruchzahl
Anzahl der Unterteilungen →	natürliche Zahl
Nummer der Stelle →	natürliche Zahl
	Bruchzahl → Unterteilungsstelle

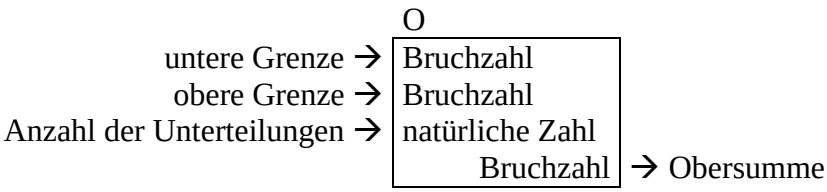
Beispiel:
s(0,1,10,2) beschreibt die 2. Unterteilungsstelle bzgl. des Intervalls [0; 1] bei 10 äquidistanten Unterteilungen; hier also den Wert 0.2.

Schnittstellenbeschreibung: U
Mit Hilfe der Funktion „U“ werden Untersummen beschrieben.



Beispiel:
U(0,1,10) beschreibt für die (vorgegebene) Funktion f die Untersumme bzgl. des Intervalls [0; 1] bei 10 äquidistanten Unterteilungen.

Schnittstellenbeschreibung: O
Mit Hilfe der Funktion „O“ werden Obersummen beschrieben.



Beispiel:
O(0,1,10) beschreibt für die (vorgegebene) Funktion f die Obersumme bzgl. des Intervalls [0; 1] bei 10 äquidistanten Unterteilungen.

Realisierung der Funktionen in DERIVE

$$f(x) :=$$

$$d(a, b, n) := \frac{b - a}{n}$$

$$s(a, b, n, i) := a + i \cdot d(a, b, n)$$

$$U(a, b, n) := \sum_{i=0}^{n-1} d(a, b, n) \cdot f(s(a, b, n, i))$$

$$O(a, b, n) := \sum_{i=1}^{n} d(a, b, n) \cdot f(s(a, b, n, i))$$

Die Benutzung der Funktionen zeigt das folgende DERIVE-Protokoll:

$$f(x) := x^2$$

```
U(0, 1, 10)

  57
─────
 200

U(0, 1, n)

    2
 2·n  − 3·n + 1
─────────────────
         2
      6·n

lim U(0, 1, n)
n→∞

 1
───
 3

lim U(a, b, n)
n→∞

   2          2
 (a  + a·b + b )·(b − a)
─────────────────────────
            3

lim O(a, b, n)
n→∞

   2          2
 (a  + a·b + b )·(b − a)
─────────────────────────
            3
```

Die Vorgehensweise oben kann wie folgt beschrieben werden: Ausgehend von einer Problemanalyse werden zunächst geeignete Funktionen entwickelt und implementiert. Diese können dann zur Lösung von Problemen benutzt werden.

Folgende Beobachtung führt jetzt einen Schritt weiter: Die Kenntnis der Schnittstellenbeschreibungen reicht aus, um die Funktionen in einem gegebenen Zusammenhang einsetzen zu können. Diese für die Informatik zentrale Erkenntnis wird mit dem Begriff „Modulkonzept“ beschrieben.

Informatisches Konzept: *Modul*

Ein ***Modul*** gliedert sich in eine Schnittstelle und einen Rumpf. In der Schnittstelle wird spezifiziert, welche Dienstleistungen der Modul seiner Umgebung zur Verfügung stellt. Der Rumpf enthält die Implementierung der in der Schnittstelle spezifizierten Dienstleistungen. Für die Benutzung eines Moduls ist nur die Kenntnis der Schnittstelle erforderlich.

Letzteres ist in einer immer komplexer werdenden Welt von großer Bedeutung. Man muss die „inneren“ Details eines Moduls nicht kennen und verstanden haben, um es sachgerecht benutzen zu können. Allerdings muss man über eine präzise Verhaltensbeschreibung des

Moduls verfügen. Wir zeigen dies jetzt anhand der folgenden Module zur Bestimmung und Veranschaulichung von Rechtecksummen. Module nennen wir im folgenden – wie es in der fachdidaktischen Literatur üblich ist – auch ***Bausteine***.

Baustein: Rechtecksummen / Rechteckfiguren

Dieser Baustein dient dazu, mit Hilfe der Funktionen „LS“ und „RS“ Rechtecksummen bzgl. einer beliebig vorgegebenen Funktion zu bestimmen. Des weiteren können mit diesem Baustein auch die entsprechenden Rechteckfiguren gezeichnet werden. Die vorgegebene Funktion muss dabei jeweils den Funktionen als Funktionsargument übergeben werden.

Schnittstellenbeschreibung: LS
Die Funktion „LS“ soll Rechtecksummen beschreiben, die sich an den Funktionswerten der linken Intervallgrenzen der Streifen orientieren. Also: Sind s_i und s_{i+1} die beiden Unterteilungsstellen, die einen Rechteckstreifen festlegen, so soll die Höhe des Streifens durch den Funktionswert $f(s_i)$ der „linken“ Unterteilungsstelle festgelegt werden.

LS	
Funktionsterm →	Term
untere Grenze →	Bruchzahl
obere Grenze →	Bruchzahl
Anzahl der Unterteilungen →	natürliche Zahl
	Bruchzahl → (linke) Rechtecksumme

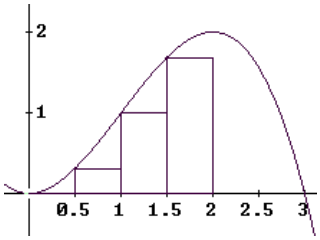
Beispiele für die Benutzung:
LS(x^2,0,1,10) beschreibt für die Funktion f mit $f(x) = x^2$ die linke Rechtecksumme (entspricht hier der Untersumme) bzgl. des Intervalls [0; 1] bei 10 äquidistanten Unterteilungen.

Analog hierzu wird die Funktion „RS“ zur Beschreibung der „rechten“ Rechtecksumme spezifiziert.

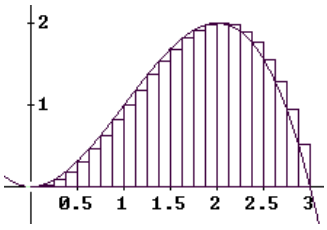
Schnittstellenbeschreibung: LRechteckfigur
Die Funktion „LRechteckfigur“ erzeugt die Rechteckfigur, die sich an den Funktionswerten der linken Intervallgrenzen der Streifen orientieren.

LRechteckfigur	
Funktionsterm →	Term
untere Grenze →	Bruchzahl
obere Grenze →	Bruchzahl
Anzahl der Unterteilungen →	natürliche Zahl
	Bruchzahl → (linke) Rechteckfigur

Beispiele für die Benutzung:
LRechteckfigur(-0.5x^3+1.5x^2, 0, 2, 4) erzeugt eine Rechteckfigur, die im Schaubild wie folgt aussieht:



LRechteckfigur(-0.5x^3+1.5x^2, 0, 3, 24) erzeugt eine Rechteckfigur, die im Schaubild wie folgt aussieht:



Analog hierzu wird die Funktion „RRechteckfigur“ zur Beschreibung der „rechten“ Rechteckfigur spezifiziert.

Die Anwendung der Bausteine soll jetzt anhand von Aufgaben gezeigt werden. Die Aufgaben lehnen sich an [Schmidt&Grabinger&Noll 1998] an und dienen dazu, den Unterschied zwischen Flächeninhalten und Integralen aufzuzeigen.

Aufgaben

Wir betrachten die Funktion f: x → $-x^2 + 1$.

a) Bestimme LS(f(x),0,1,2), LS(f(x),0,2,4) sowie LS(f(x),0,3,6). Was fällt auf?
b) Um diese Beobachtung erklären zu können, zeichne mit Hilfe der Funktion „LRechteckfigur“ die entsprechenden Rechteckfiguren. Berechne mit Hilfe von „LS“ den Beitrag eines Streifens, z. B. des Streifens bzgl. des Intervalls [2, 2.5]. Inwiefern werden die Ergebnisse von a) jetzt verständlich?
c) Es soll der Flächeninhalt der Fläche zwischen Graph f und der x-Achse im Intervall [0; 2] bestimmt werden.

a)

$$f(x) := -x^2 + 1$$

LS(f(x), 0, 1, 2)

$$\frac{7}{8}$$

LS(f(x), 0, 2, 4)

$$\frac{1}{4}$$

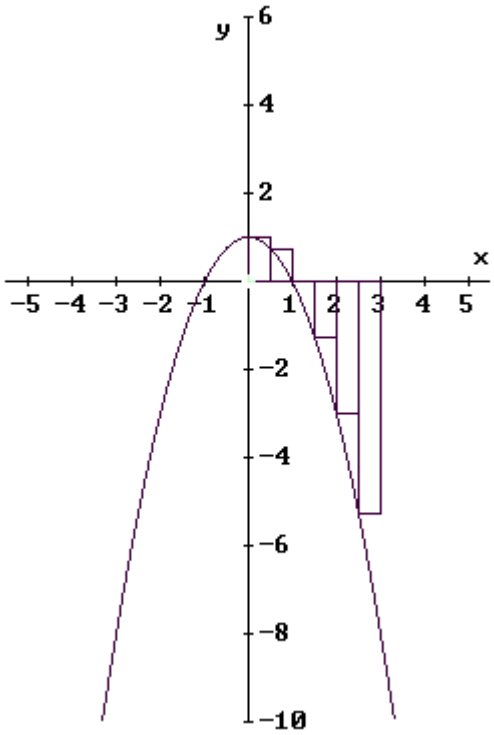
LS(f(x), 0, 3, 6)

$$-\frac{31}{8}$$

b)

```
LS(f(x), 2, 2.5, 1)
```

$$-\frac{3}{2}$$



c)

```
lim LS(f(x), 0, 1, n)
n→∞
```

$$\frac{2}{3}$$

```
lim LS(f(x), 1, 3, n)
n→∞
```

$$-\frac{20}{3}$$

$$A := \frac{2}{3} + \frac{20}{3}$$

$$\frac{22}{3}$$

Die Beispiele zeigen, dass man zum Umgang mit den Bausteinen keine Kenntnisse über deren Implementierung benötigt. Wer sich dennoch für die Implementierungen interessiert, sollte die Abschnitte 7 und 8 lesen.

Didaktisch-methodische Hinweise

Das Arbeiten mit Bausteinen ist eine der zentralen Methoden der Informatik. Hierzu gehören u. a. die folgenden Aktivitäten:

- Einen Baustein gemäß der gegebenen Schnittstellenbeschreibung benutzen.
- Einen Baustein entwerfen, d. h. die Schnittstellen konzipieren.
- Einen Baustein entwickeln, d. h. die Schnittstellen konzipieren und die Funktionseinheiten implementieren.
- Einen Baustein analysieren, d. h. die Details der Implementierung verstehen.

In der fachdidaktischen Literatur zum Einsatz von Computeralgebrasystemen im Mathematikunterricht werden die beiden folgenden Vorgehensweisen diskutiert:

(white-box)-(black-box)-Methode:
Erst wird der Baustein Schritt für Schritt entwickelt, dann wird die entwickelte Funktionseinheit eingesetzt, um weitere Probleme zu lösen.

(black-box)-(white-box)-Methode:
Erst wird ein gegebener Baustein gemäß seiner Schnittstellenbeschreibung benutzt, dann erst wird versucht zu verstehen, was sich dahinter versteckt.

In der heutigen Welt ist es oft so, dass man einen Baustein bzw. ein Modul nur gemäß seiner Schnittstellenbeschreibung benutzt. Die Realisierung des Moduls ist so kompliziert, dass es für einen Laien unmöglich ist, diese zu verstehen. Meines Erachtens hat daher auch die folgende Methode ihre Berechtigung im Mathematikunterricht:

(black-box-only)-Methode:
Ein Baustein wird nur gemäß seiner Schnittstellenbeschreibung benutzt.

Schülerinnen und Schülern fällt es gelegentlich schwer, Bausteine zu benutzen, ohne zu wissen, wie sie realisiert sind. Hier bestünde also die Möglichkeit, den sachgerechten Umgang mit dokumentierten Funktionseinheiten einzuüben.

7. Funktionen als Eingabeobjekte

Schauen wir uns die Schnittstellenbeschreibung der Funktion „LS“ (siehe Abschnitt 6) einmal genauer an. Hier wird neben Zahlen auch ein Funktionsterm als Eingabeobjekt erwartet. Die zu diesem Term gehörende Funktion soll zur Berechnung der Rechtecksummen benutzt werden. Die Funktion „LS“ stellt somit eine Funktion höherer Ordnung dar. Eines ihrer Argumente ist selbst wieder eine Funktion. Im Folgenden soll die Möglichkeit untersucht werden, mit DERIVE solche Funktionen höherer Ordnung zu erzeugen.

In DERIVE ist es problemlos möglich, einer Funktion Terme als Eingabeobjekte zu übergeben, wie das folgende Beispiel zeigt.

```
g(t1, t2) := t1 + t2

   2   2
g(x , y )

 2    2
x  + y

   2   2
g(x , x  + 1)

   2
2·x  + 1
```

Schwierig wird es aber, wenn ein übergebener Term als Funktion fungieren soll. DERIVE stellt dem Benutzer zunächst keine Möglichkeit zur Verfügung, die dem Term zugrunde liegende Funktion auf ein beliebig vorgegebenes Argument anzuwenden. Wir versuchen, mit DERIVE eine Funktion h höherer Ordnung zu definieren, die als Argument eine Funktion f hat. Die Eingabe

```
h(f, y1, y2) := f(y1) + f(y2)
```

wird in DERIVE wie folgt umgesetzt:

```
h(f, y1, y2) := f·y1 + f·y2
```

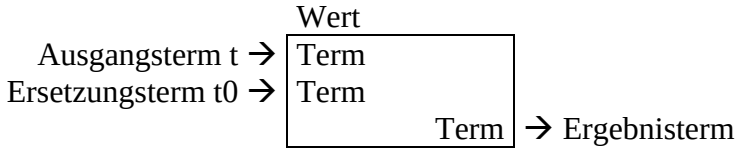
DERIVE interpretiert „f“ hier als eine Konstante. Auch der Eingabeversuch

```
h(f(x), y1, y2) := f(y1) + f(y2)
```

scheitert. DERIVE meldet einen Syntaxfehler.

DERIVE scheint also das Erstellen von Funktionen höherer Ordnung nicht vorzusehen. Mit Hilfe eines – sagen wir – Tricks wird es aber doch möglich. Was ist zu tun? Wir müssen selbst eine Funktion erstellen, die die Funktionsanwendung bei einem vorgegebenen Term simuliert. Betrachten wir zunächst als Beispiel die Funktion „x → x+x“. Angenommen, wir wollen diese Funktion auf das Argument „y-1“ anwenden. Hierzu muss der Wert des Funktionsterms „x+x“ für das Argument „y-1“ bestimmt werden. Als Ergebnis soll sich dann der neue Term „(y-1)+(y-1)“ oder vereinfacht „2y-2“ ergeben. Um diese Termersetzung und anschließende Vereinfachung durchzuführen, führen wir eine neue Funktion „Wert“ ein. Eine Schnittstellenbeschreibung sieht wie folgt aus:

Schnittstellenbeschreibung: Wert
Die Funktion „Wert“ dient dazu, in einem vorgegebenen Term „t“ die dort vorkommende Variable – wir setzen stets voraus, dass nur eine in diesen Term vorkommen soll – durch den Term „t0“ zu ersetzen.



Beispiele für die Benutzung:
Wert(x^2,2) liefert den neuen Term 4.
Wert(x+x,y-1) liefert den neuen Term 2y-2.
Wert(x^2,y-1) liefert den neuen Term $(y-1)^2$.
Wert($(z-1)^3$,y+2) liefert den neuen Term $(y+1)^3$.

Wie kann diese Funktion „Wert“ nun in DERIVE realisiert werden? Hier der „Trick“:

```
WERT(t, t0) :=              lim              t
                ELEMENT(VARIABLES(t), 1)→t0

h(f, y1, y2) := WERT(f, y1) + WERT(f, y2)

   2
h(x , 2, 3)

13

         2
h((x - 1) , z + 1, z - 1)

   2
2·z  - 4·z + 4
```

Implementierung der Funktionen LS und RS

Mit der Hilfsfunktion „Wert“ können die Rechtecksummenberechnungen jetzt so gestaltet werden, dass die zu Grunde liegende Funktion auch als Argument übergeben werden kann. Für die Implementierung des Bausteins „Rechtecksummen und Rechteckfiguren“ ist entscheidend, dass es mit der Hilfsfunktion „Wert“ gelingt, die benötigten Funktionswerte „$f(s_i)$“ zu bestimmen.

```
Wert(T, T0) :=              lim              T
                ELEMENT(VARIABLES(T), 1)→T0

                 b - a
d(a, b, n) := ———————
                  n

s(a, b, n, i) := a + i·d(a, b, n)

                    n - 1
LS(fx, a, b, n) :=   Σ   d(a, b, n)·Wert(fx, s(a, b, n, i))
```

$$RS(fx, a, b, n) := \sum_{i=0}^{n} d(a, b, n) \cdot Wert(fx, s(a, b, n, i))$$

Didaktisch-methodische Hinweise

Funktionen höherer Ordnung sind interessant für den Mathematikunterricht. Anhand dieser Funktionen kann man sehr gut aufzeigen, dass Funktionen als eigenständige Objekte betrachtet werden können. Sie können – wie Zahlen auch – anderen Funktionen als Argumente übergeben werden. Eine Computer unterstützte Einführung in diese Thematik bietet sich an. Allerdings ist DERIVE – wie oben beschrieben – nicht sonderlich geeignet, mit Funktionen höherer Ordnung zu arbeiten. Hier gibt es bessere Möglichkeiten. Vor allem funktionale Programmiersysteme wie z. B. Caml sehen das Konzept „Funktion höherer Ordnung“ als eines ihrer Grundkonzepte vor. Die didaktischen Möglichkeiten solcher Systeme im Mathematikunterricht sind noch nicht hinreichend untersucht. Eine ad-hoc Benutzung im Unterricht bereitet meines Erachtens durch die ungewohnten Syntax Schwierigkeiten. Mehr zur funktionalen Programmierung (mit Caml) findet man in [Becker 1999].

8. Mit Listen Daten modellieren

Zielsetzung

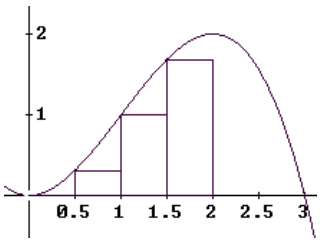
Der Baustein „Rechtecksummen und Rechteckfiguren“ stellt dem Benutzer u. a. Funktionen zum Zeichnen von Rechteckfiguren zur Verfügung. Hier soll jetzt untersucht werden, wie diese Funktionen „gemacht“ sind.

Wir geben zunächst noch einmal die Schnittstellenbeschreibung der interessierenden Funktion(en) wieder.

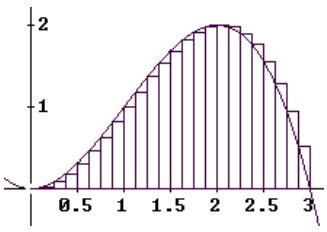
Schnittstellenbeschreibung: LRechteckfigur
Die Funktion „LRechteckfigur“ erzeugt die Rechteckfigur, die sich an den Funktionswerten der linken Intervallgrenzen der Streifen orientieren.

	LRechteckfigur
Funktionsterm →	Term
untere Grenze →	Bruchzahl
obere Grenze →	Bruchzahl
Anzahl der Unterteilungen →	natürliche Zahl
	Bruchzahl → (linke) Rechteckfigur

Beispiele für die Benutzung:
LRechteckfigur(-0.5x^3+1.5x^2, 0, 2, 4) erzeugt eine Rechteckfigur, die im Schaubild wie folgt aussieht:



LRechteckfigur(-0.5x^3+1.5x^2, 0, 3, 24) erzeugt eine Rechteckfigur, die im Schaubild wie folgt aussieht:



Analog hierzu wird die Funktion „RRechteckfigur“ zur Beschreibung der „rechten“ Rechteckfigur spezifiziert.

Die Schwierigkeit bei der Implementierung dieser Funktionen besteht darin, dass zunächst Datenmodelle für die interessierenden geometrischen Objekte (hier die Rechteckfiguren) erstellt werden müssen, bevor diese dann von DERIVE visualisiert werden können. Die

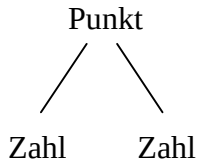
folgende Datenmodellierung erfolgt zunächst sehr allgemein und benutzt dabei die in der Informatik übliche Datenstrukturen Verbund und Sequenz.

Informatische Konzepte: *Datenstrukturen Verbund und Sequenz*

Ein ***Verbund*** fasst eine feste Anzahl von Objekten zu einer Einheit zusammen.
Eine ***Sequenz*** fasst eine variable Anzahl von Objekten zu einer Einheit zusammen.

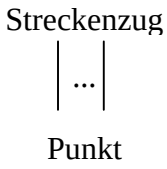
Datenmodellierung

Ein *Punkt* ist ein Verbund bestehend aus zwei Zahlen, der x-Koordinate und der y-Koordinate.



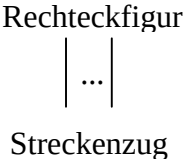
Beispiel:
Der Verbund „(3, 5)“ beschreibt den Punkt mit den Koordinaten (3|5).

Ein Rechteck wird über einen Streckenzug modelliert. Ein *Streckenzug* ist eine Sequenz von Punkten (bei der der erste Punkt identisch mit dem letzten Punkt der Sequenz ist).



Beispiel:
Die Sequenz „(0,0)(4,0)(4,2)(0,2)(0,0)“ beschreibt einen (geschlossenenen) Streckenzug, der ein Rechteck darstellt.

Eine *Rechteckfigur* ist eine Sequenz von Streckenzügen (die aneinanderpassen).



Beispiel:
Die Sequenz „(0,0)(1,0)(1,2)(0,2)(0,0) (1,0)(2,0)(2,4)(1,4)(1,0) (2,0)(3,0)(3,8)(2,8)(2,0)“ beschreibt eine Rechteckfigur bestehend aus drei Rechtecken.

Diese Datenmodelle müssen jetzt mit Hilfe von DERIVE realisiert werden. DERIVE stellt uns hierzu die Datenstruktur „Liste“ zur Verfügung.

Das Listenkonzept von DERIVE

Eine Liste ist (in DERIVE) eine endliche Folge von „Objekten“ (Zahlen, Terme, Listen, ...).

Beispiele:

[1, 2, 3, 4]
[1, x, x^2, x^3]
[ ]
[[1.0, 2.3], [x, x^2]]

Die Erzeugung von Listen kann auf unterschiedliche Weise erfolgen:

a) durch Aufzählung der Listenelemente
Beispiel: [1, 2, 3]

b) durch Generierung der Listenelemente mit Hilfe des VECTOR-Operators
Beispiele: $\underbrace{\text{VECTOR}(x\text{^}n, n, 0, 4, 1)}_{[x\text{^}0, x\text{^}1, x\text{^}2, x\text{^}3, x\text{^}4]}$, $\underbrace{\text{VECTOR}(x * n, n, 0, 1, 0.25)}_{[x*0, x*0.25, x*0.5, x*0.75, x*1]}$

Der VECTOR-Operator arbeitet dabei wie folgt:

Schnittstellenbeschreibung: VECTOR

	VECTOR	
x*n →	Ausdruck	
n →	Laufvariable	
0 →	Anfangswert	
1 →	Endwert	
0.25 →	Schrittweite	
	Listenobjekt	→ [x*0, x*0.25, x*0.5, x*0.75, x*1]

Die folgende Darstellung zeigt, wie man eine Rechteckfigur mit Hilfe von DERIVE aufbaut.

Rechteck – vereinfachte Darstellung

Punkte:
LU (links unten)
RU (rechts unten)
...
	`LU := [1, 0]`
	`RU := [2, 0]`
	`RO := [2, 1]`
	`LO := [1, 1]`

Rechteck als Streckenzug	`Rechteck := [LU, RU, RO, LO, LU]`

Rechteckfigur – vereinfachte Darstellung

Punkte:
LU (links unten)
RU (rechts unten)
...
	`LU(i) := [i, 0]`
	`RU(i) := [i+1, 0]`
	`RO(i) := [i+1, f(i)]`
	`LO(i) := [i, f(i)]`

Rechteck als Streckenzug	`Rechteck(i) := [LU(i), RU(i), RO(i), LO(i), LU(i)]`
Rechteckfigur	`VECTOR(Rechteck(i), i, 1, n)`

Implementierung der Rechteckfiguren

Abschließend listen wir die DERIVE-Funktionen des Bausteins „Rechteckfiguren“ auf. Damit die Punkte zu einem Streckenzug verbunden werden, muss man den Punkt-Status „connected“ einstellen.

```
Wert(T, a) :=          lim          T
            ELEMENT(VARIABLES(T), 1)→a

                b - a
d(a, b, n) := -------
                 n

s(a, b, n, i) := a + i·d(a, b, n)

LU1(fx, a, b, n, i) := [s(a, b, n, i - 1), 0]

RU1(fx, a, b, n, i) := [s(a, b, n, i), 0]

LO1(fx, a, b, n, i) := [s(a, b, n, i - 1), Wert(fx, s(a, b, n, i - 1))]

RO1(fx, a, b, n, i) := [s(a, b, n, i), Wert(fx, s(a, b, n, i - 1))]

URechteck(fx, a, b, n, i) := [LU1(fx, a, b, n, i), RU1(fx, a, b, n, i), R
O1(fx, a, b, n, i), LO1(fx, a, b, n, i), LU1(fx, a, b, n, i)]

LU2(fx, a, b, n, i) := [s(a, b, n, i - 1), 0]

RU2(fx, a, b, n, i) := [s(a, b, n, i), 0]

LO2(fx, a, b, n, i) := [s(a, b, n, i - 1), Wert(fx, s(a, b, n, i))]

RO2(fx, a, b, n, i) := [s(a, b, n, i), Wert(fx, s(a, b, n, i))]

ORechteck(fx, a, b, n, i) := [LU2(fx, a, b, n, i), RU2(fx, a, b, n, i), R
O2(fx, a, b, n, i), LO2(fx, a, b, n, i), LU2(fx, a, b, n, i)]

LRechteckfigur(fx, a, b, n) := VECTOR(URechteck(fx, a, b, n, i), i, 1, n)

RRechteckfigur(fx, a, b, n) := VECTOR(ORechteck(fx, a, b, n, i), i, 1, n)
```

Ein Aufruf der Funktion „LRechteckfigur“ erzeugt ein Datenmodell der Rechteckfigur.

$$\text{LRechteckfigur}\left(-0.5\cdot x^3+1.5\cdot x^2,\ 0,\ 2,\ 4\right)$$

$$\left[\begin{bmatrix}0 & 0\\ \frac{1}{2} & 0\\ \frac{1}{2} & 0\\ 0 & 0\\ 0 & 0\end{bmatrix},\ \begin{bmatrix}\frac{1}{2} & 0\\ 1 & 0\\ 1 & \frac{5}{16}\\ \frac{1}{2} & \frac{5}{16}\\ \frac{1}{2} & 0\end{bmatrix},\ \begin{bmatrix}1 & 0\\ \frac{3}{2} & 0\\ \frac{3}{2} & 1\\ 1 & 1\\ 1 & 0\end{bmatrix},\ \begin{bmatrix}\frac{3}{2} & 0\\ 2 & 0\\ 2 & \frac{27}{16}\\ \frac{3}{2} & \frac{27}{16}\\ \frac{3}{2} & 0\end{bmatrix}\right]$$

Dieses Datenmodell kann jetzt von DERIVE veranschaulicht werden. Es liefert die oben wiedergegebene Darstellung.

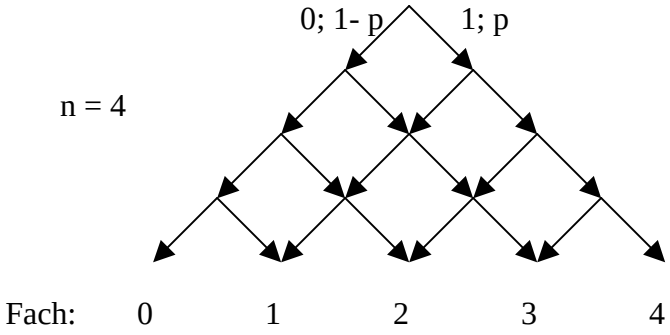
Didaktische und methodische Hinweise

Für die Erstellung von Grafiken mit DERIVE ist wesentlich zu erkennen, dass die geometrischen Objekte mittels Datenstrukturen konstruiert werden müssen, bevor sie von DERIVE gezeichnet werden können. DERIVE stellt hierzu u. a. die Datenstruktur „Liste“ zur Verfügung. Inwieweit solche Aspekte im Mathematikunterricht thematisiert werden müssen, hängt von den jeweiligen Zielsetzungen ab. Meines Erachtens reicht es in vielen Fällen aus, den Schülerinnen und Schülern ein fertiges Zeichenmodul vorzugeben. In diesem Fall können die Implementierungsdetails unberücksichtigt bleiben. Es gibt aber auch Unterrichtseinheiten, in denen gerade die Erstellung der zu zeichnenden Figuren im Vordergrund steht. Hier sollte dann auch die DERIVE-Realisierung diskutiert werden.

9. Listen auswerten

Beispiel 6

Ziel ist es, ein Galton-Brett zu simulieren. Die folgende Abbildung zeigt ein mögliches Galton-Brett mit 4 Stufen. Die Wahrscheinlichkeit für eine Bewegung nach rechts werde durch eine reelle Zahl $p \in]0; 1[$ dargestellt. Die simulierten Kugeln „wählen“ einen Weg durch das Galton-Brett und fallen in eines der Fächer, die von 0 bis 4 (allg. n) durchnumeriert sind.



Der erste Schritt besteht darin, die bei der durchzuführenden Simulation anfallenden Daten zu modellieren.

Datenmodellierung

Wir betrachten ein n-stufiges Galton-Brett.

links/rechts-Entscheidung:
Eine links/rechts-Entscheidung wird durch eine 0 („links“) bzw. 1 („rechts“) kodiert.

Kugelweg:
Ein Kugelweg ist eine n-elementige Sequenz von links/rechts-Entscheidungen.
Listendarstellung: z. B. [0, 1, 1, 1]

Fach:
Ein Fach wird durch eine natürliche Zahl (seine Nummer) aus dem Bereich 0 .. n kodiert; z. B. 3.

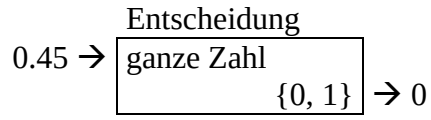
Galtonsimulation:
Eine Galtonsimulation ist eine m-elementige Sequenz von Fachnummern. Diese repräsentieren die Fächer, in die m Kugeln gefallen sind.
Listendarstellung: z. B. [3, 2, 3, 3, 0, 4, 2, 3]

Häufigkeitsverteilung:
Eine Häufigkeitsverteilung ist eine Sequenz von Zahlenpaaren, die für jede Fachnummer die relative Häufigkeit beschreibt.
Bsp.: (0, 0.1) (1, 0.2) (2, 0.2) (3, 0.4) (4, 0.1). Hier ist der Fall dargestellt, dass von 10 Kugeln eine in Fach 0, zwei in Fach 1 usw. fallen.
Listendarstellung:
[[0, 0.1], [1, 0.2], [2, 0.2], [3, 0.4], [4, 0.1]]

Funktionen zur Simulation eines Galtonbretts

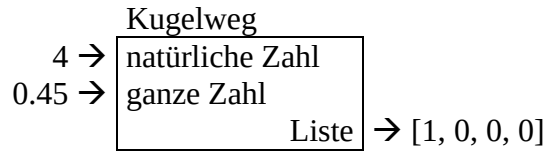
Wir spezifizieren jetzt Funktionen, mit deren Hilfe die oben modellierten Daten erzeugt werden sollen.

Schnittstellenbeschreibung: Entscheidung



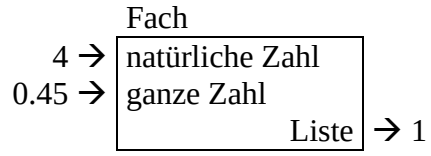
Beispiel für die Benutzung:
Entscheidung(0.45) erzeugt eine 1 mit der Wahrscheinlichkeit 0.45 und eine 0 mit der Wahrscheinlichkeit 0.55.

Schnittstellenbeschreibung: Kugelweg



Beispiel für die Benutzung:
Kugelweg(4, 0.45) erzeugt einen Folge von 4 rechts/links-Entscheidungen, wobei eine rechts-Entscheidung mit der Wahrscheinlichkeit 0.45 getroffen wird.

Schnittstellenbeschreibung: Fach

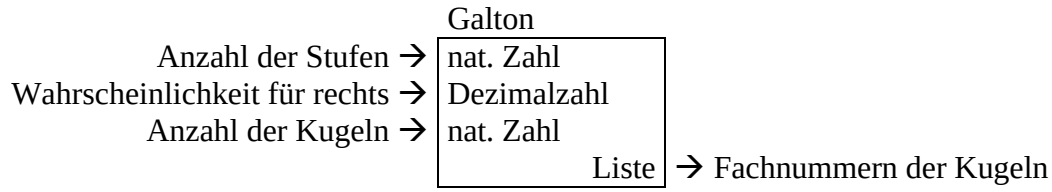


Beispiel für die Benutzung:
Fach(4, 0.45) erzeugt einen Folge von 4 rechts/links-Entscheidungen (z. B. [1, 0, 0, 0]) und bestimmt die Nummer des Fachs (hier 1), in die die Kugel fällt.

Schnittstellenbeschreibung: Anzahl

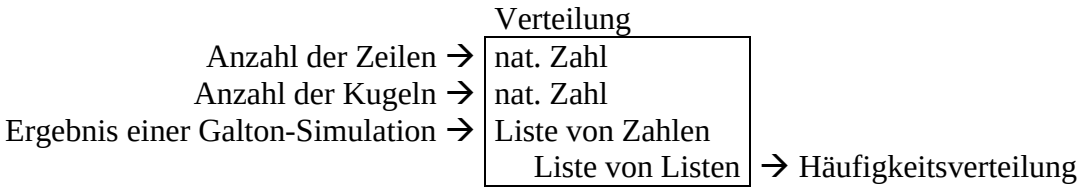
Beispiel für die Benutzung:
Anzahl(3, [3, 2, 3, 3, 5, 4, 2, 3]) bestimmt, wie oft die Fachnummer 3 im Simulationsergebnis [3, 2, 3, 3, 5, 4, 2, 3] vorkommt; hier erhält man also also die Zahl 4.

Schnittstellenbeschreibung: Galton



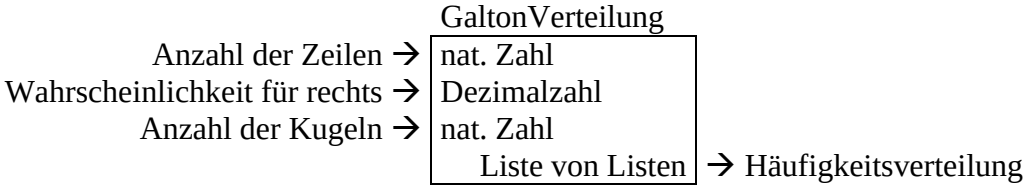
Beispiele für die Benutzung:
Galton(4, 0.1, 100) liefert die Fachnummern von 100 simulierten Kugeln beim Durchlauf durch ein 4-stufiges Galtonbrett mit der Rechts-Wahrscheinlichkeit 0.1.

Schnittstellenbeschreibung: Verteilung



Beispiele für die Benutzung:
Verteilung(5, 8, [3, 2, 3, 3, 5, 4, 2, 3]) → [[0, 0], [1, 0], [2, 2/8], [3, 4/8], [4, 1/8], [5, 1/8]]

Schnittstellenbeschreibung: GaltonVerteilung

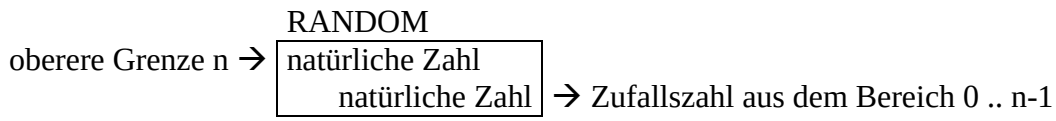


Beispiele für die Benutzung:
GaltonVerteilung(4, 0.5, 15) → [[0, 1/15], [1, 3/15], [2, 6/15], [3, 5/15], [4, 0]]
Also: GaltonVerteilung(4, 0.5, 15) erzeugt eine Galton-Simulation und wertet sie direkt aus. Das Ergebnis ist eine Häufigkeitsverteilung, die beschreibt, wie viele Kugeln sich in den jeweiligen Fächern befinden.

Erzeugung von Zufallszahlen

Zur Simulation von Entscheidungsprozessen benötigen wir eine Operation, die Zufallszahlen erzeugt. Dies leistet in DERIVE die Operation „RANDOM“.

Schnittstellenbeschreibung: RANDOM



Beispiele für die Benutzung:
RANDOM(6) liefert eine Zufallszahl aus dem Bereich [0, ..., 5].
RANDOM(6)+1 liefert eine Zufallszahl aus dem Bereich [1, ..., 6].

Operationen zur Verarbeitung von Listen

Zur Auswertung der Simulationsdaten werden weitere Operationen benötigt, mit deren Hilfe man Listen verarbeiten kann. Die von DERIVE zur Verfügung gestellten Listenoperationen werden zunächst hier kurz vorgestellt.

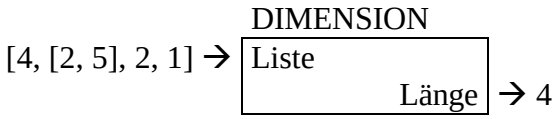
a) Zugreifen auf die Listenelemente

Schnittstellenbeschreibung: ELEMENT

Beispiel für die Benutzung:
ELEMENT([4, 3, 2, 1], 2) beschreibt das Element der gegebenen Liste [4, 3, 2, 1], das an Position 2 in der Liste steht; hier also die Zahl 3.

b) Länge der Liste / Anzahl der Elemente

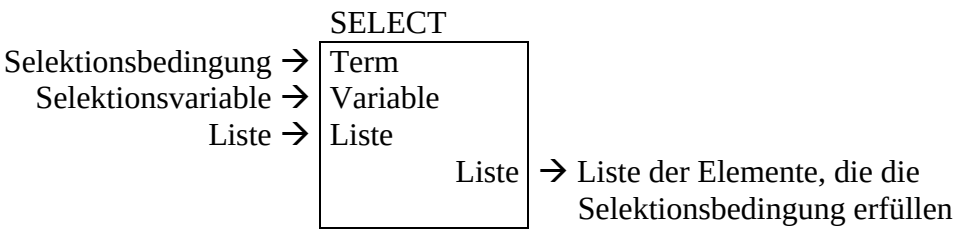
Schnittstellenbeschreibung: DIMENSION



Beispiel für die Benutzung:
DIMENSION([4, [2, 5], 2, 1]) beschreibt die Anzahl der Elemente der gegebenen Liste; hier also die Zahl 4.

c) Auswahl von Listenelementen

Schnittstellenbeschreibung: SELECT



Beispiele für die Benutzung:
SELECT(x = 1, x, [0, 1, 1, 0, 0]) beschreibt die Liste [1, 1].
SELECT(x > 0, x, [0, 1, 2, -3, 5]) beschreibt die Liste [1, 2, 5].

Implementierung der Funktionen

Wir sind jetzt in der Lage, die oben spezifizierten Funktionen mit DERIVE zu implementieren.

```
Entscheidung(p) := IF(RANDOM(100) + 1 ≤ 100·p, 1, 0)

Kugelweg(n, p) := VECTOR(Entscheidung(p), i, 1, n)

Fach(n, p) := DIMENSION(SELECT(x = 1, x, Kugelweg(n, p)))

Galton(n, p, m) := VECTOR(Fach(n, p), i, 1, m)

Anzahl(k, sim) := DIMENSION(SELECT(x = k, x, sim))
```

$$\text{Verteilung}(n, m, sim) := \text{VECTOR}\left(\left[k, \frac{\text{Anzahl}(k, sim)}{m}\right], k, 0, n\right)$$

```
GaltonVerteilung(n, p, m) := Verteilung(n, m, Galton(n, p, m))
```

Diese DERIVE-Funktionen können jetzt wie folgt zur Simulation eines Galton-Bretts benutzt werden:

```
GaltonVerteilung(5, 0.1, 10)
```

$$\begin{bmatrix} 0 & \frac{7}{10} \\ 1 & \frac{1}{5} \\ 2 & \frac{1}{10} \\ 3 & 0 \\ 4 & 0 \\ 5 & 0 \end{bmatrix}$$

```
GaltonVerteilung(5, 0.1, 1000)
```

$$\begin{bmatrix} 0 & 0.589 \\ 1 & 0.333 \\ 2 & 0.071 \\ 3 & 0.006 \\ 4 & 0.001 \\ 5 & 0 \end{bmatrix}$$

Didaktisch-methodische Bemerkungen

Die Bedeutung von Computer erzeugten Simulationen im Stochastikunterricht wird im nächsten Abschnitt diskutiert. Hier sollen die Probleme, die bei einer Verwirklichung mit DERIVE auftreten, kurz angesprochen werden.

DERIVE stellt mit seinem Zufallsgenerator und dem Listenkonzept ein sehr mächtiges und recht einfach zu handhabendes Werkzeug zur stochastischen Simulation zur Verfügung: Listen ermöglichen es, eine Vielzahl von Simulationsergebnissen zu erzeugen. Will man die weitere Verarbeitung der Simulationsergebnisse automatisiert durchführen – dies ist bei einer sehr großen Anzahl solcher Ergebnisse unerlässlich –, so kommen zwangsläufig die oben eingeführten Listenoperationen ins Spiel. Der Umgang mit diesen Operationen erfordert zunächst etwas Übung, bereitet Schülerinnen und Schülern dann aber keine großen Schwierigkeiten.

10. Simulation als Problemlösestrategie

Computer basierte Simulationen ersetzen zunehmend reale Experimente – vor allem dann, wenn diese sehr aufwendig, teuer oder gefährlich sind. Simulation kann somit in vielen Bereichen bereits als gleichrangige Problemlösestrategie angesehen werden. Wir zeigen im Folgenden, wie man mit Simulation ein Problem der Stochastik lösen kann, das mit den herkömmlichen Mitteln des Mathematikunterrichts kaum zu lösen ist.

Beispiel 7
(vgl. [Engel 1973], S. 111f)

Problem:
Ein Würfel wurde 120 mal mit folgendem Ergebnis geworfen:

Augenzahl	1	2	3	4	5	6
beobachtete Häufigkeit	15	21	25	19	14	26

Es stellt sich die Frage, ob dieser Würfel fair ist. Zur Lösung dieses Problems wird zunächst ein Maß für die Abweichung vom „idealen“ Würfelergebnis eingeführt.

Wir betrachten zunächst den allgemeinen Fall: Ein Zufallsexperiment habe s mögliche Ergebnisse, die mit den Wahrscheinlichkeiten p_1, ..., p_s eintreten. Wird das Experiment n-mal wiederholt, dann sind für die einzelnen Ergebnisse die Häufigkeiten np_1, ..., np_s zu erwarten. In Wirklichkeit werden die Häufigkeiten X_1, ..., X_s beobachtet. Als Maß für die Abweichung zwischen den eingetretenen Häufigkeiten X_i und den erwarteten Häufigkeiten np_i verwendet man die folgende Größe

$$c^2 = \sum_{i=1}^{s} \frac{(X_i - np_i)^2}{np_i}$$ (gelesen: Chi-Quadrat).

Man berechnet also die Differenzen der eingetretenen zu den erwarteten Häufigkeiten, quadriert diese Differenzen, teilt sie durch die erwartete Häufigkeit und summiert sie alle auf. Im oben gezeigten Beispiel ergeben sich die folgenden Werte:

Augenzahl i	1	2	3	4	5	6
beobachtete Häufigkeit X_i	15	21	25	19	14	26
erwartete Häufigkeit	20	20	20	20	20	20
Differenz^2 / erw. Haufigkeit	$(-5)^2/20$	$(+1)^2/20$	$(+5)^2/20$	$(-1)^2/20$	$(-6)^2/20$	$(+6)^2/20$

Man erhält $c^2 = 6.2$ als Maß für die Abweichung vom idealen Würfelergebnis. Um diesen Wert beurteilen zu können, müsste man die Verteilung der Größe c^2 kennen. Hieraus könnte man dann z. B. erschließen, wie wahrscheinlich es ist, dass c^2 einen Wert annimmt, der mindestens 6 beträgt.

Die Bestimmung der Verteilung von c^2 ist nicht so einfach. Wir benutzen jetzt DERIVE, um die Verteilung von c^2 näherungsweise zu bestimmen. Hierzu erzeugen wir wiederholt Würfelserien der Länge 120 und werten sie wie oben gezeigt aus. Bei einer großen Zahl von Wiederholungen kann man dann die Verteilung in etwa bestimmen.

Das folgende DERIVE Protokoll führt die benötigten Hilfsfunktionen ein und testet sie direkt im Anschluss. Statt c^2 benutzen wir die Funktion „Abweichung“. Man beachte, dass wir die Verteilung der c^2-Werte nur im Intervall [0,12[betrachten. Es sind natürlich auch Werte außerhalb dieses Intervalls möglich.

```
n := 120

      1
p := ---
      6

E := [20, 20, 20, 20, 20, 20]

Wuerfeln := RANDOM(6) + 1

Wuerfeln

4

Wuerfelserie := VECTOR(Wuerfeln, i, 1, 120)

Anzahl(k, serie) := DIMENSION(SELECT(x = k, x, serie))

Anzahl(3, Wuerfelserie)

28

Haeufigkeiten(serie) := VECTOR(Anzahl(k, serie), k, 1, 6)

Haeufigkeiten(Wuerfelserie)

[16, 19, 28, 13, 23, 21]

                        2
                 (H - E)
Abweichung(H) := -------
                    20

Abweichung([16, 19, 28, 13, 23, 21])

7

Abweichung([15, 21, 25, 19, 14, 26])

6.2

Testserie(m) := VECTOR(Abweichung(Haeufigkeiten(Wuerfelserie)), i, 1, m)

Testserie(10)

[12.7, 4.7, 4.5, 1.5, 3.7, 4.1, 2.5, 2.9, 2.6, 7.2]

Int(a, b, x) := IF(a ≤ x ∧ x < b, 1, 0)

                      DIMENSION(L)
IntAnzahl(a, b, L) :=      Σ       Int(a, b, ELEMENT(L, i))
                         i=1

IntAnzahl(4, 5, [12.7, 4.7, 4.5, 1.5, 3.7, 4.1, 2.5, 2.9, 2.6, 7.2])

3
```

$$\text{Verteilung}(L) := \text{VECTOR}\left(\left[i, \frac{\text{IntAnzahl}(i - 1, i, L)}{\text{DIMENSION}(L)}\right], i, 1, 12\right)$$

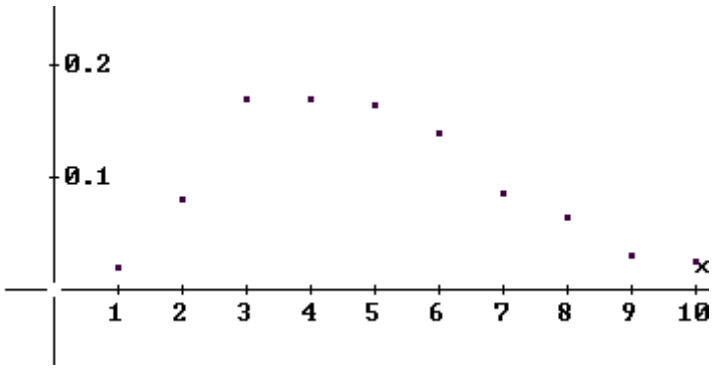
```
Verteilung([12.7, 4.7, 4.5, 1.5, 3.7, 4.1, 2.5, 2.9, 2.6, 7.2])
```

1	0
2	0.1
3	0.3
4	0.1
5	0.3
6	0
7	0
8	0.1
9	0
10	0
11	0
12	0

```
Verteilung(Testserie(200))
```

1	0.02
2	0.08
3	0.17
4	0.17
5	0.165
6	0.14
7	0.085
8	0.065
9	0.03
10	0.025
11	0.015
12	0.005

Die Verteilung der Abweichungen läßt sich mit DERIVE auch graphisch darstellen:



Es handelt sich um eine Näherung der sog. Chi-Quadrat-Verteilung (siehe [Engel 1973]).

Mit den oben per Simulation bestimmten Verteilungswerten ergibt sich:
$P(c^2 \geq 6) = 1 - (P(0 \leq c^2 < 1) + ... + p(P(5 \leq c^2 < 6)) = 0.255$.
Die eingangs aufgelisteten Beobachtungsdaten geben also noch keinen Anlass, an der Echtheit des Würfels zu zweifeln.

Didaktisch-methodische Bemerkungen

Die didaktische Bedeutung von Simulationen im Mathematikunterricht und vor allem im Stochastikunterricht kann nicht hoch genug eingeschätzt werden.

Gerade im Stochastikunterricht vertrauen viele Schülerinnen und Schüler erst in die Richtigkeit von analytisch erzielten Ergebnissen, wenn sie durch eine konkrete Durchführung – hierzu zählen auch Computer-basierte Simulationen – bestätigt werden. Diese Skepsis ist zu begrüßen. Das blinde Vertrauen in Computer-Simulationen dagegen nicht. Wir zeigen im nächsten Abschnitt, wie leicht man Fehler bei einer Modellierung von zufallsbasierten Abläufen machen kann. Weitere Hinweise zur Simulation von Galton-Brettern finden sich in [Schmidt 1985].

Mit Hilfe Computer basierter Simulationen können Probleme gelöst werden, die mit den Mitteln der Schulmathematik sonst nicht bearbeitet werden können. Dies ist von besonderem Interesse, da sich realitätsnahe Problemstellungen sehr oft als außerordentlich schwierig erweisen. In diesen Fällen stellt Simulation dann eine schulgerechte Problemlösestrategie dar. In [Bößer 1999] wird z. B. gezeigt, wie man im Anfangsunterricht zur Wahrscheinlichkeitsrechnung mit Simulation recht anspruchsvolle realitätsnahe Problemstellungen bearbeiten kann.

11. Vereinfachen als Reduzieren

Wir betrachten hier zwei Beispiele, die auf den ersten Blick Probleme bereiten. Eine genauere Analyse des Vereinfachungsschritts, den DERIVE stets automatisch ausführt, hilft diese Probleme zu lösen.

Beispiel 8

In [Grenacher 2000] findet man die folgende „überraschende Reaktion von DERIVE (und des TI 92)“:

$$f(x) := x^2 \cdot t$$

$$f1(x) := \left(\frac{d}{dx}\right)^1 f(x)$$

$$f1(u)$$

$$2 \cdot t \cdot u$$

$$f1(t)$$

$$3 \cdot t^2$$

Überraschend ist das Ergebnis wohl deshalb, weil f1(t) hier nicht $2t^2$ liefert. Dies könnte man erwarten, wenn man im Ergebnis von „f1(u)“ die Konstante „u“ durch die Konstante „t“ ersetzt.

Beispiel 9

Wir betrachten eine zweite Implementierung der Funktion „GaltonVerteilung“ (siehe auch Beispiel 7).

```
Entscheidung(p) := IF(RANDOM(100) + 1 ≤ 100·p, 1, 0)

Kugelweg(n, p) := VECTOR(Entscheidung(p), i, 1, n)

Fach(n, p) := DIMENSION(SELECT(x = 1, x, Kugelweg(n, p)))

Galton(n, p, m) := VECTOR(Fach(n, p), i, 1, m)

Anzahl(k, n, p, m) := DIMENSION(SELECT(x = k, x, Galton(n, p, m)))

GaltonVerteilung(n, p, m) := VECTOR([k, Anzahl(k, n, p, m)/m], k, 0, n)
```

Auf den ersten Blick sieht alles ganz richtig aus. Ein Test dieser Implementierung zeigt aber, dass hier etwas schief gelaufen ist.

```
GaltonVerteilung(5, 0.1, 10)
```

$$\begin{bmatrix} 0 & \frac{7}{10} \\ 1 & \frac{2}{5} \\ 2 & \frac{1}{10} \\ 3 & 0 \\ 4 & 0 \\ 5 & 0 \end{bmatrix}$$

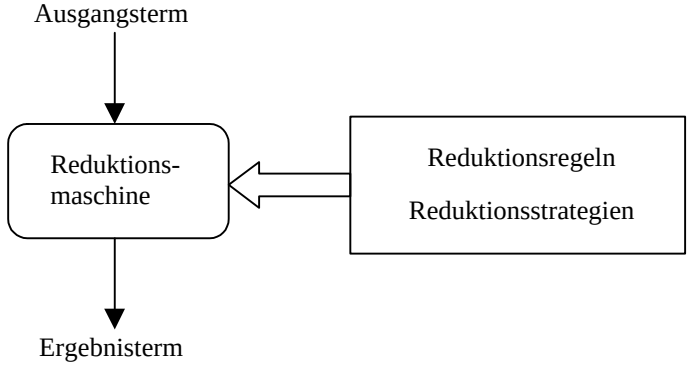
Die Summe der relativen Häufigkeiten ist nicht 1. Woran liegt das?

Zur Klärung dieser Frage und des „überraschenden Ergebnisses“ in Beispiel 8 muss man sich genauer anschauen, wie DERIVE funktionale Ausdrücke auswertet. Wir können zwar nicht „in DERIVE hineinschauen“, wir können aber Berechnungsmodelle entwickeln, mit deren Hilfe man das gezeigte Verhalten erklären kann. Eines der zentralen Berechnungsmodelle der Informatik basiert auf dem Reduktionskonzept. Dieses Konzept soll hier vorgestellt werden. Mit Hilfe dieses Konzeptes kann man genauer verstehen, was im Vereinfachungschritt vor sich geht.

Informatisches Konzept: *Reduktion*

Das Reduktionskonzept lässt sich wie folgt zusammenfassen:
a) Die Definitionen von Funktionen werden als ***Reduktionsregeln*** aufgefasst.
b) Die Berechnung von Funktionswerten erfolgt durch wiederholtes Anwenden der Reduktionsregeln. Man erhält hierdurch ***Reduktionsketten***. Das letzte - nicht mehr reduzierbare - Glied der Kette stellt den errechneten Funktionswert dar.
c) Die Wahl der jeweils nächsten Regelanwendung wird durch ***Reduktionsstrategien*** festgelegt.

Reduktion besteht also im systematischen Vereinfachen eines gegebenen Terms. Wir können uns das so vorstellen, dass eine hypothetische Reduktionsmaschine den Term schrittweise vereinfacht. Die Reduktionsmaschine wird durch Reduktionsregeln und Reduktionsstrategien gesteuert.



Zur Erläuterung des Reduktionskonzepts betrachten wir zunächst einige sehr einfache Funktionsdefinitionen.

```
f(x) := x + 1

g(x) := x – 1

h(x) := f(x) + g(x)
```

Diese Funktionsdefinitionen werden als Reduktionsregeln interpretiert. Wir schreiben sie in Regelform wie folgt:

```
f(x) → x + 1

g(x) → x – 1

h(x) → f(x) + g(x)
```

Berechnungen werden durch Regelanwendungen erzeugt. Wir benutzen außer den zu den Funkionsdefinitionen gehörenden Regeln auch vordefinierte Regeln zur elementaren Arithmetik.

```
f(3) → 3 + 1 → 4

g(f(2)) → g(2 + 1) → g(3) → 3 – 1 → 2

h(f(2)) → f(f(2)) + g(f(2)) → ... → 4 + 2 → 6
```

Eine Funktionsanwendung erfolgt dadurch, dass der Argumentterm in den Funktionsterm an den entsprechenden Stellen eingesetzt wird.

Mit diesem einfachen Konzept lässt sich bereits das „überraschende Ergebnis“ von Beispiel 8 erklären. Zunächst interpretieren wir die Funktionsdefinitionen als Reduktionsregeln:

```
          2
f(x) → x ·t

f1(x) → DIF(f(x),x,1)
```

Den Differenzierungsoperator haben wir zur besseren Verdeutlichung in der DERIVE-Syntax geschrieben. Wir führen jetzt die beiden oben wiedergegebenen Berechnungen mit Hilfe von Regelanwendungen aus:

```
                                  2
f1(u) → DIF(f(u),u,1) → DIF(u ·t,u,1) → 2·t·u

                                  2                  3            2
f1(t) → DIF(f(t),t,1) → DIF(t ·t,u,1) → DIF(t ,t,1) → 3·t
```

Es zeigt sich, dass DERIVE die Berechnungen ordnungsgemäß durchgeführt hat. Man beachte, dass der zu differenzierende Term zunächst vereinfacht werden muss, bevor der DIF-Operator angewandt werden kann.

Berechnung durch Regelanwendungen bzw. Reduktion ist nicht ganz unproblematisch. Betrachten wir noch einmal die folgende Berechnung:

`g(f(2))` → `g(2 + 1)` → `g(3)` → `3 – 1` → `2`

Hier wird zunächst die Regel zur Berechnung von „f“ angewandt. Man könnte natürlich auch zunächst die Regel zur Berechnung von „g“ auswerten:

`g(f(2))` → `f(2) – 1` → `(2 + 1) – 1` → `3 – 1` → `2`

Man erhält dasselbe Ergebnis. Das scheint selbstverständlich zu sein. Ist das aber immer so? Es ergibt sich die Frage, ob man immer dasselbe Ergebnis erhält, egal in welcher Reihenfolge man die Regeln anwendet? Das folgende Beispiel liefert eine negative Antwort.

`VECTOR(RANDOM(2), i, 1, 3)` → `[RANDOM(2),RANDOM(2),RANDOM(2)]` → `[1,1,0]`

`VECTOR(RANDOM(2), i, 1, 3)` → `VECTOR(1, i, 1, 3)` → `[1,1,1]`

Bei der ersten Reduktionskette wird erst der äußere VECTOR-Operator ausgewertet. Anschließend werden die RANDOM-Ausdrücke ausgewertet. Wir nehmen hier an, dass RANDOM(2) die Werte 1, 1 und 0 liefert.

Bei der zweiten Reduktionskette wird erst der innere RANDOM-Ausdruck ausgewertet. Wir nehmen hier an, dass RANDOM(2) den Wert 1 liefert.

Offensichtlich erhält man hier durch unterschiedliche Auswertungsstrategien zwei verschiedene Ergebnisse. Woran liegt das? Der eigentliche Grund ist in einer Durchbrechung der sogenannten referentiellen Transparenz begründet.

Informatisches Konzept: *referentielle Transparenz*

Der Begriff ***referentielle Transparenz*** beschreibt eine fundamentale Eigenschaft von Termen: Der Wert eines Teilterms hängt nicht vom Auswertungszeitpunkt ab.

Betrachten wir als Beispiel den Term „`f(f(2)) + g(f(2))`“. Der Teilterm „`f(2)`“ kommt hier zweimal vor. Egal in welcher Reihenfolge die beiden Teilterme ausgewertet werden, sie liefern stets dasselbe Ergebnis – hier den Wert „`3`“.

Referentielle Transparenz ist eine Grundeigenschaft, die fast alle DERIVE-Terme haben. Eine Ausnahme bilden Terme, in denen der `RANDOM`–Operator vorkommt. Bei diesem Operator ist es gerade beabsichtigt, dass das Ergebnis einer Auswertung nicht stets gleich ist, sondern vom Auswertungszeitpunkt abhängt. Dies hat im obigen Beispiel zur Folge, dass die `RANDOM`–Teilterme im Term „`[RANDOM(2),RANDOM(2),RANDOM(2)]`“ nicht notwendig alle dasselbe Ergebnis liefern. Beim Umgang mit dem `RANDOM`–Operator ist also Vorsicht geboten.

Reduktionsstrategien von DERIVE

Betrachten wir noch einmal den Term `VECTOR(RANDOM(2), i, 1, 3)`. Wie geht DERIVE mit diesem Term um?

```
VECTOR(RANDOM(2), i, 1, 3)

[0, 0, 1]
```

Offensichtlich wertet DERIVE erst den äußeren VECTOR-Operator aus und vervielfacht den inneren Listenerzeugungsterm.

Es fragt sich, ob es immer so ist, dass DERIVE von außen nach innen auswertet. Das folgende Beispiel gibt hierüber Aufschluss. Wir führen eine Hilfsfunktion „h“ ein und werten mehrfach den Term „`h(RANDOM(2))`“ aus:

```
h(x) := VECTOR(x, i, 1, 3)

h(RANDOM(2))

[0, 0, 0]

h(RANDOM(2))

[0, 0, 0]

h(RANDOM(2))

[1, 1, 1]

h(RANDOM(2))

[1, 1, 1]
```

Auch weitere Tests bestätigen, dass in der erzeugten Liste stets immer dieselben Listenelemente stehen. Hieraus muss man schließen, dass DERIVE hier erst den inneren Ausdruck „`RANDOM(2)`“ auswertet und dann erst die äußere Funktion „h“:

```
h(RANDOM(2)) → h(0) → VECTOR(0, i, 1, 3) → [0, 0, 0]
```

Leider sind die Auswertungsstrategien von DERIVE im Handbuch nicht dokumentiert. Wir sind also auf Experimente wie die obigen angewiesen, um sie herauszufinden.
Herausgefunden haben wir die beiden folgenden Strategien:
- Werte zunächst die innersten Teilausdrücke aus.
- Falls ein VECTOR-Ausdruck vorliegt, werte erst den VECTOR-Operator aus.

Im Normalfall muss man sich zum Glück nicht um solche Auswertungsstrategien kümmern. Wenn allerdings der RANDOM-Operator ins Spiel kommt, muss man diese Strategien u. U. berücksichtigen, um zu einer korrekten Implementierung zu gelangen. Wir zeigen jetzt, warum die in Beispiel 9 aufgelistete Implementierung im Gegensatz zu Beispiel 7 nicht korrekt ist.

Eine Erklärung zur fehlerhaften Implementierung

Mit Hilfe der beiden oben genannten Reduktionsstrategien lässt sich das eingangs gezeigte Verhalten erklären. Wir betrachten zunächst die fehlerhafte Implementierung der Funktion „GaltonVerteilung“ in Beispiel 9:

```
      GaltonVerteilung(5, 0.1, 10)
→     VECTOR([k, Anzahl(k,5,0.1,10)/10],k,0,5)
```

```
→     [[0, Anzahl(0,5,0.1,10)/10],
       [1, Anzahl(1,5,0.1,10)/10],
       [2, Anzahl(2,5,0.1,10)/10],
       [3, Anzahl(3,5,0.1,10)/10],
       [4, Anzahl(4,5,0.1,10)/10],
       [5, Anzahl(5,5,0.1,10)/10]]
→     [[0, DIMENSION(SELECT(x=0,x,Galton(5,0.1,10)))/10],
       [1, DIMENSION(SELECT(x=1,x,Galton(5,0.1,10)))/10],
       [2, DIMENSION(SELECT(x=2,x,Galton(5,0.1,10)))/10],
       [3, DIMENSION(SELECT(x=3,x,Galton(5,0.1,10)))/10],
       [4, DIMENSION(SELECT(x=4,x,Galton(5,0.1,10)))/10],
       [5, DIMENSION(SELECT(x=5,x,Galton(5,0.1,10)))/10]]
→     ...
```

Hier sieht man, was bei ungünstiger Implementierung geschehen kann: Die Funktion „Galton“ wird hier sechsmal aufgerufen. Sie erzeugt jedesmal eine Folge aus 10 Fachnummern. Da diese Folgen jedesmal neu mit Hilfe des Zufallsgenerators erzeugt werden, kann nicht erwartet werden, dass sich die jeweiligen relativen Häufigkeiten zu 1 aufaddieren.

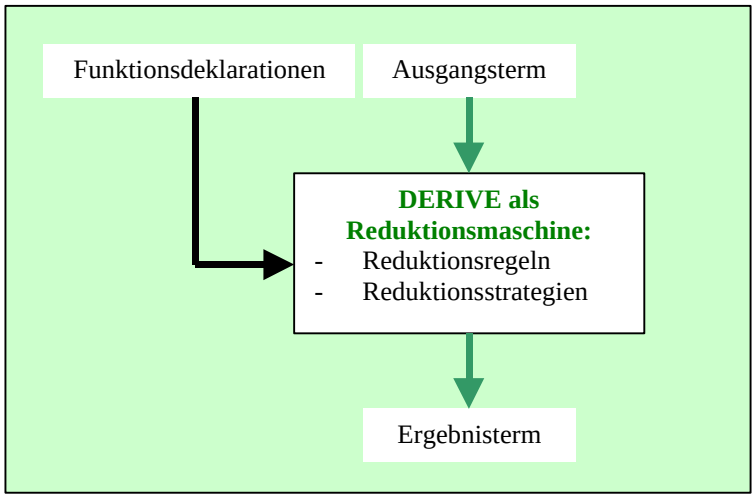
Wir schauen uns als nächstes die korrekte Implementierung der Funktion „GaltonVerteilung“ in Beispiel 7 an.

```
      GaltonVerteilung(5, 0.1, 10)
→     Verteilung(5,10,Galton(5,0.1,10))
→     ...
→     Verteilung(5,10,[0, 0, 0, 0, 0, 2, 0, 1, 1, 1])
→     ...
```

Hier wird zunächst der Teilausdruck „Galton(5,0.1,10)“ ausgewertet. Dieser liefert eine Sequenz von Fachnummern. Erst dann wird die Regel der Funktion „Verteilung“ aufgerufen. Dies führt zu einer korrekten Abfolge der zu leistenden Schritte.

Didaktisch-methodische Bemerkungen

Das Reduktionskonzept liefert uns ein Modell zur Erklärung des Vereinfachungsschrittes:



In einem DERIVE unterstützen Unterricht kann es durchaus zu „merkwürdigen Ergebnissen“ kommen, die dann auch erklärt werden sollten. Hierzu muss genauer untersucht werden, wie

DERIVE den Vereinfachungsschritt durchführt. Die Ausführungen in diesem Abschnitt können hierzu einige Ansatzpunkte liefern.

12. Mit Funktionen programmieren

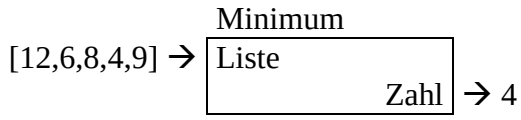
Beispiel 10

Wir betrachten ein klassisches Programmierproblem: Eine Folge (Liste) von Zahlen soll der Größe nach sortiert werden.

Das Sortierproblem hat eine Vielzahl von Lösungen. Die Grundidee des hier dargestellten Sortierverfahren besteht darin, jeweils in der verbleibenden Restliste das kleinste Element zu suchen und es an die erste Stelle der Restliste zu setzen.

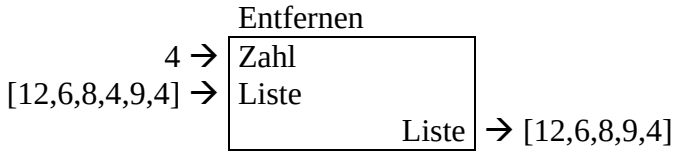
Wir führen geeignete Funktionen ein, die die oben beschriebenen Aufgaben übernehmen sollen. Zunächst werden diese Funktionen genau spezifiziert.

Schnittstellenbeschreibung: Minimum



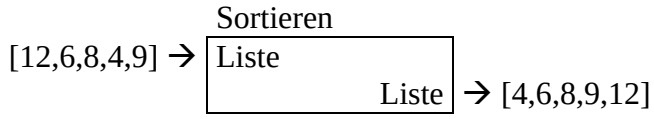
Die Funktion „Minimum“ bestimmt in der eingegebenen Liste das kleinste Element.

Schnittstellenbeschreibung: Entfernen



Die Funktion „Entfernen“ entfernt das eingegebene Element aus der eingegebenen Liste. Kommt das Element mehrfach vor, so wird nur das erste Element entfernt. Kommt das Element überhaupt nicht vor, so bleibt die Liste unverändert.

Schnittstellenbeschreibung: Sortieren



Die Funktion „Sortieren“ ordnet die Elemente der eingegebenen Liste der Größe nach.

Als nächstes müssen die Funktionsdefinitionen entworfen werden. Dies ist im vorliegenden Fall nicht so einfach. Das Verfahren der rekursiven Problemreduktion hilft hier weiter. Es soll zunächst kurz beschrieben werden.

Informatisches Konzept: *Verfahren der rekursiven Problemreduktion*

Die Grundidee ist einfach: Versuche Problemreduktionsschemata zu entwerfen. Eine Strategie besteht darin, das Problem auf ein sich selbst entsprechendes, aber „verkleinertes“ Problem zu reduzieren.

Wir zeigen diese Strategie am Beispiel der Funktion „Minimum“ und entwerfen exemplarische Reduktionsschritte*:*

Minimum([ ]) → ?
Minimum([9]) → 9
Minimum([12, 6, 8, ...]) → Minimum([6, 8, ...])
Minimum([6, 8, 4, ...]) → Minimum([6, 4, ...])

Hieraus kann man durch Verallgemeinerung die folgenden Reduktionsregeln gewinnen:

Minimum([ ]) → ?
Minimum([x]) → x
Minimum([x, y | R]) → Minimum([y | R]), falls x > y
Minimum([x, y | R]) → Minimum([x | R]), falls x ≤ y

Die Schreibweise [x, y | R] soll eine Liste andeuten, die aus einem ersten Element „x“, einem zweiten Element „y“ und einer Restliste „R“ besteht. Die Restliste kann dabei auch leer sein. Eine Liste der Gestalt [x, y | R] enthält also mindestens zwei Elemente.

Wir verfahren analog mit den beiden anderen Funktionen. Es ergeben sich die folgenden Reduktionsregeln:

Entfernen(e, [ ]) → [ ]
Entfernen(e, [x | R]) → R, falls e = x
Entfernen(e, [x | R]) → [x | Entfernen(e, R)], falls e ≠ x

Sortieren(L) → [ ], falls L = [ ]
Sortieren(L) → [Minimum(L) | Sortieren(Entfernen(Minimum(L),L))] , falls L ≠ [ ]

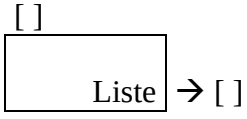
Zur Implementierung dieser Reduktionsregeln mit Hilfe von DERIVE ist es zweckmäßig, zunächst einige weitere Listenoperationen einzuführen. Wir orientieren uns hierbei an den in der funktionalen Programmierung üblichen Operationen.

Informatisches Konzept: *Listenkonstruktoren und Listenselektoren*

Konstruktoren:

leere Liste

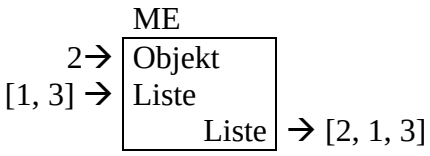
Schnittstellenbeschreibung: LeereListe bzw. [ ]



[ ] erzeugt eine leere Liste.

Hinzufügen eines Elementes

Schnittstellenbeschreibung: MitErstem bzw. ME

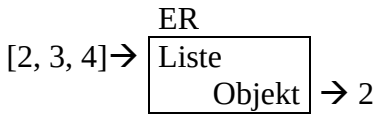


„ME“ fügt ein Objekt in eine Liste ein.

Selektoren:

Listenkopf

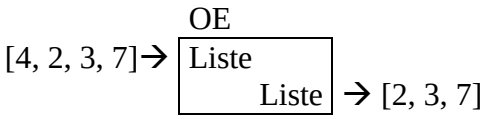
Schnittstellenbeschreibung: Erstes bzw. ER



„ER“ liefert das erste Listenelement (falls die Liste nicht leer ist).

Listenrumpf

Schnittstellenbeschreibung: OhneErstes bzw. OE



„OE“ entfernt das erste Listenelement (falls die Liste nicht leer ist).

Bedeutung der Konstruktoren:
Mit Hilfe der ***Konstruktoren*** läßt sich jede Liste aufbauen bzw. konstruieren.
Bsp.: [4, 3, 1, 7] = ME(4, ME(3, ME(1, ME(7, [ ]))))

Bedeutung der Selektoren:
Mit Hilfe der ***Selektoren*** läßt sich jedes Listenelement aus einer Liste herausgreifen bzw. selektieren.
Bsp.: 1 = ER(OE(OE([4, 3, 1, 7])))

Implementierung der Listenoperationen

```
ER(L) := ELEMENT(L, 1)

ER([2, 3, 4])

2

OE(L) := VECTOR(ELEMENT(L, i), i, 2, DIMENSION(L))
```

```
OE([4, 2, 3, 7])

[2, 3, 7]

ME(e, L) := VECTOR(IF(i = 0, e, ELEMENT(L, i)), i, 0, DIMENSION(L))

ME(2, [1, 3])

[2, 1, 3]
```

Mit Hilfe dieser Listenkonstruktoren und Listenselektoren lassen sich die oben entworfenen Reduktionsregeln sehr einfach in DERIVE-Funktionsdefinitionen übersetzen. Zur besseren Lesbarkeit sind die Definitionen nachträglich umgestaltet worden.

```
Minimum(L) :=
   IF(DIMENSION(L) = 0,
      ?,
      IF(DIMENSION(L) = 1,
         ER(L),
         IF(ER(L) > ER(OE(L)),
            Minimum(OE(L)),
            Minimum(ME(ER(L), OE(OE(L)))))))

Minimum([4, 6, 3, 9, 5, 6, 3, 9])

3

Entfernen(e, L) :=
   IF(DIMENSION(L) = 0,
      L,
      IF(ER(L) = e,
         OE(L),
         ME(ER(L), Entfernen(e, OE(L)))))

[4, 6, 9, 5, 6, 3, 9]

Sortieren(L) :=
   IF(DIMENSION(L) = 0,
      L,
      ME(Minimum(L), Sortieren(Entfernen(Minimum(L), L))))

Sortieren([4, 6, 3, 9, 5, 6, 3, 9])

[3, 3, 4, 5, 6, 6, 9, 9]
```

Informatisches Konzept: *Funktionale Programmierung*

Beispiel 10 zeigt, dass man mit Funktionen (komplexe) Berechnungsverfahren beschreiben kann bzw., dass man mit Funktionen programmieren kann. Ein ***funktionales Programm*** ist dann eine Sequenz von Funktionsdefinitionen. Bei der Konstruktion funktionaler Programme (d. h. bei der Definition von Funktionen) nutzt man die „Konstruktionsprinzipien“ ***Komposition***, ***Fallunterscheidung*** und ***Rekursion*** aus. Eine konkrete Berechnung initiiert man durch einen Funktionsaufruf. Der Funktionswert lässt sich ermitteln, indem man wiederholt Funktionsdefinitionen anwendet. Diese Aufgabe kann von einem geeigneten System (in unserem Fall DERIVE) übernommen werden. Funktionale Programmierung besteht demnach im Modellieren und Erstellen von Funktionsdefinitionen und dem anschließenden Erzeugen von interessierenden Funktionsaufrufen (mehr zur Thematik „funktionale Programmierung“ findet man in [Becker 1999]). Es ergibt sich ein

Programmierkonzept, dass sich wesentlich vom dem gängigen imperativen Konzept unterscheidet. Wir zeigen abschließend die Hauptunterschiede auf.

Ein Problem – zwei Lösungen:

Problem: Bestimmung des größten gemeinsamen Teilers

	imperative Lösung	funktionale (deklarative) Lösung
Modellierung:	{x: 8; y:12; z: ?; r: ?} ggT {x: 8; y: 12; z: 4; r: 0}	ggT (8, 12) → int * int → int → 4

Realisierung:	BEGIN
	REPEAT
	r := x mod y;
	x := y:
	y := r
	UNTIL r = 0;
	z := x
	END
		$ggT(x,y) = \begin{cases} ggT(x-y,y) & \text{falls } x > y \\ x & \text{falls } x = y \\ ggT(x,y-x) & \text{falls } x < y \end{cases}$

Anwendung:	{x: 8; y:12; z:0; r:0} {x: 8; y:12; z:0; r:8} {x: 12; y:8; z:0; r:8} {x: 12; y:8; z:0; r:4} {x: 8; y:8; z:0; r:4} {x: 8; y:4; z:0; r:4} {x: 8; y:4; z:0; r:0} {x: 4; y:4; z:0; r:0} {x: 4; y:0; z:0; r:0} {x: 4; y:0; z:4; r:0}	ggT(8,12) → ggT(8,4) → ggT(4,4) → 4

Worin bestehen die Unterschiede zwischen den beiden Lösungen?

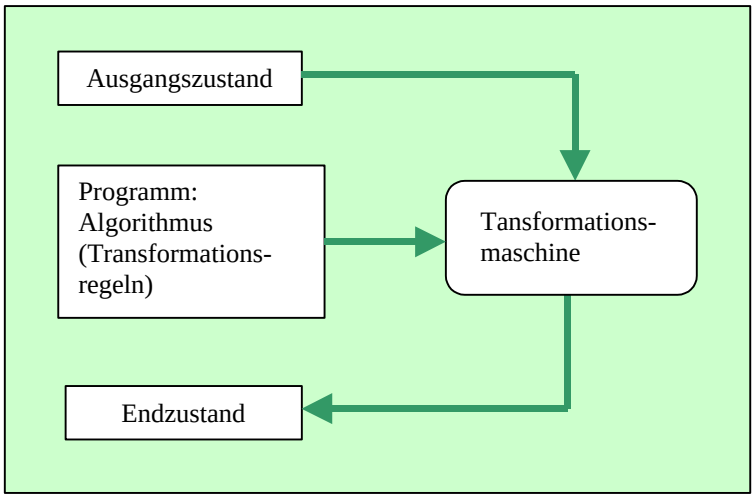
Zunächst unterscheiden sich die *Modellierungskonzepte*. Die imperative Lösung benutzt ein *Zustandskonzept*: Ein Ausgangszustand soll in einen Endzustand überführt werden. Das Modellierungskonzept „Zustand“ stellt eine Abstraktion eines Speichers dar. Die Zustände dienen dazu, Werte von Variablen zwischenzuspeichern. Die funktionale Lösung benutzt dagegen ein *Zuordnungskonzept*. Einem Ausgangsterm wird ein Ergebnisterm zugeordnet.

Die *Realisierungen* weisen bereits größere Unterschiede auf. Die imperative Lösung benutzt ein *Aktionskonzept*. Es werden Aktionen (mittels Anweisungen) festgelegt, die beschreiben, wie die Berechnung des „ggT“ schrittweise vorgenommen werden soll. Die funktionale Lösung benutzt das *Funktionskonzept*. Es wird eine Funktion festgelegt, die die gewünschte Operation beschreibt. Man beachte, dass im imperativen Fall beschrieben wird, wie der „ggT“ berechnet wird, während im funktionalen Fall der „ggT“ über fundamentale Eigenschaften beschrieben wird. Man kann also hier unterscheiden zwischen „Wie-Programmierung“ (***imperative Programmierung***) und „Was-Programmierung“ (***deklarative Programmierung***). Wie-Programmierung ist sehr stark an einer Ausführungsmaschine orientiert. Mittels Anweisungen wird genau festgelegt, wie die (hypothetische) Maschine die Aktionen ausführen soll. Was-Programmierung ist dagegen direkt am Problem orientiert. Mittels Deklarationen (hier: Funktionsdeklarationen) wird die modellierte Operation genau festgelegt.

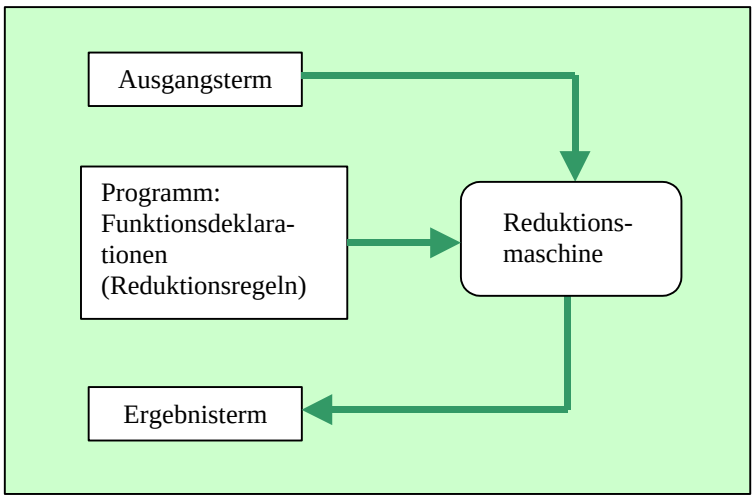
Auch die der *Anwendung* zu Grunde liegenden *Berechnungskonzepte* zeigen fundamentale Unterschiede auf. Die imperative Lösung benutzt das Konzept „*Zustandstransformation*“: Die Aktionen verändern nach und nach den jeweiligen Variablenzustand. Die funktionale Lösung benutzt das Konzept „(zielgerichtete) *Äquivalenzumformung*“: Ein Ausgangsterm wird mit Hilfe von Funktionsanwendungen in einen äquivalenten Endterm überführt.

Die folgenden Abbildungen verdeutlichen noch einmal die Unterschiede zwischen den verschiedenen Programmierformen.

Imperative Programmierung:



Deklarative (hier funktionale) Programmierung:



Wir fassen noch einmal den wesentlichen Unterschied zwischen imperativer und deklarativer Programmierung zusammen: Bei der imperativen Programmierung wird beschrieben, *wie* berechnet werden soll. Bei der deklarativen Programmierung wird beschrieben, *was* berechnet werden soll. Imperative Programmierung ist somit eher *maschinenorientiert*, während deklarative Programmierung eher *problemorientiert* ist.

Didaktisch-methodische Bemerkungen

Programmieren im Mathematikunterricht – muss das sein? Bei der Arbeit mit DERIVE beginnt das Programmieren mit dem Erstellen von Funktionsdeklarationen. Funktionsdeklarationen bilden – wie oben gezeigt – die Programme in der funktionalen Programmierung. Programmieren ist also beim Problemlösen mit DERIVE eine ganz natürliche Sache. Man wird im DERIVE-unterstützten Unterricht wohl kaum auf diese Form der Programmierung verzichten können. Fraglich ist nur, ob man im Mathematikunterricht auch bewusst machen soll, dass es sich hierbei um eine Form der Programmierung handelt. Meines Erachtens ja, sofern man die Zeit dazu hat. Es bietet sich hier die Gelegenheit, fächerverbindend zu unterrichten.

13. Wieviel Informatik braucht man beim Arbeiten mit DERIVE im Mathematikunterricht?

DERIVE kann auf unterschiedlichen Anforderungsniveaus im Mathematikunterricht eingesetzt werden:

I. *Erledigung von Routineaufgaben*: Mit Hilfe von DERIVE-Operatoren lässt sich auf einfache Weise z. B. ein Grenzwert berechnen usw..

II. *Interaktives Problemlösen*: Hierbei müssen in einem abgesteckten Rahmen selbständig Operatoren ausgewählt und benutzt werden und dabei evtl. Hilfsobjekte deklariert werden.

III. *Funktionale Programmierung*: Zur Lösung eines komplexeren Problems werden Daten und Funktionen modelliert und mit Hilfe der Datenstruktur Liste sowie der Kontrollstrukturen Komposition, Fallunterscheidung und Rekursion implementiert.

Wieviel informatisches Grundlagenwissen wird auf den verschiedenen Niveaus benötigt? Niveau I benötigt kaum informatisches Wissen, vielmehr Bedienerwissen. Niveau II setzt auf jeden Fall den Dreischritt „Beschreiben-Vereinfachen-Deuten“ und ein Grundverständnis der Sprachkonstrukte von DERIVE voraus. Je nach Problemkontext müssen weitere Strukturen (z. B. das Listenkonzept) bekannt sein. Die Arbeit auf Niveaustufe II setzt damit ein Grundverständnis von informatischen Strukturen voraus. Niveau III ist eigentlich der Informatik zuzurechnen. Es setzt eine intensive Auseinandersetzung mit den zugrunde liegenden informatischen Strukturen voraus.

Welches Niveau kann im Mathematikunterricht angestrebt werden? Niveau I ist sicher ein häufig anzutreffendes Arbeitsniveau im Unterricht. Meines Erachtens kann man durchaus auch Niveau II anstreben. Niveau III dürfte dagegen eher die Ausnahme sein (und evtl. in Projektphasen erreicht werden). Entscheidet man sich für Niveau II (oder III), so sollten meines Erachtens die benötigten und benutzten informatischen Grundstrukturen im Unterricht mit thematisiert werden. Die Materialien in diesem Papier sollten hierbei eine Hilfestellung leisten.

Ein solcher Einsatz von DERIVE würde zu einer Öffnung des Mathematikunterrichts hinsichtlich der Integration informatischer Konzepte bzw. zu einem informatisch fundierten Mathematikunterrichts führen. Der Unterricht könnte dabei fachübergreifend (Inhalte und Methoden des Fachs Informatik werden integriert) oder auch fächerverbindend (Inhalte und Methoden beider Fächer werden kombiniert) angelegt werden. Meines Erachtens würde hierdurch nicht nur die informatische Seite gestärkt, die mathematische Seite würde auch bereichert. Nach [Weigand 1993] müssen informatische Entwicklungen und Ergebnisse im Mathematikunterricht dann genutzt werden, wenn dadurch neue Sichtweisen mathematischer Inhalte erschlossen werden können. Wie dies bei einem informatisch fundierten DERIVE-Einsatz erfolgen kann, sollte in diesem Beitrag gezeigt werden..

14. Literaturhinweise

[Baumann 1998]
Rüdeger Baumann: Analysis 1, Ein Arbeitsbuch mit Derive, Stuttgart 1998.

[Becker 1999]
Klaus Becker: Funktionale Programmierung, Materialien zum Lehrplan Informatik, Landesmedienzentrum Rheinland-Pfalz 1999.

[Bößer 1999]
Annette Bößer: Propädeutik zum Testen von Hypothesen – Eine Unterrichtsreihe zur experimentellen Behandlung wahrscheinlichkeitstheoretischer Problemstellungen in einem Grundkurs der Jahrgangsstufe 12, Pädagogische Hausarbeit am Studienseminar für Gymnasien Kaiserslautern, 1999.

[Engel 1973]
Arthur Engel: Wahrscheinlichkeitsrechnung und Statistik, Band 1, Klett Studienbücher, Stuttgart 1973.

[Grenacher 2000]
Fritz Grenacher: Eine überraschende Reaktion des Computer-Algebra-Systems TI-92, Praxis der Mathematik 2/2000, S.83.

[Köhler 1995]
Reinhard Köhler: Mathematische Standardsoftware und Informatik – Zur Problematik des Zuweisungsoperators und des Gleichheitszeichens in DERIVE, MNU 48/1995, S. 195-198.

[Materialien 1995]
Materialien zum Mathematikunterricht mit Computer und DERIVE, Landesmedienzentrum Rheinland-Pfalz 1995.

[Noll&Schmidt 1997]
Gregor Noll und Günter Schmidt: Anschaulicher und lebendiger Mathematkunterricht mit dem Werkzeug Computer – Welche Kompetenzen brauchen Lehrerinnen und Lehrer?, MU 2/1997, S. 5-11.

[Schmidt 1985]
Günter Schmidt: Eigenbau und Simulation von Galton-Brettern, mathematik lehren 12/1985, S. 22-30.

[Schmidt&Grabinger&Noll 1998]
Günter Schmidt, Benno Grabinger und Gregor Noll: Entdecken Verstehen Anwenden, Analysisunterricht mit dem TI-92, Texas Instruments 1998.

[Wagenknecht 1999]
Christian Wagenknecht: Informatikfundierter Mathematikunterricht, Praxis der Mathematik 3/1999, S. 124-125.

[Weigand 1993]
Hans-Georg Weigand: Überlegungen zum Verhältnis von Mathematik- und Informatikunterricht, MNU 46/7, S. 428-432.